STROBOSCOPE
State and Resource Based Simulation of Construction Processes

by
Julio César Martínez

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Civil Engineering)
in The University of Michigan
1996

Doctoral Committee:
Associate Professor Photios G. Ioannou, Chair
Professor Robert I. Carr
Assistant Professor Tava M. Lennon
Professor Thomas J. Schriber
Associate Professor Iris D. Tommelein
stro·bo·scope (stro'be-skop’)

Noun

1. Any of various instruments used to make moving objects appear stationary by intermittent illumination or observation.
2. An instrument for studying rapid motion.
3. Acronym for STate and ResOurce Based Simulation of COstruction ProcEsses.

Etymology: Greek strobos, a whirling round + -scope, to see.
To my wife, Nancy Patricia,
my children, Laura Patricia and Nilson Eduardo,
and my parents, Hilda Altagracia and Nilson Emigdio.
Acknowledgments

Many have contributed to this research. Some directly with their ideas, others with financial support and convenient employment, and the closest to my heart with spiritual support and comprehension. I am grateful to all.

I owe special thanks on many accounts to my principal advisor, Professor Photios Ioannou. Well before I started this research, he taught me the state-of-the art in construction process simulation. Our initial discussions motivated me to design and implement Stroboscope. Since then, and throughout the entire effort, he has been an incredible motivator and a friend. His advice, opinions, and experience with UM-CYCLONE and COOPS were an invaluable resource that helped shape this work during the numerous hours that we spent discussing it. I am grateful for his support in promoting Stroboscope, and for adopting Stroboscope as the process simulation tool for his courses well before this work was finished. I will always remember the many late nights and weekends that he dedicated to this research.

The suggestions made by the members of my doctoral committee improved this work significantly. Professor Robert Carr deserves special thanks. His experience with UM-CYCLONE, RESQUE, and CIPROS, enabled him to contribute to many aspects of this work. In addition, his role as program advisor (Construction Engineering and Management) during my doctoral studies served as a leadership example worthy of admiration. Professor Thomas Schriber taught me many aspects of simulation modeling that made their way into this work. Professor Iris Tommelein’s lessons on construction equipment where very helpful in preparing many of the examples used throughout this work. She always had me in mind and forwarded all sorts of information relevant to construction process simulation. Professor Tava Lennon’s work on polling models was a useful aid in verifying Stroboscope.
Many students at the University of Michigan learned early versions of the Stroboscope language while taking CEE 631 during the winter of 1994 and 1995. Some of them made many useful comments that are appreciated. In particular, I would like to express my gratitude to Tariq Abdelhamid, Pedro Angarita, Rasha Stino, Phongsarit Tantisuvanitchkul, Chein-Chih Ting, and Po-Wen Wu for their comments during the winter of 1994; and to Daniel Baima, Alvaro Cabrales, Peng-Jou Chen, Ruei-Guo Dai, Kosuke Hirase, Toshihisa Matsui, Teddy Mertyris, Manuel Moreno, Todd Ockaskis, Michael O'Donnell and Kenji Tanida for their comments during the winter of 1995.

Preliminary versions of the Stroboscope language have been available to researchers outside the University of Michigan since the summer of 1995. Since then, several professors at other universities have adopted Stroboscope as the basis of their construction simulation courses or have used it as a research tool. I am grateful to all of them for their early confidence on this work.

I would like to express my gratitude to the Department of Civil and Environmental Engineering, to the Rackham Pre-Doctoral Fellowship Program at the University of Michigan, and to the National Science Foundation for their financial support.

I am also grateful to Project Management Associates, Inc., for the opportunity to work in their research and development group during the period of my doctoral studies. Special thanks to Dr. Gui Ponce De Leon and John Knoke, who understood the nature of this doctoral endeavor and allowed me to design my own work schedule.

My parents in the Dominican Republic, Hilda and Nilson, have always been a source of motivation and continuous support.

My children, Nilson Eduardo and Laura Patricia, motivated me to set an example. Finally and most importantly, my wife Nancy has been a source of inspiration. They have joined me in times of joy and tolerated my passion for their strongest competitor: Stroboscope. This work is a tribute to them.
Preface

Thank you for opening this dissertation despite its intimidating size. The dissertation describes Stroboscope, a new construction simulation language that allows you to model discrete processes of any complexity with unprecedented ease. In Stroboscope you will find the simplicity characteristic of activity-based simulators such as CYCLONE coupled with the modeling power of general purpose simulation languages.

This dissertation is big for two reasons. The first is that there is a lot to say about Stroboscope. The second is that I wrote it so that it could be read and understood by a wide audience, including people without prior exposure to discrete event simulation.

You do not need to read the entire thesis in order to use Stroboscope. The chapters follow a natural progression that present subsets of the language that can stand on their own. Since each chapter builds upon the previous, however, you should read them in order.

Everything there is to know about the language itself is explained or mentioned here somewhere. There are other things, however, that are not mentioned. I devoted very little space to describe the various programs in the Stroboscope simulation system or to explain how to use them. There are no screen shots, menu hierarchy diagrams, or other types of information typically found in software documentation. The instructions on how to use the programs are available separately in each program’s on-line help. I do not discuss Stroboscope’s internal design and implementation either; there are no software design diagrams or listings containing Stroboscope’s source code.
Table of Contents

Acknowledgments ........................................................................................................ iii
Preface ......................................................................................................................... v
List of Figures ............................................................................................................. xiii
List of Tables .............................................................................................................. xvii
List of Flowcharts .................................................................................................... xviii
List of Appendices .................................................................................................... xix

Chapter

1 Overview..................................................................................................................... 1
2 Introduction............................................................................................................... 4

2.1 Discrete Event Simulation .................................................................................... 4
2.1.1 General-Purpose Programming Languages ...................................................... 5
2.1.2 Simulation-Specific Tools .................................................................................. 5
2.1.3 Review of Selected Activity-Based Simulation Tools ........................................ 7
2.1.3.1 GSP — General Simulation Program ............................................................... 8
2.1.3.2 HOCUS - Hand Or Computer Universal Simulator ......................................... 10
2.1.3.3 CYCLONE - Cyclic Operations Network ....................................................... 12
2.1.3.4 RESQUE ......................................................................................................... 13
2.1.3.5 COOPS .......................................................................................................... 14
2.1.3.6 CIPROS .......................................................................................................... 14
2.1.3.7 Recent Advances in Construction Simulation ............................................... 15

2.2 Research Objectives ............................................................................................. 15
2.3 Methodology - A Chronological Outline of Steps and Results .............................. 16
2.4 Dissertation Outline .............................................................................................. 17

3 Networks ................................................................................................................... 19

3.1 Resources and Resource Types ........................................................................... 19
3.2 Network Elements ........................................................................................................... 20
  3.2.1 Links ............................................................................................................................ 21
  3.2.2 Nodes .......................................................................................................................... 23
    3.2.2.1 Queues .................................................................................................................. 23
    3.2.2.2 Activities ............................................................................................................. 24
  3.3 Networks .......................................................................................................................... 27

4 Basic Simulation Processing ................................................................................................. 31
  4.1 Simulation Clock ............................................................................................................ 31
  4.2 Activity Instance Life-span ............................................................................................ 32
  4.3 The Future Events List ................................................................................................... 32
  4.4 Current Events ............................................................................................................... 34
  4.5 Activity Instance Termination ....................................................................................... 34
  4.6 Combi Instantiation Phase ............................................................................................. 37
  4.7 Clock Advance Phase .................................................................................................... 43
  4.8 The Simulation Loop ...................................................................................................... 44
  4.9 Recap ............................................................................................................................. 46

5 Creating and Running Simple Models .................................................................................. 48
  5.1 Structure of Stroboscope Model Files ........................................................................... 49
    5.1.1 Element Definition Statements ................................................................................. 53
    5.1.2 Attribute Statements ................................................................................................. 56
    5.1.3 Control Statements .................................................................................................... 58
  5.2 Executing a Model and Interpreting the Standard Report ............................................ 60
  5.3 Variables .......................................................................................................................... 65
    5.3.1 Pre-Defined System-Maintained Variables ................................................................. 66
      5.3.1.1 Global Variables .................................................................................................. 67
      5.3.1.2 Instance Variables ............................................................................................... 68
    5.3.2 User-defined Variables ............................................................................................... 69
  5.4 Displaying Custom Output .............................................................................................. 71
  5.5 Logical Expressions, Values and Variables .................................................................... 73
  5.6 Specifying a Stopping Condition for a Simulation ......................................................... 78
  5.7 Source File Organization ................................................................................................. 78

6 Controlling the Amounts of Resource Flows ....................................................................... 86
  6.1 When Can a Queue Support a Combi ? .......................................................................... 86
  6.2 Combi Resource Acquisition .......................................................................................... 88
    6.2.1 Acquiring Resources in One Step ............................................................................. 88
    6.2.2 Acquiring Resources in Several Steps ....................................................................... 89
6.2.3 Detailed Control of Combi Durations .............................................................. 92
6.3 Controlling the Amount of Resources Released .............................................. 94
6.4 Recap .................................................................................................................. 96

7 More Detailed Control of Combi Instantiations .................................................. 97
7.1 Combi Priorities ................................................................................................. 97
7.2 Semaphores ....................................................................................................... 102
7.3 The MCS Flag .................................................................................................. 106
7.4 Multiple CIP Passes .......................................................................................... 108

8 General Programming & Statistical Objects ....................................................... 109
8.1 Class Categories ............................................................................................... 109
8.2 Actions .............................................................................................................. 111
8.3 Timing of Actions ............................................................................................ 112
  8.3.1 Action Control Statements ........................................................................... 113
  8.3.2 Action Events ............................................................................................... 115
  8.3.3 Action Definition Statements ...................................................................... 117
8.4 Classes of Action Targets ................................................................................. 118
  8.4.1 SaveValues .................................................................................................. 119
  8.4.2 Arrays .......................................................................................................... 120
  8.4.3 Collectors ..................................................................................................... 124
  8.4.4 Moving Average Collectors .......................................................................... 125
  8.4.5 Weighted Collectors .................................................................................... 127
  8.4.6 Time-Weighted Collectors ........................................................................... 131
  8.4.7 OutFiles & AppFiles .................................................................................... 134

9 Forks And Dynaforks ........................................................................................... 142
9.1 Forks .................................................................................................................. 142
9.2 Dynaforks ........................................................................................................... 144
9.3 Fork Resolution ................................................................................................ 146
  9.3.1 Fork and Dynafork Chaining ....................................................................... 148
9.4 New Programming Facilities ............................................................................ 151

10 Consolidators ..................................................................................................... 153
10.1 Overview ........................................................................................................... 154
10.2 Consolidator Reactions to Resource Receptions ........................................... 154
10.3 Consolidator Instances .................................................................................... 156
10.4 Contrast to Other Nodes ............................................................................... 157
10.5 Examples ......................................................................................................... 159
10.6 Recap ............................................................................................................... 162
13.4 Examples ................................ ................................ ................................ .................. 252
  13.4.1 Simple Assembly of All Released Resources .............................................................. 254
  13.4.2 Simple Assembly to Base ............................................................................................. 255
  13.4.3 Simple Assembly of Some of the Released Resources ............................................... 256
  13.4.4 Parallel Assembly ........................................................................................................... 257
  13.4.5 Assembly on Top of Parallel Assembly ........................................................................ 257
  13.4.6 Assembly with Components of the Same Compound Type ......................................... 258
  13.4.7 Simple Disassembly ....................................................................................................... 260
  13.4.8 Simple Disassembly With Release Control .................................................................... 260
  13.4.9 Simple Disassembly From Base ..................................................................................... 261
  13.4.10 Generic Resource Disassembly From Base .................................................................. 262
  13.4.11 Chained Disassembly .................................................................................................... 263
  13.4.12 Disassembly With Components of the Same Type as the Container ......................... 264
  13.4.13 Assembling Disassembly Components ........................................................................ 265
  13.4.14 Parallel Dual Base Links .............................................................................................. 266
  13.4.15 A Practical Example ..................................................................................................... 268

13.5 Recap .................................................................................................................................. 270

14 Flow Control Statements ........................................................................................................... 273
  14.1 Ending Model Processing ................................................................................................. 273
  14.2 IF Blocks ............................................................................................................................ 274
  14.3 While-Wend Blocks ........................................................................................................... 276

15 Statement Preprocessing and Automatic Code Generation ................................................. 280
  15.1 The Preprocessor Operator $ <Arg> $ ................................................................................ 280
  15.2 Example With Automatically Generated Code : Shuttle Bus Problem ............................ 282
  15.3 Solution ............................................................................................................................... 283
  15.4 Extended Solution .............................................................................................................. 297
    15.4.1 Extended Solution Code ............................................................................................. 300
    15.4.2 Extended Solution Output ......................................................................................... 309

16 Performing Multiple Replications .......................................................................................... 311
  16.1 Persistent SaveValues and Collectors ............................................................................... 312
  16.2 The Confidence Function .................................................................................................. 312
  16.3 The RESETSTATS statement ......................................................................................... 314
    16.3.1 Batch Means Replications - Generic Format ............................................................ 315
    16.3.2 Simulation Warm-up Example .................................................................................... 317
    16.3.3 Example of Batch Means Replications ...................................................................... 318
  16.4 The CLEAR Statement ....................................................................................................... 325
    16.4.1 Examples of Replications in Terminating Simulations .............................................. 326
16.4.1.1 Replications that Simulate the Same Model with Different Configurations...... 330
16.4.1.2 Replications that Simulate a Model with the Same Parameters Each Time ...... 334
16.4.1.3 Replicating a Parametric Probabilistic CPM Model................................. 338

16.5 Recap ...................................................................................................................... 348

17 Random Number Generator, Stream Management and Variance Reduction..... 349

17.1 Random Number Generator ................................................................................. 349
17.2 The Random Number Streams .......................................................................... 351
  17.2.1 The Default Stream ....................................................................................... 351
  17.2.2 Additional Streams ....................................................................................... 352
  17.2.3 Antithetic Sampling ....................................................................................... 354
  17.2.4 Statistical Considerations .............................................................................. 355
17.3 Variance Reduction Techniques ......................................................................... 355
  17.3.1 Antithetic Variates ....................................................................................... 358
    17.3.1.1 Antithetic Variates Example - Job Site Snack Bar ................................. 359
  17.3.2 Common Random Numbers ......................................................................... 367

18 Extending Stroboscope — Add-Ons .................................................................... 372

18.1 Using Stroboscope Add-Ons .............................................................................. 372
  18.1.1 Probabilistic CPM Add-On for Stroboscope — Reference ........................... 373
    18.1.1.1 CPM Add-On Statements ....................................................................... 374
    18.1.1.2 CPM Add-On Pre-Defined Variables .................................................... 375
    18.1.1.3 CPM Add-On Persistent Collectors ....................................................... 377
  18.1.2 Example Using the CPM Add-On ............................................................... 378
  18.1.3 Additional Methods of Accessing Statements and Functions Defined in Add-Ons 380

18.2 Developing Add-Ons for Stroboscope ............................................................... 382
  18.2.1 Add-On Interface ......................................................................................... 382
    18.2.1.1 Registering Statements ......................................................................... 383
    18.2.1.2 Registering Functions .......................................................................... 384
    18.2.1.3 Registering Variables .......................................................................... 385
    18.2.1.4 Other Functions Exported by Stroboscope ........................................... 386
  18.2.2 CPM Add-On for Stroboscope Source Code .............................................. 390

18.3 Recap .................................................................................................................... 404

19 Conclusion ............................................................................................................. 405

19.1 Fundamental Capabilities Provided by Stroboscope to Support Construction
  Simulation ................................................................................................................ 407
  19.1.1 Access to the State of the Simulation .......................................................... 407
  19.1.2 Resource Characterization .......................................................................... 407
  19.1.3 Compound Resources .............................................................................. 408
19.1.4 Bulk Resources........................................................................................................408
19.1.5 Programmability - Simulation Languages Vs. Simulators .....................................409
19.2 Contributions..............................................................................................................410
19.3 Validation and Implementation.....................................................................................414
19.4 Capabilities, Limitations and Directions for Future Research.................................415

Appendices .........................................................................................................................417

Bibliography .......................................................................................................................504
# List of Figures

Figure 1 - Tocher Wheel-chart With Intersecting Machine Cycles ........................................... 9  
Figure 2 - Tocher Wheel-Chart With Variable Activity Sequence ........................................... 9  
Figure 3 - HOCUS Activity Cycle Diagram .................................................................................. 11  
Figure 4 - CYCLONE Network .................................................................................................... 12  
Figure 5 - Simplified Model of a Bank ......................................................................................... 21  
Figure 6 - Parallel Links Between Withdraw and CustomersLeave .......................................... 22  
Figure 7 - Classic Earth-Moving Operation .................................................................................. 27  
Figure 8 - Simplified Simulation Loop ......................................................................................... 45  
Figure 9 - Output Produced by the REPORT Statement ............................................................. 63  
Figure 10 - Classic Earth-Moving Operation .............................................................................. 80  
Figure 11 - Earth-moving With Two Types of Haulers (Using Generic Resources) .................... 99  
Figure 12 - Eight Hour Clock ...................................................................................................... 104  
Figure 13 - Two Shift Clock ......................................................................................................... 106  
Figure 14 - Routing Haulers With a Fork .................................................................................... 143  
Figure 15 - Routing Haulers With a Dynafork ............................................................................ 145  
Figure 16 - Release Graph With Forks and Dynaforks ................................................................ 148  
Figure 17 - Consolidating with Two Queues and a Combi ....................................................... 159  
Figure 18 - Consolidating With One Queue ............................................................................... 160  
Figure 19 - Consolidating With a Consolidator and a Queue .................................................... 160  
Figure 20 - Preemptable Earth-Moving Operation ..................................................................... 161  
Figure 21 - Characterized Resource Structure .......................................................................... 171  
Figure 22 - Ten Loaders in the LoadersWait Queue .................................................................... 177  
Figure 23 - LoadersWait Cursoring the Loader at Position 9 .................................................... 179  
Figure 24 - Earth-Moving Operation With Scrapers .................................................................. 184  
Figure 25 - Data for Earth-Moving Operation With Scrapers .................................................. 185
Figure 26 - First Loader to Enter LoadersWait

Figure 27 - LoadersWait After the Entry of the First Loader

Figure 28 - Second Loader to Enter LoadersWait

Figure 29 - LoadersWait After the Entry of the Second Loader

Figure 30 - Third Loader to Enter LoadersWait

Figure 31 - LoadersWait After the Entry of the Third Loader

Figure 32 - Complex Release Graph

Figure 33 - Network Fragment for Bus Stop

Figure 34 - Commuter Transportation System

Figure 35 - Commuter Lot 2 (Bus Stop 4)

Figure 36 - Work Area 3 (Bus Stop 5)

Figure 37 - Population Graph

Figure 38 - Commuter Lot 2 (Bus Stop 4) With Minimum Waiting Time

Figure 39 - Work Area 3 (Bus Stop 5) With Minimum Waiting Time

Figure 40 - Commuter Transportation System With 2 Buses

Figure 41 - Population Graph With Two Small Buses

Figure 42 - Population Graph With 2 Small Buses and Statistical Reset at 12 Hours

Figure 43 - Earth-Moving Operation With Scrapers

Figure 44 - CPM Network for Construction of Fast Food Outlet

Figure 45 - Representation of CPM Activity # 9 "Counter"

Figure 46 - Circular Random Number Chain

Figure 47 - Snack Bar Network

Figure 48 - Random Number Scheme in the Snack Bar Model

Figure 49 - CYCLONE Network

Figure 50 - CYCLONE Earth-moving Operation With 2 Types of Haulers

Figure 51 - CYCLONE Network Fragment With Clock

Figure 52 - RESQUE Model Fragment

Figure 53 - Stroboscope Network Fragment Analogous to Figure 52

Figure 54 - About to Start CIP # 1

Figure 55 - CIP #1 After the Instantiation of Load(0)

Figure 56 - CAP #1 Just Before the Termination of Load(0)
Figure 57 - CAP #1 After the Termination of Load(0) .................................................. 485
Figure 58 - About to Start CIP #2 ................................................................. 485
Figure 59 - CIP #2 Just After the Instantiation of Load(1) ....................................... 486
Figure 60 - About to Terminate Load(1) in CAP #2 ............................................. 486
Figure 61 - CAP #2 After the Termination of Load(1) ............................................ 487
Figure 62 - About to Start CIP #3 ................................................................. 487
Figure 63 - About to Terminate Haul(1) in CAP #3 ............................................. 488
Figure 64 - CAP #3 After the Termination of Haul(1) ............................................ 488
Figure 65 - About to Start CIP #3 ................................................................. 489
Figure 66 - About to Terminate Dump(0) in CAP #4 .......................................... 489
Figure 67 - CAP #4 After the Termination of Dump(0) ......................................... 490
Figure 68 - About to Start CIP #5 ................................................................. 490
Figure 69 - About to Terminate Haul(0) in CAP #5 ............................................ 491
Figure 70 - CAP #5 After the Termination of Haul(0) ............................................ 491
Figure 71 - About to Start CIP #6 ................................................................. 492
Figure 72 - About to Terminate Dump(1) in CAP #6 ............................................ 492
Figure 73 - CAP #6 Just After the Termination of Dump(1) ...................................... 493
Figure 74 - About to Start CIP #7 ................................................................. 493
Figure 75 - About to Terminate Return(1) in CAP #7 ........................................... 494
Figure 76 - CAP #7 After the Termination of Return(1) .......................................... 494
Figure 77 - About to Start CIP #8 ................................................................. 494
Figure 78 - Just After the Instantiation of Load(2) in CIP #8 .................................. 495
Figure 79 - About to Terminate Return(0) in CAP #8 ........................................... 495
Figure 80 - CAP #8 Just After the Termination of Return(0) .................................... 496
Figure 81 - About to Start CIP #9 ................................................................. 496
Figure 82 - About to Terminate Load(2) in CAP #9 ............................................ 497
Figure 83 - CAP #9 Just After the Termination of Load(2) ...................................... 497
Figure 84 - About to Start CIP #10 ............................................................... 498
Figure 85 - About to Terminate Haul(2) in CAP #10 ........................................... 498
Figure 86 - CAP #10 Just After the Termination of Haul(2) ..................................... 499
Figure 87 - About to Start CIP #11 ............................................................... 499
Figure 88 - About to Terminate Dump(2) in CAP #11 .................................................. 500
Figure 89 - CAP #11 Just After the Termination of Dump(2)........................................ 500
Figure 90 - About to Start CIP #12 ............................................................................. 500
Figure 91 - About to Terminate Return(2) in CAP #12 .................................................. 501
Figure 92 - CAP #12 Just After the Termination of Return(2)......................................... 501
Figure 93 - About to Start CIP #13 ............................................................................. 502
List of Tables

Table 1 - FEL Snapshot #1........................................................................................................... 33
Table 2 - FEL Snapshot #2........................................................................................................... 36
Table 3 - FEL Snapshot #3........................................................................................................... 37
Table 4 - FEL Snapshot #4........................................................................................................... 40
Table 5 - FEL Snapshot #5........................................................................................................... 41
Table 6 - FEL Snapshot #6........................................................................................................... 42
Table 7 - Action Events .............................................................................................................. 116
Table 8 - Format Specification Fields.......................................................................................... 136
Table 9 - Type Field Characters ................................................................................................. 137
Table 10 - Flag Directives ........................................................................................................... 137
Table 11 - Interpretation of Precision and Defaults...................................................................... 139
Table 12 - Escape Sequences ....................................................................................................... 140
Table 13 - Format String Examples............................................................................................. 140
Table 14 - Number of Random Numbers Retrieved by Statistical Sampling Functions 356
Table 15 - StrbMathError Error Constants ............................................................................... 390
Table 16 - Abbreviations for User-Defined Identifiers ............................................................... 418
Table 17 - Stroboscope Operators and Their Precedence ......................................................... 418
Table 18 - Stroboscope Functions ............................................................................................... 420
Table 19 - Stroboscope Statements.............................................................................................. 424
Table 20 - Type Field Characters ............................................................................................... 432
Table 21 - Flag Directives ........................................................................................................... 433
Table 22 - Interpretation of the Precision Specification............................................................... 434
Table 23 - Escape Sequences ....................................................................................................... 435
Table 24 - CIPROS Attribute Conditional Dependency List.................................................... 476
Table 25 - Simple Haul Data....................................................................................................... 481

Copyright © 1996 Julio C. Martínez
List of Flowcharts

Flowchart 1- Simulation Loop ................................................................. 437
Flowchart 2- Combi Instantiation Phase .................................................. 438
Flowchart 3 - Combi Instantiation Attempt .............................................. 439
Flowchart 4 - Generic Resource Drawing Process .................................... 440
Flowchart 5 - Characterized Draw and Release Process ............................ 441
Flowchart 6 - Activity Instantiation .......................................................... 442
Flowchart 7 - Advance Clock ................................................................. 443
Flowchart 8 - Activity Termination .......................................................... 444
Flowchart 9 - Generic Release Process .................................................... 445
Flowchart 10 - Queue Reactions to Messages ......................................... 446
Flowchart 11 - Characterized Resource Insertion in Queue ....................... 447
Flowchart 12 - Normal Activity Reactions to Messages ............................. 448
Flowchart 13 - Fork Reactions to Messages .............................................. 449
Flowchart 14 - Sending Resources Through Links .................................... 450
Flowchart 15 - Dynafork and Disassembler Reactions to Get In and Out of Context and to Stop Receiving ................................................................. 451
Flowchart 16 - Dynafork Reactions to Receive ......................................... 452
Flowchart 17 - Disassembler Reactions to Receive .................................... 453
Flowchart 18- Assembler Reactions to Get In Context and Receive .......... 454
Flowchart 19 - Assembler Reactions to Stop Receiving and Get Out of Context .... 455
Flowchart 20 - Consolidator Reactions to Messages .................................. 456
List of Appendices

A. Stroboscope Reference ................................................................. 418
B. Flow Charts ................................................................................. 436
C. Terminology ................................................................................ 457
D. Comparison to Other Construction Simulation Tools.................. 460
E. Trace Through a Short Simulation Run .......................................... 481
Chapter 1
Overview

Construction processes range from the very simple to the very complex. Complex processes are difficult to analyze and optimize using standard mathematical methods. Simulation is an alternative method of analysis that offers numerous benefits. Construction simulation systems previously available could not model typical construction processes with the necessary level of detail. General purpose simulation systems, generally based on the process interaction strategy, cannot easily model the multiple resource requirements and dynamic complexity of construction processes.

The purpose of this research was to remedy this situation and address the simulation modeling requirements of construction processes. The end result of this effort is Stroboscope, an acronym for STate and ResOurce Based Simulation of COnstruction ProcEsses. It is a general purpose simulation programming language specifically designed to model construction operations. Stroboscope models consist of a series of programming statements that define a network of interconnected modeling elements, give the elements unique behavior, and control the simulation.

Stroboscope’s ability to dynamically access the state of the simulation and the properties of the resources involved in an operation differentiate it from other construction simulation tools. The state of the simulation refers to such things as the number of trucks waiting to be loaded; the current simulation time; the number of times an activity has occurred; and the last time a particular activity started. Access to the properties of resources means that operations can be sensitive to resource properties — such as size, weight, and cost — on an individual (the size of the specific loader used in
an operation) or an aggregate basis (the sum of the weights of a set of steel shapes waiting to be erected).

Stroboscope modeling elements have attributes — defined through programming statements — that define how they behave throughout a simulation. Attributes represent such things as the duration or priority of an activity, the discipline of a queue, and the amount of resource that flows from one element to another. It is possible to specify most attributes with expressions. Attributes have default values that provide the most common and expected behavior. Expressions are composed of constants; system-maintained variables that access the state of the simulation and the properties of resources; user-defined variables; logical, arithmetic, and conditional operators; and scientific, statistical, and mathematical functions.

Stroboscope simulation models can consider uncertainty in any aspect of the real system being represented (and not just time). A typical example is the uncertainty in the quantities of resources produced or consumed (e.g., the volume of rock resulting from a dynamite blast). Stroboscope models can also select the routing of resources and the sequence of operations dynamically; allocate resources to activities based on complex selection schemes; combine resources and dynamically assign properties to the resulting compound resource; and activate operations subject to complex startup conditions not directly related to resource availability (e.g., do not blast rock until all crews of all trades have left the vicinity, the wiring has been inspected, and there are less than 10 minutes left in the current shift).

The Stroboscope language includes most of the capabilities desirable of general purpose simulation languages such as multiple random number streams; sophisticated stream management; antithetic variates; built-in functions that sample from a wide range of probability distributions; the ability to reset statistical registers or the model itself; a source level flow control language; and a statement pre-processor that allows the parameterized generation of code (i.e., code that writes code). Stroboscope also has some of the characteristics of general purpose programming languages such as built-in logarithmic and trigonometric functions; conventional variables and arrays; and structured flow control with while-wend and if-elseif-else-endif blocks.
In addition, Stroboscope can be seamlessly extended with Add-Ons. Add-Ons are dynamic link libraries written according to the Stroboscope Add-On Interface with conventional compiled languages such as C, C++, Pascal, and Fortran.
Chapter 2

Introduction

In the design of construction operations it is often necessary to make decisions regarding complex processes. These decisions include determining crew sizes, selecting equipment, establishing operating logic, or selecting construction methods. Associated with each decision are a series of outcomes such as construction cost and time. Decisions are made on the basis of their expected outcomes. For example, the equipment fleet to use in an earth moving operation (a decision) may be the one associated with the lowest expected cost.

Several techniques are available to assess the outcomes associated with a particular method of performing a process. Experimentation with the real system, on one extreme, is very realistic but is expensive, slow, lacks generality, and is sometimes impossible to do. Mathematical modeling, on the other extreme, is very precise but requires that important aspects of the process be disregarded, requires a high degree of mathematical ability, and becomes too complex for most real life construction situations. Simulation is the third technique. It is very convenient because, while being realistic, it is also inexpensive, fast, and flexible.

2.1 Discrete Event Simulation

Simulation is a modeling process that imitates a real or imaginary dynamic system. Simulation involves the design of a model of the system and the performance of
experiments on that model. The behavior of the real or imaginary system can be predicted by observing the results of experiments in the model.

Dynamic systems or processes involve the passage of time. At any given point in time, the system is characterized by its state. In discrete-event simulation, it is assumed that the state of a system changes instantaneously at specific times marked by events. Most construction processes can be effectively modeled using discrete-event simulation.

Discrete event simulations can be performed on a computer through the use of general purpose programming languages, or through programming languages or tools designed specifically for simulation.

2.1.1 General-Purpose Programming Languages

Models created in general programming languages can represent almost any real life process. They can be tailored to the very precise requirements of the model in question and can work very fast. Their use in construction has been demonstrated with models for equipment selection (Teicholz 1963), for the assessment of uncertainty in time and cost of underground construction (Moavenzadeh and Markow 1976), for the estimation of project durations (Carr 1979), for the evaluation of resource allocation strategies (Morua-Padilla 1986), and for the modeling of underground geological conditions (Ioannou 1984).

Although some libraries are available to ease development of simulation models using general purpose programming languages, models created with them require that many components be built from scratch. This requires a tremendous amount of effort that is seldom justified. Moreover, these models are geared towards a limited range of processes and are only useful for the particular model or class of models for which they are created.

2.1.2 Simulation-Specific Tools

Many domain-specific and general purpose simulation tools exist. They can be classified as simulators or as simulation languages (Law and Kelton 1991). Simulators are computer packages that allow the simulation of a specific class of systems with little
or no programming. Simulation languages are general in nature, but may have special features for certain types of applications. In general, simulation languages have the ability to model almost any kind of system.

Simulators and simulation languages can adopt one of several approaches, strategies, or decomposition methodologies. Three simulation strategies are commonly recognized: Event Scheduling (ES), Activity Scanning (AS), and Process Interaction (PI). The strategy used by the simulation tool has a strong impact on the way a model is presented to the computer and on how the modeler views the world (Evans 1988). For this reason, the superiority of one strategy over the others has been the source of much discussion, and several comparisons have been made between them (Hills 1973, Zeigler 1976, Hooper and Reilly 1982, Birtwistle et al. 1985, Hooper 1986). All strategies are considered equally general and powerful in terms of being or not being able to represent a particular problem. Particular strategies, however, lend themselves to model certain classes of models more easily.

ES is at the lowest level in terms of the support provided to the modeler and at the highest level in terms of efficiency. An event-based simulation model is driven by the scheduling and execution of subroutines (events) that in turn schedule the execution of other subroutines. Since the ES strategy is very efficient, simulation tools often combine it with the PI or AS strategy.

A PI model is written from the point of view of the entities (transactions) that flow through a system. These entities undergo a process in which they attempt to acquire, take hold of, and release scarce resources. Consider as an example the loading of a steel shape onto a flatbed using a crane. A modeler using a PI approach may try to model this from the point of view of the steel shape. The steel shape (an entity) waits until it can acquire the crane (a resource). After it has acquired the crane it tries to acquire the flatbed (another resource). When the flatbed is acquired, the steel shape uses the crane for a period of time and then releases the crane.

An AS model is written from the point of view of the various activities that can take place. The modeler focuses on identifying activities and the conditions under which the activities can happen. There is no distinction between flowing entities and machines; they are all resources. An AS tool constantly scans the activities to see if they can take
place. When an activity can take place, it is carried out. A model using an AS approach may represent the act of picking up a steel shape with a crane and placing it in a flatbed as an activity called “load”. The conditions necessary for “load” to happen are that a steel shape, a crane, and a flatbed be available and in the correct state. If the conditions are met, then “load” happens and the steel shape, crane, and flatbed are simultaneously acquired, held for some time, and then released.

The PI strategy is very effective in the modeling of systems where the entities that move have many attributes that differentiate them; and where the machines or resources that serve the entities have few attributes, a limited number of states, and do not interact too much. These systems are common in manufacturing and other industries that have been traditional users of simulation. For this reason, the PI strategy alone or combined with ES are the basis of most simulation tools and languages in use in the United States (e.g., GPSS, SLAM, SIMAN, Q-GERT, SIMSCRIPT).

In most construction processes there is heavy interaction between machines, each of which can occupy several locations, have many attributes, and be in several states. This makes it very difficult to use PI tools in construction. Despite these difficulties, languages based on the PI strategy have been used for earth-moving operations (Willenbrock 1972) and repetitive housing unit construction (Ashley 1980).

Simulation languages based on the AS strategy, in contrast, are very strong in modeling systems with highly interdependent components subject to complex activity startup conditions (i.e., many machines with distinct properties and states that must collaborate according to highly dynamic conditions). Since this is the very nature of construction operations, it is no surprise that construction academics and practitioners have used AS tools almost exclusively. The section that follows describes specific AS simulation tools.

2.1.3 Review of Selected Activity-Based Simulation Tools

Civil engineers and construction practitioners make heavy use of graphical sketches and drawings to visualize problems and specify details. Networks are a form of graphical sketch capable of communicating complex concepts that would otherwise
require lengthy explanations. In project level planning, for example, networks are very effectively used in the Critical Path Method (CPM) and the Project Evaluation and Review Technique (PERT).

All the AS simulation tools described here make use of networks that consist of nodes (Activities and Queues) connected together directionally (Arcs or Links) as an aid in describing simulation models. Some describe simulation models entirely through a network.

In addition, most of these tools combine AS with ES into what is known as Three-Phase AS (Tocher and Owen 1960). A three-phase activity scanner distinguishes between conditional activities (C-Activities or Combis) and bound activities (B-Activities or Normals). Conditional activities need to be scanned to see if they can take place, while bound activities are simply scheduled to occur. This division allows for significant improvements in speed since no time is spent scanning bound activities.

### 2.1.3.1 GSP — General Simulation Program

The General Simulation Program (GSP) (Tocher and Owen 1960) introduced the concept of three-phase AS. GSP was regarded as a “machine based” (the original name for AS) “automatic programmer”. The main design objective of GSP was runtime efficiency. As a consequence, a program written in GSP resembles a cipher with many single letter identifiers and keywords (Evans 1988).

Wheel-charts were the first AS simulation networks, which later became known as Activity Cycle Diagrams (ACDs). Wheel-charts were developed by (Tocher 1964) as an aid in identifying conditional activities (C-Activities in GSP) and bound activities (B-Activities in GSP). A wheel-chart consists of a set of boxes linked by arcs that represent a sequence of activities for each machine. Figure 1 shows a wheel-chart involving two machines with intersecting cycles.

When the number of arcs entering a node is one, the activity is bound (the activity can be scheduled to start as soon as the predecessor finishes). When more than one arc enters a node, the activity is conditional (a scan needs to be made to determine if all of its predecessors have finished). Thus, Figure 1 identifies activity “Load” as a conditional activity and the rest as bound activities.
When the sequence of activities in which a machine participates can change, wheel-charts include circles. Figure 2 shows a star-shaped wheel-chart involving a crane. The crane may perform any of the activities (Load, Unload, or Reposition) in any sequence. When an arc enters an activity from a circle, the activity is conditional regardless of the number of arcs that enter it.

Tocher used wheel-charts to describe GSP programs. An actual GSP program would contain instructions such as the following (for the coding of a C-Activity involving cranes similar to one of those shown in Figure 2) from (Tocher 1964):

Figure 1 - Tocher Wheel-chart With Intersecting Machine Cycles

Figure 2 - Tocher Wheel-Chart With Variable Activity Sequence
It is clear that GSP models, while quite simple to understand as networks, become indecipherable when represented in machine readable form.

2.1.3.2 HOCUS - Hand Or Computer Universal Simulator

Hand Or Computer Universal Simulator (HOCUS) (Hills 1971) enhanced and popularized the concept of Activity Cycle Diagrams. A HOCUS ACD consists of Queues (circles) and Activities (boxes) connected by arrows. In contrast to a wheel-chart, the path followed by entities must alternate between Queues and Activities. Figure 3 shows a HOCUS ACD. The connection between the nodes have a pattern that indicates the type of entity that flows through it. Queues and Activities are identified by their numbers, which are placed towards the top on Queues and towards the top-left on Activities.

A HOCUS model is conveyed to the computer through interactive input forms where the details of the nodes and the entities of the model are specified. The information specified inside the Activity usually describes it completely. For example, Activity 10 (Load), requires that a pusher exist in Queue 2 (E2) and that a scraper exist in Queue 3 (E3). When Activity 10 finishes, the pusher is released to the tail of Queue 1 (T1) and the scraper is released to the tail of Queue 4 (T4). The “x10” on top of Activity 10 indicates that up to 10 instances of the Activity can be active at the same time.

When different Activities compete for resources from the same Queues, HOCUS gives priority to the Activity with the lowest number.
The entities in the system can have several integer-valued attributes identified with letters. The specification for Activities allows the manipulation of these attributes through two-letter options. The specification for Activity 10, for example, could be “E2 Pusher T1 L TR -20” and “E3 Scraper T4 K TR +20”, to specify that 20% of the “L” attribute of the pusher be moved to the “accumulator” and then transferred to the “K” attribute of the scraper.

It is also possible to override the implicit “AND” operator among the specifications with “OR”, and to repeat the same entity type more than once to indicate that more than one entity of the given type is required in order to start the Activity.

Although HOCUS is not well known in the United States, it is popular in Europe where it has been the subject of several books (Poole and Szymankiewicz 1977; McDonald, Turner and Szymankiewicz 1988) and has been used for numerous large-scale simulations in several industries.
2.1.3.3 CYCLONE - Cyclic Operations Network

Cyclic Operations Network (CYCLONE) (Halpin & Woodhead 1976) was specifically designed for construction. CYCLONE is purely network based (i.e., the network contains the complete model) and as a consequence is very simple.

A CYCLONE network is an extended version of an ACD. Conditional activities are called Combis and are drawn with a slash on the top left corner of the box. Bound activities are called Normals, they are distinguished from conditional activities and are drawn as plain rectangles. Queues are drawn as circles but with a slash in the bottom right corner so as to resemble the letter Q. All the nodes in a CYCLONE network are identified by a unique integer. Figure 4 shows a CYCLONE network.

![CYCLONE Network Diagram](image)

**Figure 4 - CYCLONE Network**

In CYCLONE, only the conditional Activities (Combis) need to be preceded exclusively by Queues. Combis start when none of the preceding Queues are empty. When several Combis contend for the resources in a Queue, priority is given to the Combi with lowest number. Bound Activities (Normals) can be preceded by any node but a Queue and start immediately after a predecessor finishes.

The entities that flow through a CYCLONE network are indistinguishable and interchangeable. They cannot have properties assigned to them. Special function nodes can multiply and consolidate entities as well as control the simulation run length. The “GEN 4” in Queue number 70, for example, indicates that every entity that enters the
Queue is converted into 4 entities. The “CON 4” in node number 20 indicates that the node will accumulate 4 entities and then release one.

The small number of nodes and simple rules of CYCLONE make it very easy to use as both an analysis and communication tool. Numerous construction processes have been modeled using CYCLONE. They include concrete batch plant operations (Woods and Harris 1980, Lluch and Halpin 1982), and tunneling (Touran and Asai 1987). There are at least four CYCLONE implementations: main-frame CYCLONE (Halpin 1976), Insight (Kalk 1980), UM-CYCLONE (Ioannou 1989), and Micro-CYCLONE (Halpin 1990).

Unfortunately, the pure network characteristic of CYCLONE imposes limits that do not allow us to model processes at the level of detail required to make decisions. Three limitations are recognized to have the most impact: the inability to recognize differences between similar resources (i.e., the properties of resources); the inability to recognize the state of the simulated process; and the inability to make dynamic use of resource properties and the state of the simulation to define model behavior.

2.1.3.4 RESQUE

RESQUE (Chang 1986) was designed as a significant enhancement to CYCLONE where the model is not limited to the information conveyed by the network. In addition to the CYCLONE network, a RESQUE model has an overlay that defines resource distinctions and increases simulation control. The overlay follows a Process Description Language (PDL) specific to RESQUE.

RESQUE sought to overcome the resource characterization capabilities missing in CYCLONE. The solution presented by RESQUE through PDL is a significant improvement over CYCLONE insofar as recognizing distinctions among resources that flow through the same path.

RESQUE identifies resources through a single integer identifier. This identifier represents all the properties of a resource. The PDL statements use the identifier, called attribute in RESQUE, to look up the appropriate probability distributions for activity durations and to look up resource routing rules (RDVLIST). RESQUE activities can be subject to conditional tests (CONDLIST) that can compare the current and total number
of resources at a Queue, or the number of instantiations of an activity, to a constant. In addition, the RESQUE PDL has statements to manage assembly and disassembly of resources into and from sets.

### 2.1.3.5 COOPS

The COOPS construction simulation system (Liu 1991) is an extension to CYCLONE that was completely designed and implemented using an object oriented programming language. The simulation network is a collection of objects such as activities, queues, and links that are drawn interactively on the screen. These perform the simulation by reacting to messages sent from other objects. Moreover, “specific resources” are represented as separate objects to allow the collection of statistical information at the individual level. In addition, COOPS uses calendars to preempt activities during breaks and has the ability to generate and consolidate resources at links.

COOPS' interactive graphical model definition is a great improvement over previous construction simulation systems. Modeling elements are picked, placed and moved directly on the screen, and the need to enter a textual equivalent of the network is removed.

### 2.1.3.6 CIPROS

CIPROS (Odeh 1992) is both a process level and project level planning tool. It contains an expandable knowledge base of construction techniques and methods; and makes ample use of a hierarchical object oriented representation for resources and their properties.

CIPROS extends its resource characterization capabilities beyond RESQUE by allowing multiple real properties for resources as well as more complex resource selection schemes. It integrates process level and project level planning by representing activities through process networks, all of which can use a common resource pool.

CIPROS does not provide access to the state of the simulation.
2.1.3.7 Recent Advances in Construction Simulation

DISCO (Huang and Halpin 1994) is pre-processor and post-processor to Micro-CYCLONE. The pre-processor is an interactive graphical user interface similar to COOPS’. The post-processor animates a simulation by “playing back” various statistics as they occurred during the simulation run.

AP3 (Sawhney and AbouRizk 1994) is a three-tiered planner that divides work into the project, operation, and process level. The process level component is based on CYCLONE. AP3 generates SLAM code.

2.2 Research Objectives

The objective of this research was to create a simulation system capable of modeling construction processes realistically and with ease.

In terms of modeling capability, the objectives were to:

• Consider the diversity of resources and their specific characteristics.

• Allow the state of the simulation to control the sequence of tasks and their relative priorities.

• Model resource selection schemes so that they resemble the way resources are selected for tasks in actual construction operations.

• Model probabilistic material utilization, consumption and production.

• Create a tool that could be easily enhanced and extended.

In order to make the simulation system easy to use, the design of the system had to capitalize on the knowledge likely to have been acquired by its intended users. In these terms, the objectives were to:

• Build upon the traditional Activity Cycle Diagrams.

• Use the Three-Phase Activity Scanning strategy.
• Adapt concepts found in Structured Query Language (SQL) to the selection of resources for operations and the aggregation of their properties.

• Provide access to the state of the simulation and the properties of resources in a manner that resembles the use of objects in modern desktop productivity applications.

2.3 Methodology - A Chronological Outline of Steps and Results

The following steps outline this research:

• Analyzed very carefully the needs of the construction industry in terms of modeling tools. The result of this analysis is formulated in the previous section, “Research Objectives”. Basically, the construction industry needs a tool that is easy to use and at the same time extremely powerful so that it can model construction processes realistically.

• Analyzed existing simulation tools to evaluate their advantages and disadvantages. This analysis showed that existing tools geared towards construction were not capable of modeling construction processes at the level of detail necessary to make decisions. The main problem with the existing tools stemmed from the fact that they were not programmable. It was found, however, that these tools provided a good foundation that could be studied and developed to satisfy the modeling needs of the industry.

• Selected a software analysis and design tool and used it to design a simulation system that met the necessary requirements. The Booch Object-Oriented Analysis and Design Method (Booch 1991, 1993) was selected. The method was very effectively used to design Stroboscope without having to implement it first. The resulting design represents the current Stroboscope architecture.
• Selected a simulation language to implement the design. The C++ programming language (Stroustrup 1991) was selected due to its object-oriented nature, flexibility, speed, and industry support.

• Implemented the design as a kernel that could serve several clients, and implemented the simplest of the clients (a command line driver) to test the kernel. This was completed towards the end of 1993. The kernel design was then tested by modeling the most complex documented simulation models available. The results were satisfactory.

• Designed and implemented an Integrated Development Environment (IDE) and tested for usability. The IDE was completed early in 1994 and was tested by using it to teach simulation to graduate students at the University of Michigan (CEE 631, Winter 1994). The students were able to learn the system in about three weeks and used it to model several complex construction processes.

• Added facilities to the language to simplify the implementation of certain classes of models, to perform multiple automated replications, to implement variance reduction techniques, and to allow for extensibility by others. These were all carried out successfully throughout the remainder of the research, and were again tested by students taking CEE 631 during the Winter term of 1995.

• Designed and implemented a Graphical User Interface hosted on a popular drawing package.

2.4 Dissertation Outline

The bulk of this dissertation describes the Stroboscope simulation language. Chapters 3 through 18 describe the lanugage following a natural progression. They have been written so that they can be read and understood by a wide audience, including people without any prior exposure to discrete event simulation. It is not necessary to study the entire manuscript in order to use Stroboscope for modeling construction processes. Progressive subsets of the language can stand alone and do not require the
more advanced concepts presented later. By the end of chapter 5 this manuscript
describes a subset of the language that can be used to effectively analyze many systems.
By the end of chapter 10 the subset of the language can be used to model almost any
system that does not require the characterization of resources. By the end of chapter 13
the subset includes everything that is necessary to create simulation models of any
complexity. Chapters 14 through 17 contain more advanced material for automatic code
generation, multiple replications, and the implementation of variance reduction
techniques. Chapter 18 provides information on how to extend Stroboscope. The
dissertation concludes with chapter 19, which summarizes the contributions and
achievements of this research.

Appendix A contains summary information about the various Stroboscope
statements, functions, system-maintained variables, and action targets. Appendix B
contains a series of flowcharts that completely describe how the Stroboscope simulation
ingine processes a simulation model. The reader can refer to these flowcharts as different
concepts are introduced in the body of the manuscript. Appendix C includes the
definition of terms used throughout the manuscript. Appendix D compares Stroboscope
to other process and project level construction simulation tools.

Example models are presented throughout the manuscript. The examples range
from very simple small models to complex examples of models that build models.
Chapter 3

Networks

A network is a high level representation of a simulation model. Networks in Stroboscope consist of nodes connected by links through which resources of different types flow. The purpose of this chapter is to provide an introduction to Stroboscope networks.

At the essence of networks are resources and resource types. These are the units of traffic that flow through networks. Resources and resource types will be discussed in section 3.1.

Resources flow from one node to another through links. The basic network elements, namely the nodes and links that compose a network, will be discussed in section 3.2.

Resources, nodes, and links are put together to form a simulation network. Section 3.3 presents a small but complete network and discusses the process modeled by the network.

3.1 Resources and Resource Types

Resources are things required to perform tasks. These can be machinery, space, materials, labor, permits, or anything else needed to perform a particular task. The most important characteristic of a resource is its type.

The type of a resource places the resource within a category of resources that share common traits or characteristics. Truck, Bulldozer, Loader, Cement, Water, and
Mason are examples of resource types. Note that the resource types listed as examples do not represent specific resources, they represent a class of resources. The CAT D8 with serial number 211-RDQ that is sitting in Joe Contractor’s back yard is a resource of type Bulldozer. A construction setting may include several resources of type “Bulldozer”, all of which share common traits and can be used for similar purposes.

Some resources represent unique individual entities. Such is the case of the bulldozer mentioned above, a specific truck, a particular concrete block, etc. These resources are examples of discrete or non-bulk resources. In the English language the word “many” is used to refer to a large collection of non-bulk resources of the same type. For example, “there are many trucks in the highway today,” or “how many concrete blocks did you order?”.

Other resources do not represent individual entities that can be uniquely identified. These resources are bulk. Sand and water are examples of bulk resources. It is impossible to refer to ‘a’ bulk resource. In order to be specific when referring to a bulk resource, it is typically necessary to specify its quantity using suitable units, its location, or the container it is in. Examples of valid references to bulk resources are “add 3.75 cubic meters of sand to 87.5 liters of water,” or “empty the cement in this bag to the pile in front of the mixer.” Statements such as “order a sand,” or “we need a water,” are not meaningful. In the English language the word “much” (in contrast to “many”) is used to refer to a quantity of bulk resources of the same type. For example, “there is much water in the tank,” or “how much sand did you order?”.

Stroboscope strongly enforces the types of resources. The concept of resource type is at the heart of a Stroboscope simulation model. In Stroboscope it is easy to represent discrete as well as bulk resources.

### 3.2 Network Elements

The different kinds of modeling elements are interrelated. In order to fully define one kind, it is necessary to use another. There is a circular definition involved. This makes it necessary to take a quick glimpse at the network fragment shown in Figure 5. The intention of the fragment is to provide context for the explanation of the different
kinds of modeling elements. Figure 5 will not be used to discuss networks per se. Networks will be illustrated later on with other models.

Figure 5 is a simplified model of a bank. Customers arrive to make deposits or withdrawals and then leave. The bank has separate lines (queues) for deposits and withdrawals. The customers are served by a pool of tellers. When the tellers serve clients that wish to make a deposit, they receive money and put the money in the cash register. When the tellers serve clients that wish to make a withdrawal, they remove money from the cash register and give it to the customers.

Three types of resources are involved in the banking operation of Figure 5: customers, tellers, and cash. Customers and tellers are discrete resources whereas cash is a bulk resource. Explanations of some of the nodes and links of this network follow.

![Figure 5 - Simplified Model of a Bank](image)

### 3.2.1 Links

Links connect network nodes and indicate the direction and type of resources that flow through them. The node at the tail of the link is the predecessor and the node at the head (where the arrow is) the successor. Resources flow from the predecessor node to the successor node. The most important
characteristic of a link is its resource type — only resources of the specified type flow through it.

For example, link TL2 in Figure 5 is for tellers. It indicates that tellers flow from Deposit to TellersWait, two of the nodes in the network. Deposit is the predecessor and TellersWait is the successor (Deposit precedes TellersWait). Any teller can flow through TL2. Resources of other types, such as cash and customers, cannot flow through TL2.

Sometimes resources of different types need to flow from the same predecessor node to the same successor node. In these cases it is necessary to connect the two nodes with more than one link, one for each resource type. In Figure 5, link CU13 is for customers. It assumes that the withdrawn cash is carried implicitly by the customer. In order to model the cash flowing from Withdraw to CustomersLeave explicitly, it is necessary to use a separate link for the cash (e.g., CS3). In this case the connection between Withdraw and CustomersLeave would look as shown in Figure 6.

![Figure 6 - Parallel Links Between Withdraw and CustomersLeave](image)

Stroboscope requires that links be named. This is necessary in order to distinguish one link from another. It is convenient to name a link in a manner that indicates the type of resource that flows through it. Although there is no limit to the length of a link name, a useful convention is for the first two letters to be an abbreviation of the resource type. A link for cash could be named CS1, and a link for customers could be named CU20.

Links have many attributes. Some attributes control the flow of resources from the predecessor node to the successor node. Other attributes establish other relationships between these nodes. More details about links will be introduced later.
### 3.2.2 Nodes

During simulation, the resources that are part of a system are held by the various nodes of the associated network model. In particular, resources spend their time in two types of nodes: “Activities” and “Queues”. Activities are nodes in which resources spend time actively (performing a task). Resources involved in Activities are productive, sometimes in collaboration with other resources. The time resources spend in an Activity is the time required to perform the task represented by that Activity. Queues are nodes in which resources spend time passively (they are either stored there, or waiting to be used). The time resources spend in Queues is external to Queues themselves — a resource stays in a Queue until it is removed because some Activity needs the resource to accomplish its task.

#### 3.2.2.1 Queues

Queues hold resources that are idle. Each Queue is associated with a particular resource type. That is, a Queue for tellers can only hold tellers and a Queue for cash can only hold cash. A traditional example of a Queue is the line formed by people waiting at a bank to make a withdrawal; such a Queue appears in Figure 5, it is named `CustWaitToWithdraw`, and holds resources of type `Customer`. Another example, a storage location, is the cash register in a bank; such a Queue appears in Figure 5, it is named `CashRegister`, and holds resources of type `Cash`. A third example, of servers, is the pool of tellers in the bank; such a Queue also appears in Figure 5, it is named `TellersWait`, and holds resources of type `Teller`.

The most important fact about a Queue at any particular point in time is its contents. The manner in which the contents of a Queue is measured depends on the type of resource it holds. If the resource is bulk then its amount is expressed in some unit of measurement (e.g. dollars). If it is non-bulk (i.e., discrete) then its amount is simply a count of the number of resources in the Queue. When a discrete resource enters a Queue, the content of the Queue increases by one. When a discrete resource leaves a Queue, the content of the Queue decreases by one. The content of a Queue that holds discrete resources is never fractional. When a bulk resource enters a Queue, the content of the
Queue increases by the amount of resource that enters. When a bulk resource leaves a Queue, the content of the Queue decreases by the amount of resource that leaves. The content of a Queue that holds bulk resources can be fractional.

In Queues that hold discrete, uniquely identifiable resources, only the resources that enter the Queue can leave it. For example, if John, Paul, George, and Ringo are the only customers that enter CustWaitToWithdraw (and have not left the Queue yet), then only John, Paul, George, and Ringo can leave CustWaitToWithdraw. Although not of interest now, Queues that hold these types of resources have attributes that control the ordering of the individual resources within the Queue.

In contrast, Queues that hold bulk resources make no distinction between the resources that enter the Queue and those that leave the Queue. This is because the bulk resources stored in a Queue are indistinguishable and interchangeable.

### 3.2.2.2 Activities

Activities are nodes that represent work or tasks to be performed using the necessary resources. In Stroboscope there are three types of Activities. The Normal Activity and the Combi Activity will be discussed below and a third type will be discussed in chapter 10. Combi and Normal Activities differ in the way in which the tasks that they represent may start. They also differ in the manner in which they acquire the resources they need.

An Activity represents a task that can take place zero, one, or several times during simulation. The repetitive tasks represented by an Activity can take place in series, in an overlapped fashion, or even in parallel. In the case of the Withdraw Activity (Figure 5), for example, several tellers can be serving several customers simultaneously. If two customers arrive at the bank at opening time, and two or more tellers are in their spots waiting for customers, then two occurrences of Activity Withdraw will start at the same time. Every occurrence of an Activity is called an instance of the Activity. Thus, during simulation several instances of Withdraw can happen concurrently.

Each instance of an Activity has its own duration that represents how long it takes to do the associated work. Activity instances also hold those specific resources that were acquired in order to start it.
Once created, an Activity instance exists for an amount of time equal to its duration. After this amount of time elapses during simulation, the instance of the Activity is terminated (destroyed). When this happens, the resources that were packaged inside the Activity instance are released to successor nodes through the links that leave the Activity.

### 3.2.2.2.1 Combi Activities

Combi Activities represent tasks that start when certain conditions are met. For the sake of brevity, the term “Combi” will from here onwards refer to “Combi Activity”. At appropriate moments during simulation, Combis are scanned (examined one by one) to determine if the necessary conditions exist for them to start. In the majority of cases, these startup conditions relate to resource availability. For example, `Withdraw` is a Combi that requires three types of resources: a waiting teller, a customer who wishes to withdraw cash, and cash in the cash register. When cash, customer, and teller are all available, `Withdraw` can start (an instance of it can be created).

Combis can acquire only resources that are inactive; they cannot interrupt (preempt) other tasks to obtain resources from them. Since inactive resources can only reside in Queues, Combis must draw resources from Queues. For this reason, all the predecessors to a Combi must be Queues (i.e., those Queues that hold the resources needed to start the Combi).

The default condition for a Combi to start is that none of its directly preceding Queues be empty. Thus, in order to determine whether a Combi can start it is necessary to examine the contents of its directly preceding Queues. If the contents of each of these Queues is non-zero (not empty), then the Combi can start and an instance of it may be created. It is possible to change the default startup conditions through attributes of the Combi and/or the links that come into the Combi. A detailed explanation of how to achieve this, however, is not necessary at this point and will be postponed until later.

When a Combi starts it removes resources from the Queues that precede it. By default, a Combi draws (removes) one unit of resource through each of the links that enter it. If this default behavior is not the one desired, it is possible to specify the number
or quantity of resources that a Combi draws through a link by using link attributes. Furthermore, it is possible to specify the exact subset of resources that a Combi will draw from those stored in a preceding Queue (this, of course, applies only for Queues that hold a set of discrete, individually identifiable resources). For the time being, however, these attributes will not be used to control the resources drawn from Queues; the current discussion continues on the presumption that only one unit of resource is removed from each preceding Queue.

3.2.2.2 Normal Activities

Normal Activities represent tasks that start immediately after other tasks end. For the sake of brevity, the term “Normal” will from here onwards refer to “Normal Activity”. A Normal acquires the resources required to perform its task from the task that has just finished. CustomersLeave in Figure 5 is an example of a Normal. This Activity starts immediately after an instance of Withdraw ends. In order for this to happen, there must be a link from Withdraw to CustomersLeave that allows resources of type Customer to flow through, and that transmits the signal for CustomersLeave to start when Withdraw ends. This is link CU13. CustomersLeave receives the customer from the terminating instance of Withdraw. Notice that the terminating instance of Withdraw releases the teller through TL4 to TellersWait. The cash leaves with the customer and is not modeled explicitly once it is in the customer’s possession.

Among all nodes in a network, only Activity instances represent tasks that end and release resources. For this reason, only other Activities can be predecessors to a Normal. More than one Activity can precede a Normal. For example, CustomersLeave (Figure 5) can happen not only after an instance of Withdraw ends, but also after an instance of Deposit ends. In this case, a separate instance of CustomersLeave gets created every time an instance of either one of its predecessors finishes.
3.3 Networks

Nodes connected by links form a network that provides a high-level description of the operation being modeled. The network in Figure 7 shows a typical earth-moving operation. The purpose of this operation is to move soil from one place to another using loaders and haulers.

This model uses resources of type Loader, Hauler, and Soil. Soil is a bulk resource type. Loader and Hauler are discrete resource types. The network contains one Combi named Load; three Normals named Haul, Dump, and Return; and four Queues named SoilToMove, LoadersWait, HaulersWait, and MovedSoil. SoilToMove and MovedSoil hold resources of type Soil; HaulersWait holds resources of type Hauler, and LoadersWait holds resources of type Loader.

At the beginning of a simulation the resources initially in the system reside in Queues. In the operation shown in Figure 7, HaulersWait contains some haulers, LoadersWait contains some loaders, and SoilToMove contains some soil. How many resource there are in each is not relevant at present.

Links SL1, SL2, SL3, and SL4 indicate that soil is initially at rest in SoilToMove. It is then loaded to a hauler, hauled, and finally dumped to become part of MovedSoil.

Figure 7 - Classic Earth-Moving Operation
Links \textit{LD1} and \textit{LD2} indicate that loaders are withdrawn from \textit{LoadersWait} to load a hauler, and after loading a hauler with soil, they move back to wait to load again. Links \textit{HL1}, \textit{HL2}, \textit{HL3}, \textit{HL4}, and \textit{HL5} indicate that haulers are initially waiting to be loaded in \textit{HaulersWait}, get loaded, haul, dump, return, and wait to get loaded again.

Note that loaders and haulers follow cyclic paths. In contrast, the soil originates in one place and ends in another.

All three of these resource types are involved in \textit{Load}. Haulers and soil are involved in \textit{Haul} and \textit{Dump}. Only haulers are involved in \textit{Return}.

Note that the links are named in a manner that indicates the type of the resource that flows through them. Links for soil begin with \textit{SL}, links for haulers begin with \textit{HL}, and links for loaders begin with \textit{LD}. The naming convention adopted in this network also indicates the relative order in which the different types of resources traverse the links. Soil will first be drawn through link \textit{SL1}, and then be successively released through links \textit{SL2}, \textit{SL3}, and \textit{SL4}. The 1, 2, 3, and 4 appended to \textit{SL} in the names of the links indicate a certain sequence. Naming for the hauler and loader links follow a similar pattern. The links are named in this example purely by convention. Link \textit{SL1} could have been named \textit{SoilsDrawnFromPileToTruck}. Although such a long link name is legal, it is not practical for two reasons. First, it is simply too long and cumbersome to write above the link itself. Second, it does not convey any information about the relative order of the link in the path followed by the resource (soil in this case).

The network shown in Figure 7 contains only one Combi — \textit{Load}. The conditions necessary for \textit{Load} to start depend on the contents of its preceding Queues. In order for any Combi to start, the preceding Queues must contain enough resources to support the needs of the Combi. By default, any non-zero amount of a bulk resource or at least one discrete resource is considered enough. Thus, \textit{Load} will start whenever there is soil in \textit{SoilToMove}, at least one hauler in \textit{HaulersWait}, and at least one loader in the \textit{LoadersWait}.

Several loaders can be loading an equal number of haulers with soil simultaneously. Each occurrence would correspond to a different \textit{Load} instance. At the beginning of the day (the start of the simulation), several \textit{Load} instances could start at exactly the same time.
By default, a starting Combi removes one unit of resource from each of its preceding Queues. Thus, every time Load starts, it will remove one unit of soil (provided one full unit is available, a unit could be a hauler-load) from SoilToMove, one hauler from HaulersWait, and one loader from LoadersWait. The loader, hauler and soil will be packaged into an instance of Load. This instance starts at the current simulation time and will be terminated sometime in the future.

The startup of an Activity and the creation of an instance of the Activity take no simulation time. The computer, however, needs to perform several steps sequentially in order to simulate this. During these steps, resources are removed from Queues and packaged into an instance of the Activity. For this reason, the contents of Queues before and after a Combi starts are not the same. Note that before and after, as used in the previous statement, refer to sequential moments that occur at the same simulation time.

It is possible that after Stroboscope starts and instantiates Load (i.e., removes resources from the preceding Queues and creates an instance of the Combi) its startup conditions can still be met. In this case, Load will start again, removing more resources from each of the preceding Queues and creating another instance of itself. Load will continue to start until one of the preceding Queues is empty. This will eventually happen when there are no loaders in LoadersWait, there are no haulers in HaulersWait, or there is no soil in SoilToMove. Note that all the instances of Load created in this scenario start at the same time and run in parallel.

For example, if there are 3 loaders, 5 haulers, and 1000 hauler-loads of soil at the start of the simulation, 3 instances of Load would be created at time 0. At that moment it would not be possible to create more instances because LoadersWait would be empty. The 2 haulers that remain in HaulersWait, and the 997 hauler-loads of soil that remain in SoilToMove, would need to wait until some loaders become available (i.e., until some loaders enter LoadersWait).

When the time comes to terminate an instance of Load, this instance will release the loader, hauler and soil packaged within the instance through links LD2, HL2 and SL2, respectively. The loader will be released to LoadersWait. The hauler and soil will go to Haul.
The entry of resources to Queues creates the possibility that any of the Combis that follow the Queue may be able to start. For example, assume that the last attempt to start Load was not successful because LoadersWait was empty. It is then possible that the entry of a loader to LoadersWait will enable Load to start the next time it gets scanned (this will happen sequentially later, but at the same simulation time).

Because Load is a predecessor to Haul, the termination of an instance of Load causes Haul to start. The hauler and soil received by Haul will be packaged into an instance of Haul that will in turn be terminated sometime in the future. Note that Haul starts because an instance of Load terminates. No conditions have to be checked to make it start (no Activity scanning is necessary).

The termination of an instance of Haul causes Dump to start. Dump will receive the hauler and soil from Haul, and package them into an instance of Dump. This instance of Dump will in turn be terminated further into the future. The termination of an instance of Dump will release soil to MovedSoil and a hauler to Return. Return will then start and create an instance of itself. When this instance of Return terminates, the hauler is released to HaulersWait. The entry of a hauler to HaulersWait can (if LoadersWait and SoilToMove are not empty) set the conditions necessary for Load to start the next time it is scanned.

Note that, at any point in time, several instances of the different Activities in the network could be taking place. A hauler can be loaded while another hauler is dumping; several haulers could be loaded, or be dumping concurrently; etc.

From the above example shows that resources spend time in Activities and in Queues. The amount of time that resources spend in an Activity depends on the duration of the particular instance of the specific Activity in which they are involved. This time is determined when the instance is created.

The amount of time resources have to wait in Queues depends on factors external to the Queues themselves. This amount of time is not known at the time a resource enters a Queue. It is only known when a resource actually leaves the Queue. Resources stay in Queues until a successor Combi removes them. This happens when the conditions necessary to start the Combi are satisfied.
Chapter 4

Basic Simulation Processing

This chapter is an introductory discussion of the Stroboscope simulation engine. Its purpose is to develop a basic understanding of a Stroboscope simulation. This discussion is at a rather high level and does not go deeply into details. The concepts and terms introduced in this section, however, are fundamental. A thorough understanding of these concepts will be invaluable to comprehend the chapters that follow.

4.1 Simulation Clock

The simulation clock keeps the current time in the simulation. It is a double-precision floating-point number that has no intrinsic relationship to any specific time unit. It is up to the model developer to establish the meaning of the unit of time used by the simulation clock. One unit of time can represent 1 second, 1 minute, 20 minutes, 1 day, etc. All times in the model need to be consistent with whatever unit is chosen for the simulation clock. It is also necessary to interpret simulation output in terms of that unit. If the base unit of time is the minute, for example, then all Activity durations should be expressed in minutes, and all time-related results will also be in minutes.

The simulation clock has a value of zero at the beginning of the simulation. As the simulated process advances in time, the simulation clock increases in value (the clock is advanced). These increments are discrete steps that are not always equal in size. Sometimes the increment is very small, at other times the increment can be quite large.
Later sections explain exactly when and by how much Stroboscope advances the simulation clock.

The computer performs a very large number of steps without advancing the clock. In terms of simulated time, these computations (steps) are simultaneous and instantaneous. The computer, however, performs these computations sequentially. The order in which the computer performs these calculations is very important because one step usually depends on the result of the previous step. The terms “before” and “after” will be used throughout this manuscript to indicate the relative sequence of these steps even though they may happen at the same real time.

### 4.2 Activity Instance Life-span

During simulation, instances of Activities are created and terminated. The start time of an Activity instance is the value of the simulation clock at which the instance of the Activity is created. The end time of an Activity instance is the value of the simulation clock at which the Activity instance will be terminated. The duration of an Activity instance is the simulated time between its instantiation and its termination. A particular instance of an Activity exists only during the period of simulated time that begins with its instantiation and ends with its termination.

For example, if instance # 57 of the Load Activity \((\text{Load}(57))\) is created at 10:30 (start-time), and lasts 10 minutes (duration), it will be terminated at 10:40 (end-time). \(\text{Load}(57)\) exists only between 10:30 and 10:40.

### 4.3 The Future Events List

The Future Events List (FEL) is a list that keeps all the Activity instances that exist (i.e., have not been terminated) at any point during a simulation. This list is sorted according to the “end-time” of the Activity instances it contains. As a result, browsing this list tells us the chronological sequence of future Activity instance terminations. This list represents all the tasks that are currently being performed in a model.
The content of the FEL is dynamic. As instances of Activities are created, they are added to the FEL. As instances of Activities are terminated, they are removed from the FEL.

Each entry in the FEL actually holds an event that will happen at the current time or in the future. This event is the termination of the Activity instance listed in the entry.

Events appear in the FEL in chronological order. In other words, Activity instances are sorted in ascending order based on their end-time. Thus, the FEL is a chronological list of all Activity termination events known to happen in the future. In the case where two or more Activity instances happen to end at exactly the same simulation time, the instances appear in the FEL in the order in which they were created.

Table 1 shows the FEL during a simulation of the model shown in Figure 7 on page 27. This FEL exists as shown in the table only for a short while. As the simulation proceeds, some instances will be terminated and thus removed from the list. Other instances will be created and thus inserted in their appropriate place in the list.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Started At</th>
<th>Will End At</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load(5)</td>
<td>11.32</td>
<td>12.37</td>
</tr>
<tr>
<td>Return(1)</td>
<td>8.33</td>
<td>13.44</td>
</tr>
<tr>
<td>Haul(4)</td>
<td>11.32</td>
<td>16.21</td>
</tr>
<tr>
<td>Haul(3)</td>
<td>10.83</td>
<td>16.97</td>
</tr>
<tr>
<td>Return(2)</td>
<td>9.66</td>
<td>17.12</td>
</tr>
</tbody>
</table>

*Table 1 - FEL Snapshot #1*

The first event shown in the table will occur at simulation time 12.37 (time is now represented by a decimal number, not in hours and minutes), the termination of Load instance number 5. The table also shows that two instances of Haul are ‘active’: instance number 4 will terminate at time 16.21, and instance number 3 at time 16.97. Notice that Haul(4) appears before Haul(3), even though Haul(3) started first. This is because the event that marks the termination of Haul(4) will occur before the event that marks the termination of Haul(3). In physical terms, this implies that one hauler will overtake (pass) the other even though the latter started hauling first.
4.4 Current Events

The start and end times of the Activity instances in Table 1 indicate that the current value of the simulation clock is between 11.32 and 12.37 (inclusive). The clock has a value of 11.32 if Load(5) or Haul(4) have just been created; a value between 11.32 and 12.37 if some instance of some Activity has just been terminated; and a value of 12.37 if Load(5) is about to be terminated.

If the simulation clock has a value of 12.37, then the FEL contains an event that will not occur in the future: The event for the termination of Load(5) is bound to occur at the current simulated time.

Events bound to occur at the current simulation time are called “Current Events”. Current events are ready to be removed from the FEL. Whenever the FEL contains current events, Stroboscope focuses all its attention into removing them from the FEL. This is done by terminating each of the Activity instances represented by the events.

Of all things performed by Stroboscope while processing a simulation model, processing current events is the most important and must be accomplished immediately whenever it is necessary. This is done by terminating each instance of an Activity that appears in the FEL whose end-time matches the current value of the simulation clock. If several Activity instances are due to terminate at the current simulation time, then Stroboscope will terminate them one by one in the order in which they appear in the FEL. Although the terminations happen at the same simulated time, Stroboscope must process them in a sequential fashion.

4.5 Activity Instance Termination

The termination of an Activity instance releases resources through the links that leave the terminating Activity. These resources eventually reach the terminating Activity’s successors. Some of these successors are Queues and others are Normals. The content of a Queue increases when it receives resources. A successor Normal receives all the resources it will get from the terminating instance. When this transfer of resources in complete, Stroboscope packages the resources received into an instance of the Normal.
Just before Stroboscope terminates an Activity instance, it removes it from the FEL. **During** termination, Stroboscope creates instances of the Normals that are successors to the terminating Activity. Stroboscope inserts the instances it created into the FEL. Thus, the termination of an Activity instance does not necessarily reduce the number of events listed in the FEL. If the terminating Activity has no Normals as successors, the number of events in the FEL decreases by one. If, in contrast, the terminating Activity has 4 Normals as successors, the FEL actually grows by 3 (Stroboscope removes 1 event and adds 4).

If during the termination of an Activity instance Stroboscope instantiates zero-duration Normals, then the FEL still contains current events. Stroboscope must therefore continue processing the FEL by terminating the zero-duration instances that were just created.

As examples, let us examine two scenarios for the termination of **Load(5)** (Table 1) in the network of Figure 7. In the first scenario, the instance of **Haul(5)** created during the termination of **Load(5)** is assumed to have a non-zero duration. In the second scenario, it is assumed to have a duration exactly equal to zero. Both scenarios assume that **Load(5)** terminates at time 12.37.

**Scenario 1, Haul(5) has a non-zero duration:**

1) Stroboscope removes **Load(5)** from the FEL.

2) Stroboscope releases the loader packaged with **Load(5)** to **LoadersWait**. **LoadersWait** now contains one more loader.

3) Stroboscope releases the soil and hauler packaged with **Load(5)** to **Haul**.

4) **Haul** starts and gets ready to create an instance of itself.

5) **Haul** determines the duration of the instance it will create. Assume this duration is 4.52 (non zero).

6) **Haul** packages the hauler and soil it received into instance # 5 of **Haul**. **Haul** also packages the start time (12.37), duration (4.52), and end time (16.89) into the instance.

5) Stroboscope inserts **Haul(5)** in the FEL between **Haul(4)** and **Haul(3)**.
6) The termination of \textit{Load}(5) is complete.

When the termination of \textit{Load}(5) is complete, the FEL looks as shown in Table 2. At this point the current simulation time is still 12.37 and the FEL contains no current events.

\begin{table}
\centering
\begin{tabular}{|l|c|c|}
\hline
Instance & Started At & Will End At \\
\hline
\textit{Return}(1) & 8.33 & 13.44 \\
\textit{Haul}(4) & 11.32 & 16.21 \\
\textbf{\textit{Haul}(5)} & \textbf{12.17} & \textbf{16.89} \\
\textit{Haul}(3) & 10.83 & 16.97 \\
\textit{Return}(2) & 9.66 & 17.12 \\
\hline
\end{tabular}
\caption{FEL Snapshot #2}
\end{table}

To further understand how Stroboscope processes the FEL, consider the second scenario in which the duration of \textit{Haul}(5) is zero instead of 4.52.

\textbf{Scenario 2, \textit{Haul}(5) has a zero duration:}

1) Stroboscope removes \textit{Load}(5) from the FEL.

2) Stroboscope releases the loader packaged with \textit{Load}(5) to \textit{LoadersWait}. \textit{LoadersWait} now contains one more loader.

3) Stroboscope releases the soil and hauler packaged with \textit{Load}(5) to \textit{Haul}.

4) \textit{Haul} starts and gets ready to create an instance of itself.

5) \textit{Haul} determines the duration of the instance it will create. Assume this duration is 0.

6) \textit{Haul} packages the hauler and soil it received into instance # 5 of \textit{Haul}. \textit{Haul} also packages the start time (12.37), duration (0), and end time (12.37) into the instance.

5) Stroboscope places \textit{Haul}(5) at the beginning of the FEL, just before \textit{Return}(1).

6) The termination of \textit{Load}(5) is complete.
The resulting FEL is shown in Table 3.

At this point the current simulation time is still 12.37. The FEL contains an event for the termination of Haul(5) that is bound to occur at the current simulation time. Since the FEL still contains current events, it is necessary to process it further. Haul(5) needs to be terminated immediately.

### 4.6 Combi Instantiation Phase

Once the FEL contains no current events, Stroboscope turns its attention to the possible startup of Combis. In order to determine if any Combi can start, Stroboscope needs to examine each Combi, the Queues that precede the Combi, and possibly other aspects of the simulation. The procedure by which Stroboscope scans (examines for possible startup) and instantiates the Combis in a network is called the Combi Instantiation Phase (CIP).

At any point during a simulation, the terminations of all currently existing Activity instances are listed in the FEL. Therefore, from the point of view of the system, these terminations are “known” events. In contrast, Stroboscope has no explicit information that indicates whether any particular Combi is able to start at the current simulation time. It is only after Stroboscope scans a Combi that it knows whether the Combi can indeed start. Therefore, the event that marks the instantiation of a Combi is “unknown” to the system prior to the CIP.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Started At</th>
<th>Will End At</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haul(5)</td>
<td>12.37</td>
<td>12.37</td>
</tr>
<tr>
<td>Return(1)</td>
<td>8.33</td>
<td>13.44</td>
</tr>
<tr>
<td>Haul(4)</td>
<td>11.32</td>
<td>16.21</td>
</tr>
<tr>
<td>Haul(3)</td>
<td>10.83</td>
<td>16.97</td>
</tr>
<tr>
<td>Return(2)</td>
<td>9.66</td>
<td>17.12</td>
</tr>
</tbody>
</table>

*Table 3 - FEL Snapshot #3*
Recall from the scenarios described in section 4.5 that when the duration of \textit{Haul}(5) is zero, Stroboscope continues processing the FEL. It has to process “known” events and does not enter the CIP.

In contrast, when the duration of \textit{Haul}(5) is non-zero, the FEL contains no current events and looks as shown in Table 2. There are no more “known” events to process at the current simulation time and thus Stroboscope enters the CIP. At this point it may be possible to create a new \textit{Load} instance. \textit{Load} needs loaders, and \textit{LoadersWait} has just received a loader. If there is still soil in \textit{SoilToMove} and at least one hauler in \textit{HaulersWait}, then \textit{Load} can create its instance # 6. During the CIP, Stroboscope will scan and start \textit{Load} as many times as possible. The CIP will stop when it is no longer possible to satisfy the starting conditions for \textit{Load} or when the instantiation of a zero-duration Combi inserts a current event in the FEL.

Stroboscope simulation networks can contain many Combis. Stroboscope scans these Combis one by one in a specific order. The scan order has important implications in the processing of the simulation model.

An example is when the same Queue is predecessor to more than one Combi. This means that several different tasks may require the resources stored in the Queue. If the resources stored in the Queue are sufficient to support the task represented by either of the succeeding Combis, but not enough to support all of them, only one of the Combis will be able to start. The Combi that will start is the one that Stroboscope scans first. Scanning order is also important for other reasons that will be discussed later.

The current CIP discussion ignores the issues related to the order in which Stroboscope scans Combis. These issues will be considered in depth later.

During the CIP, Stroboscope scans and starts the Combis in a simulation network. Stroboscope continues to do so until it is not possible to start any Combi, or until the instantiation of a zero-duration Combi inserts a current event in the FEL. This implies that, during the CIP, Stroboscope can create several instances of the same or of several Combis. Thus, upon completion of the CIP, the simulation model contains neither known events (Activity instance terminations) nor unknown events (Combi instantiations) that can be processed at the current simulation time. When this happens, another phase in the simulation must start (this phase is described in the next section).
During the CIP, Stroboscope inserts instances of Combis in the FEL as the instances are created. When one of these instances has a zero duration, the FEL suddenly contains current events. Stroboscope abandones the CIP and proceeds to terminate the Combi instance that was just created (the current event).

To further understand the CIP this discussion continues with the example described in scenario 1 in section 4.5. Assume that Stroboscope has just terminated Load(5), and that Haul(5) (created during the termination of Load(5)) has a non-zero duration of 4.52. The current simulation time is thus 12.37, and the FEL looks as shown in Table 2 on page 36.

Let us also assume that HaulersWait contains 3 haulers, that LoadersWait contains 2 loaders, and that SoilToMove contains 100 hauler-loads of soil. (Note that it is almost impossible to encounter a situation like this in the processing of a non-deterministic simulation model. Such a situation occurs only if two instances of Load terminate at exactly the same time. This situation, however, is useful for illustration purposes.)

Let us investigate two different scenarios. In the first scenario, the next two instances of Load (Load(6) and Load(7)) have durations of 1.23 and 0.78, respectively. In the second scenario, they have durations of 0 and 1.12, respectively.

**Scenario 1, duration of Load(6)=1.23, duration of Load(7)=0.78:**

1) The CIP begins.

2) Stroboscope scans Load. Since none of the Queues that precede Load are empty, the scan indicates that Load can start.

3) Load removes one loader from LoadersWait, one Hauler from HaulersWait, and one hauler-load of soil from SoilToMove. HaulersWait now contains 2 haulers, LoadersWait contains 1 loader, and SoilToMove contains 99 hauler-loads of soil.

4) Load starts and gets ready to create an instance of itself.

5) Load determines the duration of the instance it will create. This duration is 1.23.

6) Load packages the hauler, loader, and soil into Load(6). Load also packages the start time (12.37), duration (1.23), and end time (13.60) into Load(6).
7) Stroboscope inserts Load(6) in the FEL between Return(1) and Haul(4). The FEL now looks as shown in Table 4.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Started At</th>
<th>Will End At</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return(1)</td>
<td>8.33</td>
<td>13.44</td>
</tr>
<tr>
<td>Load(6)</td>
<td>12.37</td>
<td>13.60</td>
</tr>
<tr>
<td>Haul(4)</td>
<td>11.32</td>
<td>16.21</td>
</tr>
<tr>
<td>Haul(5)</td>
<td>12.17</td>
<td>16.89</td>
</tr>
<tr>
<td>Haul(3)</td>
<td>10.83</td>
<td>16.97</td>
</tr>
<tr>
<td>Return(2)</td>
<td>9.66</td>
<td>17.12</td>
</tr>
</tbody>
</table>

*Table 4 - FEL Snapshot #4*

8) Stroboscope scans Load again. Since none of the Queues that precede Load are empty, the scan indicates that Load can start again.

9) Load removes one loader from LoadersWait, one Hauler from HaulersWait, and one hauler-load of soil from SoilToMove. HaulersWait now contains 1 hauler, LoadersWait is empty, and SoilToMove contains 98 hauler-loads of soil.

10) Load starts and gets ready to create an instance of itself.

11) Load determines the duration of the instance it will create. This duration is 0.78.

12) Load packages the hauler, loader, and soil into Load(7). Load also packages the start time (12.37), duration (0.78), and end time (13.15) into Load(7).

13) Stroboscope inserts Load(7) in the FEL before Return(1), at the head of the list. The FEL now looks as shown in Table 5.

14) Stroboscope scans Load again. Since LoadersWait is empty, the scan indicates that Load cannot start again.

15) The CIP is over — no more Combis can start.
At this point the current simulation time is still 12.37. The FEL contains only events that will occur in the future, and it is not possible to start any Combi in the network. There are no more known or unknown events to process. As will be discussed later, this situation calls for an advance of the simulation clock.

Before advancing the simulation clock, however, let us follow through the CIP in the second scenario.

**Scenario 2, duration of Load(6)=0, duration of Load(7)=1.12:**

1) The CIP begins.

2) Stroboscope scans load. Since none of the Queues that precede load are empty, the scan indicates that load can start.

3) load removes one loader from LoadersWait, one Hauler from HaulersWait, and one hauler-load of soil from SoilToMove. HaulersWait now contains 2 haulers, LoadersWait contains 1 loader, and SoilToMove contains 99 hauler-loads of soil.

4) Load starts and gets ready to create an instance of itself.

5) Load determines the duration of the instance it will create. This duration is 0.

6) Load packages the hauler, loader, and soil into Load(6). Load also packages the start time (12.37), duration (0), and end time (12.37) into Load(6).

7) Stroboscope inserts Load(6) in the FEL before Return(1), at the head of the list.

The FEL now looks as shown Table 6. Notice that now the FEL contains an event

<table>
<thead>
<tr>
<th>Instance</th>
<th>Started At</th>
<th>Will End At</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load(7)</td>
<td>12.37</td>
<td>13.15</td>
</tr>
<tr>
<td>Return(1)</td>
<td>8.33</td>
<td>13.44</td>
</tr>
<tr>
<td>Load(6)</td>
<td>12.37</td>
<td>13.60</td>
</tr>
<tr>
<td>Haul(4)</td>
<td>11.32</td>
<td>16.21</td>
</tr>
<tr>
<td>Haul(5)</td>
<td>12.17</td>
<td>16.89</td>
</tr>
<tr>
<td>Haul(3)</td>
<td>10.83</td>
<td>16.97</td>
</tr>
<tr>
<td>Return(2)</td>
<td>9.66</td>
<td>17.12</td>
</tr>
</tbody>
</table>

*Table 5 - FEL Snapshot #5*
for the termination of \textit{Load}(6). This event is bound to occur at the current

\begin{center}
\begin{tabular}{|l|c|c|}
\hline
Instance & Started At & Will End At \\
\hline
\textit{Load}(6) & 12.37 & 12.37 \\
Return(1) & 8.33 & 13.44 \\
\textit{Haul}(4) & 11.32 & 16.21 \\
\textit{Haul}(5) & 12.17 & 16.89 \\
\textit{Haul}(3) & 10.83 & 16.97 \\
Return(2) & 9.66 & 17.12 \\
\hline
\end{tabular}
\end{center}

\textit{Table 6 - FEL Snapshot #6}

simulation time, 12.37.

8) Stroboscope abandons the CIP. This happens despite the fact that it is possible to create another \textit{Load} instance (none of \textit{Load}'s predecessor Queues are empty). Stroboscope now focuses on processing current events, to be achieved by terminating \textit{Load}(6) immediately.

Note from the discussion of the CIP, that Stroboscope creates instances of Combis during the CIP. This is in contrast to instances of Normals, which Stroboscope creates during the termination of other Activity instances.

The points to remember about the Combi Instantiation Phase are:

- The CIP starts only after the FEL contains no current events

- The CIP ends in one of two ways,

  - When no more Combis can start, or

  - When the last Combi instance created has a zero duration (which inserts a current event in the FEL)
4.7 Clock Advance Phase

When the CIP ends normally it is not possible to start any Combi in a simulation network. The normal completion of the CIP implies that the FEL contains only events that will occur in the future. Thus, there are no more known (Activity instance terminations) or unknown (Combi instantiations) events to process at the current simulation time.

Processing the FEL and the CIP take no simulated time. Once the CIP has ended normally, nothing else can happen at the current simulation time. Although tasks are being performed (those listed in the FEL), no discrete event of interest will happen until some instance of some Activity terminates. The nearest time into the future at which such an event will take place can be determined by examining the FEL. The time corresponds to the earliest of the future events listed there — the event at the head of the list.

At this point in the processing of a simulation model, Stroboscope advances the simulation clock to the nearest time into the future at which something interesting will happen (the end-time of the Activity instance at the head of the FEL). If the FEL is empty, then there are no pending future events and it is not possible to continue processing the simulation model. The simulation stops due to a “lack of resources” (if “enough resources” were indeed available, then some Combi would have been able to start during the CIP and the FEL would not have been empty).

Advancing the clock to the earliest of the future events changes the status of some of the events in the FEL from “future” events to “current” events. Simulation processing now focuses on the current events. The advance of the simulation clock is a very simple action that has a substantial secondary effect, the immediate need to process the FEL. Advancing the simulation clock and then going on to process the FEL is called the Clock Advance Phase (CAP).

For example, consider the case described in scenario 1 in section 4.6 (the CIP normal completion scenario). After the CIP is complete, the current simulation time is 12.37 and the FEL looks as shown in Table 5 on page 41. Stroboscope enters the CAP because no Combi can start (the CIP is complete) and the FEL contains only events bound to occur in the future. In the CAP, the simulation clock advances to time 13.15
(the termination time of \textit{Load(7)}, the earliest of the events to occur in the future). By virtue of setting the new value of the simulation clock, Stroboscope must process the termination of \textit{Load(7)} immediately.

\section{The Simulation Loop}

Stroboscope enters the CAP whenever the CIP ends normally.

When the CIP ends due to the instantiation of a zero-duration Combi, simulation processing continues with the processing of the FEL without advancing the simulation clock. For all practical purposes, this is a special case of the CAP in which the earliest of the “future” events happens at the current simulation time. The simulation clock advances by zero — its value before and after the advance is the same. Thus, the CAP always follows the CIP. It’s just that when the CIP ends due to the instantiation of a zero-duration Combi, the CAP advances the clock by zero time.

A Stroboscope simulation is a cycle that alternates between the CIP and the CAP. The processing of one phase enables the other to take place. When there is nothing more to do at the current simulation time Stroboscope enters the CAP. The Activity instances that terminate during the processing of the CAP may release resources that enable some Combis to start. Stroboscope enters the CIP when the FEL contains no current events (after the completion of the CAP). The Combi instances that Stroboscope creates and places in the FEL during the CIP represent future events. Stroboscope will process these events later, during a CAP. After the CIP is complete there is nothing more to do at the current simulated time, thus Stroboscope enters the CAP again.

A simulation run can end in various ways. One simple example is when the resources in the model’s Queues are not enough to enable the instantiation of any Combi and the FEL is empty. This happens if the FEL is empty when Stroboscope enters a CAP. A simulation that stops in this manner does so due to a “lack of resources”.

In the context of the example shown in Figure 7, the simulation stops due to a lack of resources when all the soil in the system is stored in \textit{MovedSoil}, and all haulers and loaders are waiting in \textit{HaulersWait} and \textit{LoadersWait} for more soil to load.
Later sections discuss how to stop a simulation on the basis of a condition not related to the availability of resources. In the context of the example shown in Figure 7, the simulation could run until a certain number of haulers have dumped, (and/or) until the simulation clock reaches a certain value, or until any other specified condition is satisfied.

Notice that at the beginning of any simulation the FEL is empty. In Stroboscope it is not possible to initialize the FEL. For this reason, the simulation loop starts with the CIP. When Stroboscope enters the first CIP, it must be able to instantiate at least one Combi. Otherwise, the FEL remains empty and the simulation stops due to a lack of resources. This implies that a Stroboscope network must have at least one Combi, which in turn must be able to start at the beginning of the simulation.

Figure 8 illustrates a simplified version of the Stroboscope simulation loop. Appendix E contains a step by step trace through a short simulation run.

Figure 8 - Simplified Simulation Loop
4.9 Recap

- The simulation clock is a double precision floating point number that keeps the current time in a simulation.

- Each occurrence of an Activity is an Activity instance. An Activity instance exists only during the period of simulated time between its instantiation and its termination.

- The Future Events List (FEL) is a chronological list of all pending Activity instance terminations.

- Events in the FEL that are bound to occur at the current simulation time are called current events.

- The FEL is processed by sequentially terminating all Activity instances whose end-time equal the current value of the simulation clock (the current events).

- Processing the FEL is the most important thing that Stroboscope does while running a simulation model. The FEL must be processed immediately whenever it contains current events.

- Just before the termination of an Activity instance, Stroboscope removes the terminating instance from the FEL.

- During the termination of an Activity instance, Stroboscope will instantiate all the Normals that are successors to the terminating Activity. The instantiated Normals are inserted in the FEL.

- Stroboscope enters the Combi Instantiation Phase (CIP) at the beginning of the simulation and whenever it is done processing the FEL.

- During the CIP, Stroboscope scans and starts the Combis in a model until it is not possible to start any Combi or until the instantiation of a zero-duration Combi inserts a current event in the FEL.
• Stroboscope needs to scan a Combi (check for necessary startup conditions) in order to instantiate it.

• By default, Stroboscope will instantiate a Combi if none of the Queues that precede it are empty.

• When Stroboscope instantiates a Combi, it inserts the instance in the FEL.

• Upon completion of the CIP, Stroboscope advances the clock to the time of the earliest of the future events. This time corresponds to the end-time of the Activity instance at the head of the FEL.

• The simulation ends due to a lack of resources if upon completion of the CIP Stroboscope encounters an empty FEL.

• The advance of the simulation clock changes the status of some events in the FEL from “future” to “current”. Stroboscope proceeds to process the FEL immediately.

• The process of advancing the clock and proceeding to process the FEL is called the Clock Advance Phase or CAP.

• Stroboscope processes a simulation by alternating between the CIP and CAP.
Chapter 5
Creating and Running Simple Models

Simulations are not meant to be done by hand. The detailed trace through a run of a short simulation in Appendix E makes that evident.

The network and those details of a model that are not shown on the network drawing need to be conveyed to the computer for processing. The standard way to convey a model to the computer is through a text input file written in Stroboscope’s own language. The result of processing a Stroboscope model is one or more output files.

This chapter introduces several topics that will allow us to prepare files for simulation on a computer, to run simulations, and to interpret the results. The following topics are discussed:

- Stroboscope statements and comments

  - Generic format

- Kinds of arguments:

  - User-defined identifiers

  - Expressions

  - Text strings

- Statement classification:

  - Element definition statements
- Element attribute statements
- Control statements
- Processing a Stroboscope source file and interpreting the standard report
- Variables:
  - Pre-defined system-maintained variables that access the state of the simulation
  - User-defined variables for parameterization of models and simplification of expressions
- Producing custom output
- Logical values, expressions, and operators
- Specifying conditions to stop a simulation before it reaches a ‘lack of resources’ condition
- Source file organization

### 5.1 Structure of Stroboscope Model Files

Stroboscope model files are composed of statements and comments. The generic format of a statement is as follows:

```
STATEMENTKEYWORD [Arg1] [Arg2] [...] [ArgN];
```

where `STATEMENTKEYWORD` is the statement keyword, and `Arg1`, `Arg2`, `...`, `ArgN` are arguments to the statement. The semicolon at the end of each statement is required. The statement keyword and its arguments, if any, are separated by white-space (spaces, tabs, and line breaks). It is not necessary to place white-space between the last argument (or the statement keyword for statements that take no arguments) and the ending semicolon.
The Stroboscope language is case sensitive. This means that Stroboscope distinguishes between uppercase and lowercase letters. This applies to statement keywords as well as to any arguments.

The statement keyword gives the entire statement a name. For example, the Queue statement begins with the QUEUE keyword. Statement keywords are always completely capitalized. The available statement keywords are defined by the Stroboscope language. They are listed and summarized in the “Statements” section of the reference (Appendix A).

The present discussion is about the structure of statements in general and not about specific statements. Specific statements will be introduced as appropriate in what follows.

The arguments to statements can be of four types: user-defined identifiers, expressions, text strings, and Stroboscope-defined identifiers.

**User-defined identifiers**

User-defined identifiers are names that represent specific modeling elements in a particular simulation model. Such names are chosen by the user to represent resource-types, network nodes and links, variables, files, etc. These identifiers must not begin with a digit (number), (nor) contain spaces, nor include any of the following characters:

\[ $ < > \{ \} \{(\}\}\{\}\{\}\{!\}\{-\}\{^\}\{/\}\{\}\{\}\{\}\{+\}\{=\}\{\&\}\{|\}\{?\}\{:\}\{''\}\]

Identifiers can be arbitrarily long, although the Stroboscope ‘canned’ output requires identifiers shorter than 15 characters in length for proper formatting. The choice of identifiers is largely a matter of style. In general, identifiers should be meaningful but not excessively long. Good examples are some of the identifiers used previously, such as *HaulersWait*, *LoadersWait*, and *MovedSoil*. Notice that these identifiers take advantage of capitalization to stress the different words in each name.

An identifier can refer to only one modeling element. For example, the identifier *HaulersWait* cannot refer to an Activity and a Queue simultaneously.

Because Stroboscope is case-sensitive, an identifier must be used consistently with respect to capitalization throughout a source file. It is acceptable to use identifiers that differ only in capitalization. In this case, each identifier refers to a different modeling element. For example, *hauler* could refer to a resource type, and *Hauler* to
another resource type, even though they differ only in the capitalization of their first letter.

**Expressions**

Stroboscope expressions are similar to expressions in other programming languages, spreadsheets, and database managers. They are composed of variables, constants, operators and function calls. A list and explanation of the Stroboscope operators and their precedence appears in the “Operators” section of the reference (Appendix A). A list and explanation of the Stroboscope built-in functions appears in the “Functions” section of the reference. Some specific operators and functions that are peculiar to Stroboscope will be introduced as needed.

When an expression is used as an argument to a statement, it must not contain white-space. If it does, the entire expression must be enclosed in single quotes. Otherwise, Stroboscope will interpret the white-space as an argument separator. This would cause a long expression to be treated as two or more shorter expressions, each probably syntactically incorrect. The following are examples of expressions that can be used in statements:

```
Beta[34.5,10]/RoadRoughness
'Beta[34.5,10] / RoadRoughness'
'Beta[ 34.5, 10 ] / RoadRoughness'
```

**Strings**

Some Stroboscope statements require text strings as arguments. Strings must be enclosed in double quotes regardless of whether they contain white-space or not. The quotes serve a dual purpose. When the string contains embedded white-space, they tell Stroboscope that the embedded white-spaces are not argument separators. Even strings that do not contain white-space must be enclosed in double quotes because some arguments can be either text, an expression, or an identifier. When the double quotes are missing, Stroboscope interprets the argument as an expression or identifier. The following examples are strings:

```
“Stroboscope”
“This line of text is a string”
“This line and the next two are part
of the same string. The embedded line-breaks are also part of the string”

**Multiple statements on the same line**

It is possible to place more than one statement in the same line. The following hypothetical statements:

```plaintext
STATEMENT Identifier1 “text argument”; OTHERSTATEMENT ‘expression’;
```

are exactly equivalent to the following statements:

```plaintext
STATEMENT Identifier1 “text argument”; OTHERSTATEMENT ‘expression’;
```

**Comments**

Comments in Stroboscope begin with a forward slash ‘/’ and continue until the end of the line. All white-space preceding the ‘/’ is ignored. Comments can be placed on lines by themselves, or after the semicolon that marks the end of a statement. The following is an example of commented code:

```plaintext
/ Comment on a line by itself
SOMESTATEMENT SomeArgument; / comment in same line as statement
/ indented comment on line by itself
/ Note that the comment immediately after the hypothetical statement
does not start on a separate line. Also note that blank lines are
allowed and that comments do not need to start on the first column.
```

**Simulation model file processing**

Stroboscope will read and execute the statements in an input file one by one, ignoring any comments. This is done in the order in which the statements appear in the file. Stroboscope will not go on to the next statement until it has completely executed the previous one. The execution of a statement does not necessarily imply an immediate action. It means that Stroboscope recognizes or takes into consideration the statement.

Stroboscope statements are classified in three groups:

- element definition statements,
- element attribute statements, and
- control statements.
The classification of a statement is based on what it means to ‘execute’ it. A discussion of each type of statement follows.

### 5.1.1 Element Definition Statements

Element definition statements define modeling elements such as resource types, Activities, links and Variables. The keyword (i.e., the first word in the statement) for these statements is usually the generic name for elements of that category. For example, the NORMAL statement is used to define Normals, the QUEUE statement is used to define Queues, the LINK statement is used to define links, and the VARIABLE statement is used to define Variables.

In most cases, the first argument of an element definition statement is the identifier for the element being defined. Some elements require additional arguments.

When Stroboscope executes an element definition statement, it creates an object to represent the element. Stroboscope will issue a compilation error if a subsequent statement defines another element with the same identifier. Stroboscope will also issue a compilation error if the identifier for a modeling element is used in a statement prior to the definition of the element.

As mentioned earlier, both bulk and discrete (uniquely identifiable) resources can be easily represented in Stroboscope. This and the next several chapters use only bulk resources. Sometimes bulk resources represent resources that are really discrete. This is accomplished by using them only in integer quantities.

Bulk resource types are called “Generic Resources” in Stroboscope. Generic resources are of “Generic Resource Types”. A generic resource type is defined with the GENTYPE statement:

```plaintext
Syntax: GENTYPE GenTypeName;
Example: GENTYPE Hauler;
```

where GenTypeName is the name of the generic resource type being defined. Note that the GENTYPE statement defines the resource type; it does not define a resource of the type.

Queues are defined with the QUEUE statement:
Syntax: QUEUE QueueIdentifier ResourceTypeIdentifier;
Example: QUEUE HaulersWait Hauler;

where QueueIdentifier is the name of the Queue being defined, and
ResourceTypeIdentifier is the name of a resource type that has been defined previously.
Notice that a Queue can hold resources of only one type.

Combis are defined with the COMBI statement:

Syntax: COMBI CombiIdentifier;
Example: COMBI Load;

where CombiIdentifier is the name of the Combi being defined.

Normals are defined with the NORMAL statement:

Syntax: NORMAL NormalIdentifier;
Example: NORMAL Haul;

where NormalIdentifier is the name of the Normal being defined.

Note that Normals and Combis are not resource-specific nodes (i.e., their
instances can hold resources of any type), whereas Queues are resource-specific (i.e.,
specific Queues can hold resources of one type only). For this reason, a resource type is
required as an argument in the definition of a Queue.

Links are defined with the LINK statement, which takes one of two forms:

Syntax: LINK LinkName PredecessorNode SuccessorNode;
Example: LINK HL1 HaulersWait Load;

Syntax: LINK LinkName PredecessorNode SuccessorNode ResourceType;
Example: LINK HL2 Load Haul Hauler;

where LinkName is the identifier for the link, and PredecessorNode and SuccessorNode
are the identifiers for the predecessor and successor nodes, respectively. When the second
form is used, ResourceType is the name of the resource type that flows through the link.

In all cases, Stroboscope must know the specific type of resource that can flow
through a particular link.

When either the predecessor or successor node are type-specific (e.g., a Queue),
the link must be defined with the first form of the LINK statement. Stroboscope
determines the type of the link automatically by examining the type-specific node. If they both are type-specific, their types must match.

When neither of the nodes is type-specific, Stroboscope has no way of telling the type of resource that flows through the link by examining the resource type of the linked nodes. In this case, the resource-type of the link must be specified explicitly using the second form of the LINK statement.

**Example network**

The statements presented so far can be used to write the Stroboscope code that defines the simple earth-moving operation network of Figure 7 on page 27.

```plaintext
// Define the resource types
GENTYPE Soil;
GENTYPE Loader; // for now we will discretize a bulk resource
GENTYPE Hauler; // for now we will discretize a bulk resource

// Define the Queues.
// Note that the resource types of the resources held by a Queue
// must be defined before the Queue itself.
QUEUE SoilToMove Soil;
QUEUE MovedSoil Soil;
QUEUE LoadersWait Loader;
QUEUE HaulersWait Hauler;

// Define the Activities
COMBI Load;
NORMAL Haul;
NORMAL Dump;
NORMAL Return;

// Define the Links
// Note that in all cases, the nodes being linked must have been defined
// before the Link itself. This also applies to the resource type of Link

// Soil Path
LINK SL1 SoilToMove Load; // resource type inferred from SoilToMove
LINK SL2 Load Haul Soil; // resource type required
LINK SL3 Haul Dump Soil;
LINK SL4 Dump MovedSoil;

// Loader Cycle
LINK LD1 LoadersWait Load;
LINK LD2 Load LoadersWait;

// Hauler Cycle
LINK HL1 HaulersWait Load;
LINK HL2 Load Haul Hauler;
LINK HL3 Haul Dump Hauler;
LINK HL4 Dump Return Hauler;
LINK HL5 Return HaulersWait;
```
The simple earth-moving operation network defined above contains all the information shown in Figure 7.

5.1.2 Attribute Statements

Stroboscope elements have a number of attributes that define how they behave during a simulation. When an element is created, all its attributes have default values. Combis, for example, have attributes such as Duration, Semaphore, and Priority (only the Duration attribute has been discussed so far). Attributes take the form of constant numbers, expressions, or actions (the latter will be discussed later).

Whenever necessary, element attributes can be changed through element attribute statements. When these statements are executed they do not create anything new. They modify attributes that already exist.

The keyword for element attribute statements usually corresponds to the name of the attribute. Combis and Normals have a Duration attribute whose default is the constant zero. It can be changed with the DURATION statement:

Syntax: `DURATION ActivityName DurationExpression;`
Example: `DURATION Load Normal[24,6];`

where `ActivityName` is the name of the Activity (be it a Normal or a Combi) whose duration attribute is being changed, and `DurationExpression` is an expression that the Activity will use to determine the duration of its instances.

The execution of the DURATION statement does not create any Activity instance, nor does it make Stroboscope evaluate the expression. It just tells Stroboscope that it should use the provided expression to calculate the duration of each of the instances of the Activity. Thus, every time an instance of the Activity is created during simulation, Stroboscope will evaluate the expression and use the result as the duration of the instance.

An element attribute can change several times during the processing of a simulation model. In contrast to element definition statements, Stroboscope does not issue a compilation error when it finds a re-definition of an element attribute. The following code fragment illustrates this:
COMBI Load; /Load Combi created. Its Duration attribute is '0'
/ other statements that make the simulation run
/ (these statements have not been presented
/ .
/ .
/ .
/ all instances of Load that were created had a duration of 0

DURATION Load Pert[12,17,19]; / Change duration attribute for Load.
/ Pert is a built-in function used to
/ sample from a Beta distribution

/ other statements that make the simulation run
/ .
/ .
/ .
/ instances of Load created after the change of attribute have
/ durations sampled from the Pert distribution

DURATION Load 12; / change duration attribute of Load again

/ other statements that make the simulation run
/ .
/ .
/ .
/ all the instances of Load created after the change of attribute
/ have a duration of 12

Although Stroboscope modeling elements have numerous attributes (all with
default values that can be changed through element attribute statements), they will be
introduced as needed in the following discussion. For the time being, the only attribute of
interest is Duration. Later on, as attributes are introduced, the statements that change
their default behavior will be described.

Example continued

The following statements continue the simple earth-moving operation of Figure 7
by defining durations for the various Activities:

/ Pertpg and Normal are built-in probabilistic sampling functions
/ Note that function arguments are enclosed in square brackets and
/ not in parentheses.
DURATION Load Pertpg[0.9,2.6,4.5];
DURATION Haul Normal[5,1];
DURATION Dump 0.5;
DURATION Return Normal[4,1];
5.1.3 Control Statements

The element definition and element attribute statements discussed in the previous two sections define a model and the way the model behaves, but they do not initialize a model’s resources or perform a simulation. These statements have no immediate effect when they are executed.

The statements that cause an immediate effect upon execution are the control statements. Such statements create resources and place them in Queues, start a simulation, display output, and perform several other functions that will be discussed later. The keyword for most control statements is the name of the action implied by the statement. Thus, control statements are usually verbs or abbreviations of verbs such as INIT (for initialize), SIMULATE, PRINT, DISPLAY, REPORT, etc. Repeated executions of control statements simply instruct Stroboscope to perform the action several times.

The INIT statement is used to create resources and to place them in Queues:

Syntax: INIT GenQueue PositiveRealExpression;
Example: INIT SoilToMove SoilVolume/HaulerCapacity;

where GenQueue is the name of a Queue for generic resource types, and PositiveRealExpression is an expression that gets evaluated immediately to determine the amount of resource that is created and placed in GenQueue. PositiveRealExpression must return a non-negative real value, otherwise Stroboscope will issue a compilation error (it is impossible to create a negative amount of resources).

The type of the resource that is created with the INIT statement is determined by the resource type of the Queue. The PositiveRealExpression expression is evaluated only at the time of execution of the INIT statement. The INIT statement adds to the existing contents of a Queue, it does not specify the initial contents (although this is the case if the Queue is empty before the execution of the INIT statement). The following code fragment illustrates the INIT statement:

GENTYPE Cash; / Cash is a bulk, generic, resource type
QUEUE CashInReg Cash; / CashInReg holds Cash, but at creation is empty
INIT CashInReg 2000; / CashInReg now has $ 2000
INIT CashInReg Log[100]; / CashInReg gets another $ 2, now it has $ 2002
Finally, how to make the simulation run!

The SIMULATE control statement is used to start running a simulation:

Syntax: SIMULATE;
Example: SIMULATE;

The SIMULATE statement takes no arguments. When Stroboscope executes this statement, it starts running the model as it has been defined so far. It uses only the modeling elements defined prior to the statement, and with their current attributes. When the simulation stops due to a lack of resources, Stroboscope proceeds with the next statement in the source file.

When Stroboscope executes the SIMULATE statement, the simulation will only stop due to a lack of resources or due to a runtime error. Therefore, this statement must be used very carefully. If the logic of the model does not conduce to a lack of resources, the simulation will run forever.

The REPORT control statement instructs Stroboscope to display a summary of the state of the model at the time at which it executes the statement. The summary consists mainly of statistics collected during simulation.

If the REPORT statement is executed before any simulation has taken place (before a SIMULATE statement), most of the report will be empty (no statistics have been collected).

The REPORT statement is usually placed after a SIMULATE statement to show the results of the simulation. The REPORT statement takes an optional argument that indicates the device to which Stroboscope should print the report:

Syntax: REPORT [OutFile];
Example: REPORT;

where OutFile should be substituted with the name of the output device and the square brackets indicate that the OutFile argument is optional. A discussion of the optional argument will be deferred to later sections.
Example concluded

Given the previously described control statements, and assuming that the earth-moving operation starts with 3 loaders, 10 haulers, and 1000 hauler-loads of soil; the Stroboscope code for the model concludes with the following statements:

/ Create resources and populate Queues
INIT SoilToMove 1000;
INIT LoadersWait 3;
INIT HaulersWait 10;

/ Run the simulation until all the earth is moved
SIMULATE;

/ Present the results of the simulation
REPORT;

5.2 Executing a Model and Interpreting the Standard Report

The text input file prepared in the preceding section is all Stroboscope needs to run the simulation of the earth-moving operation. Stroboscope is currently implemented in several forms: a command line application, an Integrated Development Environment (IDE), a dialog based driver, and a Graphical User Interface (GUI). All versions are 32 bit MS Windows applications (for the Windows NT 3.51 and Windows 95 operating systems). All except the command line application can run under MS Windows 3.1 through the Win32S subsystem (version 1.25 or later, with OLE 2 support).

The Stroboscope Simulation System IDE complies with all the user interface guidelines for MS Windows applications. It contains extensive and complete on-line help that explains how to use the program to create and run simulations, and a summary of the Stroboscope language. Users familiar with MS Windows 3.1 should therefore have no problems using the IDE.

The Graphical User Interface is interactive and intuitive. It works by dragging and dropping modeling elements with the mouse. Thereafter, double-clicking a shape (such as Queue or a link) brings a dialog box to define or change its attributes. The
model can be run by double-clicking a button. An input file can be generated by double-clicking another button.

The command line version of Stroboscope is typically run from the Windows NT or Windows 95 command prompt. The syntax for the command in Windows NT is:

STROBOS inputfileName [c] [d:StdTrace] [f] [1>StdOutput] [2>StdError]

inputfileName file name of Stroboscope source for simulation
c produce clean output
d create trace output
StdTrace file name of Trace output, if omitted uses StdOutput
f run a fast simulation (no call trace on runtime errors)
StdOutput file name of Standard output, if omitted goes to console
StdError file name of Error output, if omitted goes to console

The syntax for the command in Windows 95 is similar, except that error output always goes to the console:

STROBOS inputfileName [c] [d:StdTrace] [f] [>StdOutput]

inputfileName file name of Stroboscope source for simulation
c produce clean output
d create trace output
StdTrace file name of Trace output, if omitted uses StdOutput
f run a fast simulation (no call trace on runtime errors)
StdOutput file name of Standard output, if omitted goes to console

Stroboscope source files typically have an ‘.str’ extension, and Stroboscope output files typically have an ‘.sto’ extension. Assuming that the source written in the previous sections was saved to file ‘haul.str’, the following command could be issued at the Windows NT command prompt:

strobos haul.str 1>haul.sto

This command will instruct Stroboscope to process the model defined in file ‘haul.str’, and to write the results of the processing to file ‘haul.sto’. The Stroboscope banner, status while processing the model, and any errors encountered are still displayed in the screen as things happen:
Stroboscope Simulation System
Copyright (c) Julio C. Martínez 1994-1995. All rights reserved.

Stroboscope Simulation System Command Line Driver Version 1, 0, 0, 2
Stroboscope Simulation Engine Version 1, 0, 2, 9
Stroboscope Model haul.str (2046493144)
Simulating(9111964)... Stopped at 1244.14 (Lack of Resources)

Stroboscope displays the ‘Simulating(xxxxxx)’ message shown above when it executes a SIMULATE statement in the source file. When it finishes executing a SIMULATE statement, Stroboscope displays the ‘Stopped at xxxx (reason)’ message.

The file ‘haul.sto’ that Stroboscope produces is shown in Figure 9. The first line contains the name of the model, with the seed value of the default random number generator at the start of processing in parentheses:

Stroboscope Model haul.str (2044167314)

The seed of the default random number generator is included because it can be used to reproduce the simulation run exactly. The second line is blank. These first two lines are the header of the output, and will show on any model processed by Stroboscope (unless the [c] option is used in the command line version of Stroboscope).

Stroboscope created the third line and almost all the rest of file ‘haul.sto’ when it executed the REPORT statement. The first line in the report indicates the value of the simulation clock at the time of the report:

Statistics report at simulation time 1244.14

This line is followed by two blank lines and several tables containing statistics about the simulated process.

The ‘AvWait’ column shows the average duration of a visit to the Queue. Note that several different resources can enter the same Queue a different number of times (and each visit can have a different duration). Therefore, the value shown under ‘AvWait’ corresponds to the average visit among all resources. The line for LoadersWait indicates that loaders had to wait (were idle), on average, 1.08 minutes between loadings.
Stroboscope Model haul.str (2046493144)

Statistics report at simulation time 1244.14

<table>
<thead>
<tr>
<th>Queue</th>
<th>Res</th>
<th>Cur</th>
<th>Tot</th>
<th>AvWait</th>
<th>AvCont</th>
<th>SDCont</th>
<th>MinCont</th>
<th>MaxCont</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaulersWait</td>
<td>Hauler</td>
<td>10.00</td>
<td>1010.00</td>
<td>0.39</td>
<td>0.32</td>
<td>0.86</td>
<td>0.00</td>
<td>10.00</td>
</tr>
<tr>
<td>LoadersWait</td>
<td>Loader</td>
<td>3.00</td>
<td>1003.00</td>
<td>1.08</td>
<td>0.87</td>
<td>0.93</td>
<td>0.00</td>
<td>3.00</td>
</tr>
<tr>
<td>MovedSoil</td>
<td>Soil</td>
<td>1000.00</td>
<td>1000.00</td>
<td>621.52</td>
<td>499.56</td>
<td>291.48</td>
<td>0.00</td>
<td>1000.00</td>
</tr>
<tr>
<td>SoilToMove</td>
<td>Soil</td>
<td>0.00</td>
<td>1000.00</td>
<td>614.53</td>
<td>493.94</td>
<td>291.40</td>
<td>0.00</td>
<td>1000.00</td>
</tr>
</tbody>
</table>

Activity

<table>
<thead>
<tr>
<th>Activity</th>
<th>Cur</th>
<th>Tot</th>
<th>1stSt</th>
<th>LstSt</th>
<th>AvDur</th>
<th>SDDur</th>
<th>MinD</th>
<th>MaxD</th>
<th>AvInt</th>
<th>SDInt</th>
<th>MinI</th>
<th>MaxI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dump</td>
<td>0</td>
<td>1000</td>
<td>6.74</td>
<td>1239.38</td>
<td>0.50</td>
<td>0.00</td>
<td>0.50</td>
<td>0.50</td>
<td>1.23</td>
<td>1.04</td>
<td>0.00</td>
<td>6.92</td>
</tr>
<tr>
<td>Haul</td>
<td>0</td>
<td>1000</td>
<td>2.41</td>
<td>1234.25</td>
<td>4.94</td>
<td>0.98</td>
<td>1.69</td>
<td>8.21</td>
<td>1.23</td>
<td>0.99</td>
<td>0.00</td>
<td>6.94</td>
</tr>
<tr>
<td>Load</td>
<td>0</td>
<td>1000</td>
<td>0.00</td>
<td>1231.33</td>
<td>2.65</td>
<td>1.09</td>
<td>0.13</td>
<td>5.83</td>
<td>1.23</td>
<td>0.92</td>
<td>0.00</td>
<td>5.35</td>
</tr>
<tr>
<td>Return</td>
<td>0</td>
<td>1000</td>
<td>7.24</td>
<td>1239.88</td>
<td>3.96</td>
<td>1.03</td>
<td>0.94</td>
<td>7.57</td>
<td>1.23</td>
<td>1.04</td>
<td>0.00</td>
<td>6.92</td>
</tr>
</tbody>
</table>

[several empty tables omitted]

Contents of the Future Events List at simulation time 1244.14

<table>
<thead>
<tr>
<th>Instance</th>
<th>Start</th>
<th>End</th>
</tr>
</thead>
</table>

Total Number of Named Objects : 26
Total Number of Variables : 0
Total Number of Statements : 32

Execution Time = 0.59 seconds
Processor Time = 0.45 seconds

Figure 9 - Output Produced by the REPORT Statement
The ‘AvCont’, ‘SDCont’, ‘MinCont’, and ‘MaxCont’ columns show the time-weighted average, standard deviation, minimum, and maximum content of the Queue, respectively. The line for LoadersWait shows that the average number of loaders that were idle during the simulation was 0.87. It also shows there was much variation in the length of the queue (waiting line), because the standard deviation (0.93) was very high in relationship to the average. There were times during which LoadersWait was empty (because the minimum content was 0). At the beginning and end of the simulation LoadersWait contained all 3 loaders in the system.

The second table in the report is the Activity table. The first column shows the name of the Activity. The ‘Cur’ column shows the number of instances of the Activity that were taking place at the time of the report. All Activities in this run have a value of 0 because this simulation ended due to a lack of resources and the FEL was empty. The ‘Tot’ column shows the total number of instances of the Activity that were created during the simulation, including those that are still in the FEL. The report shows that every Activity was instantiated 1000 times, once per hauler cycle. The ‘1stSt’ column shows the simulation time at which the first instance of the Activity started. The line for the Return Activity indicates that the first hauler to return started doing so (finished dumping) 7.24 minutes after the earth-moving process started. The ‘LstSt’ column shows the simulation time at which the last instance of the Activity started. The line for Dump shows that the last hauler to dump started doing so at simulation time 1239.38.

The next group of columns in the Activity table shows statistics about the duration of the instances of each Activity. The ‘AvDur’, ‘SDDur’, ‘MinD’, and ‘MaxD’ columns show the average, standard deviation, minimum, and maximum duration of the instances, respectively. The entry for Return shows that the average duration of its instances was 3.96 minutes, and that the standard deviation was 1.03. These numbers correspond closely to the parameters of the normal distribution from which Stroboscope sampled the durations. The shortest time it took a hauler to return was 0.94 minutes, and longest the time was 7.57 minutes.

The next group of columns show statistics about the time between successive instantiations of Activities. The ‘AvInt’, ‘SDInt’, ‘MinI’, and ‘MaxI’ columns show the average, standard deviation, minimum, and maximum time interval between successive
instantiations. The line for Return shows that on average a hauler would start returning every 1.23 minutes, with a standard deviation of 1.04 minutes. The longest time between successive starts of the Return Activity was 6.92 minutes. The ‘0.00’ under the ‘MinI’ column appears to indicate that in at least one occasion two haulers started to return at the same time. However, due to randomness it is probabilistically impossible for two haulers to start to return simultaneously. In this run the minimum inter-instantiation time was actually 0.0016 minutes; this number is rounded to 0.00 because the report uses two decimal places.

The standard report contains several other tables, all of which are empty in this example. These tables show diverse statistics about simulation elements not yet discussed.

The last table shows the contents of the FEL at the time of the report. This simulation ended due to a lack of resources and therefore the FEL was empty.

The standard report concludes with information regarding the source file. The number of named modeling elements and variables defined before the execution of the report; and the total number of statements in the source file (regardless of whether the statements have already been executed or not).

The ‘haul.sto’ file that Stroboscope produced concludes with the output footer. The output footer is a dashed line followed by the execution time and processor time consumed during the simulation. The processor time is shown only when the model is processed in a multitasking operating system such as Windows NT. The output footer will show on any model processed by Stroboscope (unless the [c] option is used in the command line version of Stroboscope).

5.3 Variables

The example expressions used so far consist entirely of functions, operators, and numbers. Stroboscope also allows the use of symbolic names to represent numbers or expressions in other expressions. This section describes a class of symbolic names called Variables. The name ‘Variable’ may be misleading because Stroboscope Variables are not storage locations for values (other Stroboscope elements, to be introduced later, are).
Stroboscope Variables are more like functions that take no arguments or like formulas in a spreadsheet program.

Some variables are defined and maintained by the system and are always available for use in expressions. Other variables are user-defined for each model with the purpose of localizing problem parameters and simplifying expressions.

5.3.1 Pre-Defined System-Maintained Variables

The information presented in the standard report gives us a snapshot of the state of the simulated process at the time of execution of the report. This includes information that describes the current conditions of the simulated process (e.g., the current content of a Queue, or the value of the simulation clock) as well as statistics that describe the performance of the system up to the time of the report (e.g., the average duration of instances of an Activity or average content of a Queue).

Most of the information in the standard report is of a dynamic nature. Information changes while a simulation model is processed. Sometimes the information changes without a change in simulation time. For example, the current number of instances of an Activity is not the same before and after an instance of the Activity is created, even though the instants just before and just after its creation exist at the same simulated time.

The system defines and maintains variables that provide comprehensive and up-to-date access to the state of the simulation. This includes all the information in the standard report, as well as other aspects of the state of the simulation.

Global variables access information, such as the information contained in the standard report, that is available all the time and in any context. Instance variables access information that is valid only during the instantiation or termination of Activity instances, such as the duration of a particular instance of an Activity.

A list of the Stroboscope pre-defined variables appears in the “System-Maintained Variables” section of the reference (Appendix A).
5.3.1.1 Global Variables

Global variables access information about the modeled process as a whole as well as information related to particular modeling elements.

Global variables that access an aspect of the modeled process that is not related to a particular modeling element consist of the name of the variable by itself. The most commonly used variable of this class is SimTime, which returns the current value of the simulation clock. Whenever SimTime is used in an expression, Stroboscopes will substitute it for the value of the simulation clock at the time of the evaluation of the expression.

Another variable that is not related to a particular modeling element (at least explicitly) is the variable CurSeed, which returns the current value of the seed for the default random number generator.

Most global variables access information related to one or more modeling elements. Stroboscopes create names for these variables by using the names of the modeling elements involved and the name of the variable, separated by periods. For example, Stroboscopes create the variable Dump.CurInst to provide access to the current number of instances of the Dump Activity. The variable Dump.CurInst can be used in any expression that appears after the definition of Dump. Whenever Stroboscope evaluates the expression (before, during, or after simulation), it substitutes Dump.CurInst for the number of instances of Dump that currently exist.

Stroboscopes define and maintain variables that access the information presented in the standard report’s Queue table. Given a Queue named QueueName, the information under the ‘Cur’, ‘Tot’, ‘AvWait’, ‘AvCont’, ‘SDCont’, ‘MinCont’, and ‘MaxCont’ columns of the report can be accessed with the variables QueueName.CurCount, QueueName.TotCount, QueueName.AveWait, QueueName.AveCount, QueueName.SDCount, QueueName.MinCount, and QueueName.MaxCount, respectively. The variable LoadersWait.AveWait, for example, returns the average waiting time at Queue LoadersWait.

Stroboscopes also define and maintain variables that access the information presented in the standard report’s Activity table. Given an Activity named ActName, the

5.3.1.2 Instance Variables

In addition to global variables, Stroboscope creates and maintains instance variables. Instance variables are related to a specific instance of a specific modeling element (Activities, so far). Stroboscope provides access to instance information only when the modeling element is in context.

An Activity is in context when one of its instances is being created or terminated. During the termination of an Activity, the terminating Activity and all the Normals that are successors to it are in context. During the instantiation of a Combi, only the Combi is in context. Queues are always in context. The global pre-defined system-maintained variable ActivityName.InContext returns TRUE if Activity AcitivityName is in context.

Instance variables for Activities access information such as the duration or instance number of the instance, as well as information regarding the resources held by the instance.

For an Activity named ActivityName, the variable ActivityName.Duration returns the duration of the ActivityName instance being created or terminated. Stroboscope will issue a runtime error if it has to access the ActivityName.Duration instance variable when ActivityName is not starting or ending (i.e., it is not in context).

Similarly, for an Activity named ActivityName, the variable ActivityName.Instance returns the instance number of the instance being created or terminated. When an Activity is starting, the ActivityName.Instance variable is the same as the global variable ActivityName.TotInst. This is not the case when an Activity instance is being terminated (unless the Activity never has overlapping instances).
Given an Activity named ActivityName, and a resource type named ResTypeName, the variable ActivityName.ResTypeName.Count returns the amount of resource of type ResTypeName held by the instance of ActivityName that is currently in context. For example, when Dump is starting and has already acquired its resources from Haul, the variables Dump.Soil.Count and Dump.Hauler.Count both return the value 1, whereas the variable Dump.Loader.Count returns the value 0 (Dump uses no loaders).

### 5.3.2 User-defined Variables

It is also possible to define our own names for particular numerical values or expressions of interest by using the VARIABLE statement:

**Syntax:**
```
VARIABLE VariableName ExpressionThatCalculatesValueOfVariable;
```

**Example:**
```
VARIABLE DaysInWeek 7;
```

```
VARIABLE DayOfWeek Mod[SimTime/60/24,DaysInWeek];
```

In the simplest of cases a variable can be defined as a number:

```
/ define variable to hold the number of haulers in the system.
VARIABLE NumberOfHaulers 10;

/ from now on we can use NumberOfHaulers instead of the number 10
/ ............
INIT HaulersWait NumberOfHaulers; / populate HaulersWait
```

Variables of this type are particularly useful for the specification of problem parameters. These parameters are likely to be used in several expressions throughout a model file. The use of a Variable to hold the value allows making changes easily in only one location. All other references to the number are updated automatically.

Variables are not limited to simple numbers. A Variable can be a synonym for any expression. The expression itself can reference other user-defined or system-maintained variables. Referenced Variables can themselves use other Variables. There is no limit to the level of nesting. Every time Stroboscope needs to use a Variable, it recomputes the Variables used to define it. This is done recursively. As a result, Stroboscope Variables are always up to date.
This recalculation mechanism is similar to an automatically recalculated spreadsheet. Stroboscope variables resemble locked spreadsheet cells. The formula for one cell can reference a second cell. The second cell can in turn refer to a third cell, etc. When the value of the third cell changes, the value of the second cell is updated, and the value of the first cell is updated in turn. Any reference to the first cell takes into consideration the values of the second and third cells. Because the cells are locked, the formula that defines them cannot be changed (but the formula may return different values at different times).

The following example clarifies this:

/ example incorporates learning curve for the erection of beams.  
/ only aspects necessary to illustrate user-defined variables are shown

/ we assume expected duration of the Erect Activity to be defined by 
/ an equation of the form d=a\(x^s\), where ‘a’ is the time 
/ required to erect the first beam, ‘x’ is the cumulative number of 
/ beams erected so far, and ‘s’ is the slope constant of the learning 
/ curve in a log-log scale

/ the data available are the expected times for the 2\(^n\) and 3\(^n\) erections
VARIABLE TimeFor2ndBeam 10.80; / expected time in hours
VARIABLE TimeFor3rdBeam 10.25; / expected time in hours

/ determine parameters for equation of expected duration of Xth erection
VARIABLE Slope Log[TimeFor2ndBeam/TimeFor3rdBeam]/Log[2.0/3.0];
VARIABLE TimeFor1stBeam TimeFor2ndBeam/2^Slope;

/ expected duration of the Xth erection
VARIABLE ExpErectDur TimeFor1stBeam*Erect.TotInst^Slope;

/ we also know the coefficient of variation of the real erection 
/ times with respect to the expected times:
VARIABLE VariationCoeff 0.05;

/ define the Activity for the erection, other modeling elements omitted
COMBI Erect;

/ the real duration deviates a little bit from the expected 
DURATION Erect ExpErectDur*Normal[1, VariationCoeff];

During the simulation of a model that incorporates the above code, Stroboscope will evaluate the Duration attribute of the Erect Activity every time it creates an instance of the Activity. The Duration attribute uses the ExpErectDur and VariationCoeff variables, so these are also evaluated. ExpErectDur in turn uses the TimeFor1stBeam, Erect.TotInst, and Slope variables, so those need to be evaluated too. This procedure will theoretically go on until everything is in terms of numbers.
Given the above code, Stroboscope knows that the values of the \textit{TimeFor1stBeam} and \textit{Slope} variables will never change. So it doesn’t actually recalculate them. Stroboscope evaluates them only once, and from then on remembers the result. This is not the case with the \textit{Erect.TotInst} variable, since this variable attains a new value every time the \textit{Erect} Activity starts. Conceptually, however, variables are always recalculated recursively. The cases mentioned above are simply optimizations and should not affect the basic concept of Stroboscope variables.

The \textsc{VARIABLE} statement is an element definition statement. When Stroboscope executes the statement it parses the expression and leaves it in a form that can be evaluated. The execution of the \textsc{VARIABLE} statement, when encountered during the processing of a simulation input file, does not actually evaluate the expression, nor does it assign any value to the variable being defined.

### 5.4 Displaying Custom Output

The output produced by the execution of the \textsc{REPORT} statement was shown in Figure 9. The standard report consists strictly of status and statistical information presented in a pre-defined format.

Usually, managerial decisions are made in terms of parameters that are derived from information in the standard report, or may depend on information that is not shown in the standard report. A typical example is when a process is gauged in terms of cost. It may be more meaningful, for example, to see the cost per cubic meter of soil moved, than it is to see the average waiting time of haulers at a Queue.

Stroboscope has facilities that allow the display of custom output so that derived calculations can be observed directly. Furthermore, it is possible to format output in a manner that is clearer or aesthetically more pleasing, that highlights results that are otherwise lost within the sea of numbers produced in the report for a large model, or that presents values with more precision than the standard report.

The \textsc{DISPLAY} control statement is the simplest method of producing custom output:
Syntax:  DISPLAY [String | Expression] [...];
Example:  DISPLAY "The answer is " TotalCost/NumberOfUnits;

where String is text enclosed in double quotes, and Expression is any valid Stroboscope expression. The square brackets indicate that the argument is optional. The symbol '|' indicates that either a String or Expression can be used, but not both. The ellipsis in the second set of square brackets indicates that the DISPLAY statement will accept any number of arguments in the format specified by the first set of square brackets. When more than one argument is used, each argument can be either a string or an expression independent of the other arguments.

When Stroboscope executes the DISPLAY statement, it simply echoes to the standard output device the arguments of the statement. Strings are displayed exactly as they appear in the argument list, including any embedded tabs or line breaks. Expressions are evaluated and the resulting value is displayed with up to 8 significant digits. Once all the arguments have been displayed, Stroboscope inserts a line break.

**Example extended**

The following set of statements could be placed after the REPORT statement in the example that concluded in section 5.1.3:

DISPLAY; /just a blank line
/D DISPLAY "
/ now let us display some information more accurately than in the report

DISPLAY "The variable Dump.MinInter has value : " Dump.MinInter;
DISPLAY "The variable Haul.MinInter has value : " Haul.MinInter;
DISPLAY "The variable Load.MinInter has value : " Load.MinInter;
DISPLAY "The variable Return.MinInter has value : " Return.MinInter;

Had these statements been included in file ‘haul.str’, then the last part of file ‘haul.sto’ would have looked like this:

```
......
Total Number of Statements : 32

The variable Dump.MinInter has value : 0.0016936928
The variable Haul.MinInter has value : 0.0027019025
```
The variable Load.MinInter has value : 0
The variable Return.MinInter has value : 0.0016936928

Execution Time = 0.59 seconds
Processor Time = 0.45 seconds

The above output explains why all the values under the ‘MinI’ column of the Activity report in file ‘haul.sto’ have the values ‘0.00’ (which could be misleading).

Other statements in Stroboscope, such as the PRINT statement, provide much more control over the precision and format with which values are displayed. These statements will be discussed later.

5.5 Logical Expressions, Values and Variables

Stroboscope relies heavily on logical expressions for the specification of certain element and model attributes. Stroboscope also provides system-maintained variables and operators that return logical values.

The purpose of model and element attributes that return logical values is to indicate a Yes/No or TRUE/FALSE response. All expressions in Stroboscope return double precision floating point values. In logical contexts, these numbers need to be interpreted as TRUE or FALSE. Stroboscope interprets the value 0 as FALSE and any other value as TRUE. When Stroboscope applies a logical operator, function, or system-maintained variable, it returns the value 0 to indicate FALSE and the value 1 to indicate TRUE.

Thus, there is a very important distinction between TRUE and FALSE. The value 0 is always FALSE, and FALSE is always 0. The relationship between TRUE and the value 1 is not the same. The value 1 is always TRUE, but so are the values 2, 3, 6.75, -0.000003, -45, 10000, etc. Stroboscope chooses to return the value 1, instead of any other value, to indicate TRUE. But Stroboscope interprets any non-zero value as TRUE (not just with the value 1).

Stroboscope offers several logical operators. The logical operator with the highest precedence (i.e., the one that is resolved first) is the ‘!’ (NOT) operator (see Table 17 on page 418). This is a unary operator (i.e., it is applied to one argument) that converts
TRUE values to FALSE, and FALSE values to TRUE. When the NOT operator is applied to any non-zero (TRUE) value, Stroboscope returns 0 (FALSE). When the NOT operator is applied to zero (FALSE), Stroboscope returns 1 (Stroboscope’s choice of TRUE). Note that the NOT operator has the same precedence as the Negation (change sign) operator ‘-’, which is also unary. Also note that the standard mathematical operators ‘^’ (Power), ‘*’ (Multiplication), ‘/’ (Division), ‘+’ (Addition), and ‘-’ (Subtraction), have lower precedence than the NOT operator. The following examples make this clear:

DISPLAY !21; /displays 0
DISPLAY !(-15); /displays 0,(unary ops must be separated by parenthesis)
DISPLAY !(-0.233345); /displays 0
DISPLAY !0; /displays 1
DISPLAY !(27/3-9); /displays 1
DISPLAY !27/3-9; /displays -9, watch operator precedence
DISPLAY !(121); /displays 1,(unary ops must be separated by parenthesis)

The logical operators next in precedence (these have lower precedence than the standard mathematical operators) are the comparison binary operators (i.e., they apply to two values). These are the ‘<’ (less than), ‘<=’ (less than or equal), ‘>’ (greater than), and (‘>=’ greater than or equal) operators. Like all Stroboscope logical operators, they return 0 (FALSE) or 1 (Stroboscope’s choice of TRUE). For example:

DISPLAY ‘21 > 20’; /will display the value 1
DISPLAY ‘21 <= 19+2’; /will display the value 1 (watch precedence !)
DISPLAY ‘14 > 22’; /will display the value 0
DISPLAY ‘(15 >= 5*3)*Log[100]’; /will display the value 2, (1*2)
DISPLAY !((14>=10+4)*(2100>5));/will display the value 0 (!(1*1)=!1=0)

The logical operators that follow in precedence are the equality binary logical operators ‘==’ (equal) and ‘!=’ (not equal). They also return values of 1 or 0:

DISPLAY 3==9/3; /will show the value 1
DISPLAY 3!=10/4; /will show the value 1
DISPLAY 3==5; /will show the value 0
DISPLAY (3==5)>-2; /will show the value 1 (0 > -2 is TRUE)
DISPLAY 3==5>-2; /will show the value 0 (3 is not equal to (5>-2 or 1)

At the next lower level is the ‘&’ (logical AND) operator, which returns TRUE only when both its operands are TRUE:
DISPLAY ‘3>5 & 1==1’; /will show the value 0 (3>5 is FALSE)
DISPLAY ‘3<5 & 1!=1’; /will show the value 0 (1!=1 is FALSE)
DISPLAY ‘3<5 & 1==1’; /will show the value 1 (both args are TRUE)

The ‘&’ operator is short circuited. This means that Stroboscope only evaluates the second operand of an ‘&’ operator if the first operand is TRUE:

/will display 0 without producing a divide by zero error
DISPLAY '3>5 & 10/0==1*10^99';
/will produce a divide by zero error
DISPLAY '10/0==1*10^99 & 3>5';

The last of the logical operators is the ‘|’ (OR). The ‘|’ operator returns the value FALSE only when both its operands are FALSE:

DISPLAY ‘2 | 3’; /will show the value 1, both 2 and 3 are TRUE
DISPLAY ‘2==4/2 | Sin[2.17]==28’; /will show the value 1, 2==4/2 is TRUE
DISPLAY ‘2==Cos[2] | Sin[2.17]==28’; /will show the value 0

The ‘|’ operator is also short circuited. This means that Stroboscope does not evaluate the 2nd argument if the 1st argument is TRUE:

/ will show the value 1 without producing divide by zero error
DISPLAY ‘3.14 | Sin[1.43]/0 | 100<10’; / 3.14 = TRUE, rest is irrelevant
/ will produce a divide by zero error
DISPLAY ‘Sin[1.43]/0 | 1==1 | 5==25/5 | 100>10’; /Stroboscope is not as /smart as we are

The precedence of operators in Stroboscope is modeled after the ‘C’ programming language. This precedence was very carefully selected so as to minimize the use of parentheses. This is especially evident in the logical operators. Operators and their precedence are summarized in the “Operators” section of the reference (Appendix A).

Economy in the use of parentheses is a convenience. This does not imply that unnecessary parentheses are harmful in any sense. Usually, parentheses make the intention of an expression clear, even if they are not required given the precedence rules. It is usually easier to read code with extra parentheses, than to read code that needs to be studied carefully and with the operator precedence table at hand. There is no penalty in terms of speed or memory consumption in the use of unnecessary parentheses.
Stroboscope provides several unary and binary operators. Stroboscope also provides one ternary operator (i.e., which requires three arguments), which is not classified as a logical operator, but whose first argument is a logical value. This ternary operator is the ‘?:’ (conditional operator). It is the operator with lowest precedence (it is resolved last). The conditional operator is applied in the following form:

\[
\text{LogicalArgument} \ ? \ \text{ResultIfTrue} : \ \text{ResultIfFalse}
\]

where \(\text{LogicalArgument}\) is a logical expression, and \(\text{ResultIfTrue}\) and \(\text{ResultIfFalse}\) are expressions that are not necessarily logical. Stroboscope applies the conditional operator by first evaluating the first argument (\(\text{LogicalArgument}\)). If the value of the first argument is TRUE, Stroboscope evaluates the second argument (\(\text{ResultIfTrue}\)), and uses the result as the result of the conditional operator. If the value of the first argument is FALSE, Stroboscope evaluates the third argument (\(\text{ResultIfFalse}\)), and uses the result as the result of the conditional operator. Stroboscope evaluates either the 2nd or 3rd argument, depending on the value of the 1st argument. The 2nd argument is not evaluated if the first argument is FALSE. The 3rd argument is not evaluated if the first argument is TRUE. The conditional operator is borrowed from the ‘C’ programming language, and is the equivalent of the IIF (immediate if) function in certain dialects of BASIC. The following example illustrates the conditional operator:

```display
DISPLAY '7==1 ? 25 : 50'; /will display the value 50
DISPLAY '28 ? 3 : 1'; /will display the value 3 (28 is TRUE)
DISPLAY '0 ? 3 : 1'; /will display the value 1 (0 is FALSE)
```

/ this will show the value -5, (the parentheses are not redundant)

```display
DISPLAY '(1==2-1 | 3==9/3 | 4==100/25) & 4.11-0.01==4.00 ? 700 : -5;
```

The conditional operator is more convenient than the IIF function used in BASIC and other languages. The fact that the conditional operator has the lowest precedence and that it is left to right associative, allow us to write lengthy nested conditional expressions without parentheses. The following example illustrates this:

```display
/ The following variables represent a discrete probability distribution
/ that models the outcome of rolling a die
/ The Rnd[] function returns a random number between 0 and 1
/ The LastRnd[] function returns the last value returned by Rnd[]
```

Copyright © 1996 Julio C. Martínez
VARIABLE DieOutcome1 ' Rnd[] <= 1/6 ? 1 :
    LastRnd[] <= 2/6 ? 2 :
    LastRnd[] <= 3/6 ? 3 :
    LastRnd[] <= 4/6 ? 4 :
    LastRnd[] <= 5/6 ? 5 : 6';

    / The following variable is the same thing, except that
    / all implicit parentheses are shown

VARIABLE DieOutcome2 ' Rnd[]<=1/6 ? 1 :
    (LastRnd[]<=2/6 ? 2 :
    (LastRnd[]<=3/6 ? 3 :
    (LastRnd[]<=4/6 ? 4 :
    (LastRnd[]<=5/6 ? 5 : 6))))';

    / The following variable is a third example of the same thing, except
    / that we take advantage of associativity in the 2nd argument, as
    / opposed to in the 3rd argument

VARIABLE DieOutcome3 'Rnd[]<=5/6 ?
    LastRnd[]<=4/6 ?
        LastRnd[]<=3/6 ?
            LastRnd[]<=2/6 ?
                LastRnd[]<=1/6 ?
                    1 :
                    2 :
                    3 :
                    4 :
                5 :
            6';

    / The last variable is a fourth example of the same thing, this variable
    / differs from DieOutcome3 in that the implicit parentheses are shown

VARIABLE DieOutcome4 'Rnd[]<=5/6 ?
    (LastRnd[]<=4/6 ?
        (LastRnd[]<=3/6 ?
            (LastRnd[]<=2/6 ?
                (LastRnd[]<=1/6 ?
                    1 :
                    2) :
                3) :
            4) :
        5) :
    6';

Generally speaking, it is clearer to arrange the arguments of the conditional
operator such that the 3rd argument is more complex than the 2nd argument. The
definition of variable DieOutcome1 above should make that evident.
5.6 Specifying a Stopping Condition for a Simulation

So far, the SIMULATE control statement has been used to start running the simulation. When the simulation is run with the SIMULATE statement, it only stops due to a lack of resources or due to a runtime error.

It is often desirable to stop a simulation before a lack of resources condition exists. In fact, many models will run forever without encountering a lack of resources. Stroboscope offers the SIMULATEUNTIL statement to run a simulation until a certain expression becomes TRUE:

Syntax: \texttt{SIMULATEUNTIL Logical\_Terminating\_Condition;}
Example: \texttt{SIMULATEUNTIL Moved\_Soil.\_Cur\_Count \geq \text{Soil\_To\_Move;}}

When Stroboscope executes the SIMULATEUNTIL control statement, it starts running the simulation. Before the beginning of every CIP and every CAP, Stroboscope evaluates the \texttt{Logical\_Terminating\_Condition} expression. If the expression returns TRUE (non-zero), the simulation stops. Otherwise the simulation continues. For example, file ‘haul.str’ (ran earlier in this chapter), could have had the following instead of the SIMULATE statement:

\texttt{/* run simulation until 700 minutes have passed, or 450 hauler-loads of soil have been dumped, whichever occurs first */}
\texttt{SIMULATEUNTIL ‘SimTime>700 | Dump.TotInst>=450’;}

In this case, instead of stopping at simulation time 1244.44 after 1000 hauler-loads of soil have been dumped, the simulation would have stopped earlier as indicated in the comment.

5.7 Source File Organization

Stroboscope executes statements in the order in which it encounters them in a source file.

Control statements have an immediate effect. Their execution takes into consideration the model and its elements as they have been defined up to that point.
The execution of element definition and element attribute statements results in the definition of the model’s elements and their attributes. They do not otherwise imply any immediate action. Therefore, the relative order of the element definition and attribute statements that exist between control statements is not important. The only requirement is that modeling elements be defined before they are used.

Despite the relaxed requirements regarding the order of statements. It is convenient to organize them judiciously. Although largely a matter of programming style, the following organizational scheme has proved convenient:

- At the top of the source file include information regarding the operation being modeled, such as the physical location of the project, the objectives of the simulation study, and the names of files that contain illustrations or drawings of the network for the model.

- Include statements that display a summary of the above information on the output device.

- Define the decision variables for the model. It is convenient to define these variables at the top so that new values can be tried out easily.

- Display in the output device the values of the decision variables being used.

- Define the modeling elements for the model without setting any attributes:
  - Define the resource types for the model.
  - Define the nodes (Queues, Combis, Normals, etc.) of the network.
  - Define the links of the network. Group links for the same types of resources together, and order them in a sequence that resembles the order in which the links will be traversed by resources (inasmuch as possible).
  - Define Variables that may simplify expressions that will later be used to set the attributes of modeling elements.
  - Set the attributes of the modeling elements. If possible, set the attributes in an order that resembles the order in which attributes will be used during simulation.
• Initialize all Queues.

• Run the simulation using the SIMULATE or SIMULATEUNTIL statements.

• Include statements that present the result of the simulation. If both a standard report and custom output are to be displayed, present the custom output first and leave sufficient white-space between the custom output and the standard report.

A complete example

File ‘haul2.str’ shown below has been prepared according to the above guidelines. The code is based on the earth-moving operation of Figure 10. Pertinent data appears in the source file itself. Note that the network is nearly the same as the network in Figure 7 on page 27, except that the Queues and links for soil have been removed. This network models soil implicitly and will never stop due to a lack of resources.

```
*****************************************************************
/* Sample earth-moving operation using haulers and loaders to move
/ soil. The purpose of this study is to determine the number of
/ loaders and haulers that will provide us with the lowest
/ unit cost for the movement of 100,000 cubic meters of soil. 
*******************************************************************/

DISPLAY "Earth-moving operation based on Figure 10";
```

Figure 10 - Classic Earth-Moving Operation
/ Problem decision variables

VARIABLE NumberOfLoaders 3;
VARIABLE NumberOfHaulers 11;

/ Other problem parameters

/ Amount of soil to move
VARIABLE SoilToMove 100000; /cubic meters
VARIABLE HaulerCapacity 12; /1 hauler-load = 12 m3
VARIABLE LoaderCapacity 4; / 1 scoop = 4 m3, 3 scoops per hauler

/ One time equipment move-in cost, per unit
VARIABLE LdrMoveInCst 2250; / $/loader
VARIABLE HlrMoveInCst 1025; / $/hauler

/ Daily Costs
VARIABLE LoaderCst 600.00; $/day
VARIABLE HaulerCst 435.00; $/day
VARIABLE OverheadCst 300.00; $/day

/ Load time parameters -- all in minutes per scoop.
VARIABLE PessLdTm 1.5;
VARIABLE LikelyLdTm 1.1;
VARIABLE OptLdTm 0.7;

/ Haul time, Normally distributed -- in minutes per trip
VARIABLE ExpectedHaulTime 5.5;
VARIABLE HaulTimeVariability 0.22; / coefficient of variation

/ Dump time, fixed -- in minutes per dump
VARIABLE DumpTime 0.50;

/ Return time, Normally distributed -- in minutes per trip
VARIABLE ExpectedReturnTime 3.8;
VARIABLE ReturnTimeVariability 0.1; / coefficient of variation

/ echo to standard output decision variables and model parameters

DISPLAY;
DISPLAY "Number of haulers : " NumberOfHaulers;
DISPLAY "Number of loaders : " NumberOfLoaders;
DISPLAY;
DISPLAY "Amount of soil to move : " SoilToMove " m3";
DISPLAY "Capacity of haulers : " HaulerCapacity " m3"
DISPLAY "Capacity of loaders : " LoaderCapacity " m3"
DISPLAY "Number of scoops per hauler : " HaulerCapacity/LoaderCapacity;
DISPLAY;
DISPLAY "Loader move-in cost" : "LdrMoveInCst $/loader";
DISPLAY "Hauler move-in cost" : "HlrMoveInCst $/hauler";
DISPLAY;
DISPLAY "Daily cost of loaders" : "LoaderCst $/day";
DISPLAY "Daily cost of haulers" : "HaulerCst $/day";
DISPLAY "Daily overhead cost" : "OverheadCst $/day";
DISPLAY;
DISPLAY "Duration of each scoop" : Pert[OptLdTm, LikelyLdTm, PessLdTm] min.;
DISPLAY "Duration of haul" : Normal[ExpectedHaulTime, ExpectedHaulTime*HaulTimeVariability] min.;

DISPLAY "Duration of dump" : "DumpTime min.";
DISPLAY "Duration of return" : Normal[ExpectedReturnTime, ExpectedReturnTime*ReturnTimeVariability] min.;
DISPLAY;
DISPLAY;
/*****************************/

/ define the network

/ Define the resource types
GENTYPE Loader; /for now we will discretize a bulk resource
GENTYPE Hauler; /for now we will discretize a bulk resource

/ Define the Queues
QUEUE LoadersWait Loader;
QUEUE HaulersWait Hauler;

/ Define the Activities
COMBI Load;
NORMAL Haul;
NORMAL Dump;
NORMAL Return;

/ Define the Links

/ Loader Cycle
LINK LD1 LoadersWait Load;
LINK LD2 Load LoadersWait;
/ Hauler Cycle
LINK HL1 HaulersWait Load;
LINK HL2 Load Haul Hauler;
LINK HL3 Haul Dump Hauler;
LINK HL4 Dump Return Hauler;
LINK HL5 Return HaulersWait;

/ define some variables to aid computations
VARIABLE HaulerLoadsRequired SoilToMove/HaulerCapacity;
VARIABLE TotalMoveInCst NumberOfLoaders*LdrMoveInCst +
    NumberOfHaulers*HlrMoveInCst;
VARIABLE TotalDailyCst NumberOfLoaders*LoaderCst +
    NumberOfHaulers*HaulerCst +
    OverheadCst;
VARIABLE DaysSimulated SimTime/60/8; /8 hour days - ignore interruptions
VARIABLE TotalCost TotalMoveInCst+DaysSimulated*TotalDailyCst;
VARIABLE UnitCost TotalCost/SoilToMove;

/ set attributes of modeling elements
/ Sample three times because there are three scoops per hauler.
/ FOR NOW, the 3 terms need to be explicitly typed.
DURATION Load "Pert[OptLdTm,LikelyLdTm,PessLdTm] +
    Pert[OptLdTm,LikelyLdTm,PessLdTm] +
    Pert[OptLdTm,LikelyLdTm,PessLdTm];"
DURATION Haul ExpectedHaulTime*Normal[1,HaulTimeVariability];
DURATION Dump DumpTime;
DURATION Return ExpectedReturnTime*Normal[1,ReturnTimeVariability];

/ Create resources and populate QUEUES
INIT LoadersWait NumberOfLoaders;
INIT HaulersWait NumberOfHaulers;

/ Run the simulation until the desired soil is moved
SIMULATEUNTIL 'HaulerLoadsDumped>=Hauler LoadsRequired';

/ present selected results
The code above may seem lengthy. It contains 77 statements. However, most of the statements are there to create a parameterized model that produces the results of interest directly. The same problem could have been modeled using only 23 statements (15 of which come from the drawing), but it would have been harder to perform experiments on the model. A sample output that results from the processing of `haul2.str` appears below:

Stroboscope Model haul2.str (2006983808)

Earth-moving operation of Figure 10

Number of haulers : 11
Number of loaders : 3

Amount of soil to move : 100000 m³
Capacity of haulers : 12 m³
Capacity of loaders : 4 m³
Number of scoops per hauler : 3

Loader move-in cost : 2250 $/loader
Hauler move-in cost : 1025 $/hauler

Daily cost of loaders : 600 $/day
Daily cost of haulers : 435 $/day
Daily overhead cost : 300 $/day

Duration of each scoop : Pert[0.7,1.1,1.5] min.
Duration of haul : Normal[5.5,1.21] min.
Duration of dump : 0.5 min.
Duration of return : Normal[3.8,0.38] min.
Selected results from experiment

Time required to move soil : 21.478138 days
Fixed cost : 18025 $
Variable cost : 6885 $/day
Total cost of operation : 165901.98 $
Unit cost of operation : 1.6590198 $/m³

Execution Time = 3.969 seconds
Processor Time = 3.828125 seconds
Chapter 6

Controlling the Amounts of Resource Flows

In a construction process, resources are utilized in various amounts. Some operations require or produce more, or less, than one unit of each of the different types of resources involved. Stroboscope provides the modeler with very detailed control of the amounts of resources required, used, consumed, and created by the different tasks in a process. This chapter is an introduction to Stroboscope’s resource drawing and releasing mechanism. The discussion will continue to refer to the earth-moving network of Figure 7 on page 27.

6.1 When Can a Queue Support a Combi?

Stroboscope will instantiate a Combi whenever the Queues that precede the Combi contain enough resources to support it. The relationship between a Combi and the Queues that precede it are established by the links that connect the Queues to the Combi. A link that goes from a Queue to a Combi is called a Draw Link to emphasize the fact that the Combi draws resources from the Queue through that particular link.

In Stroboscope, it is the responsibility of the Draw Link to determine whether the predecessor Queue contains enough resources to support the successor Combi. A Draw Link determines that a Queue can support a Combi through the link’s Enough attribute.

The Enough attribute is an expression that must return TRUE when the preceding Queue can support the task of the Combi, otherwise the expression must return FALSE. The default Enough attribute is an expression that returns the current content of the
predecessor Queue (e.g., ‘PredQueue.CurCount’). For this reason, by default, a Queue can support a Combi if the Queue is not empty. Notice that any positive amount (including a fraction) is sufficient, and that there is no need for the Queue to contain one or more resources.

The ENOUGH statement changes a Draw Link’s Enough attribute:

Syntax:  
ENOUGH DrawLink LogicalEnoughExpression;
Example:  
ENOUGH SL1 ‘SoilToMove.CurCount >= HaulerSize’;

*DrawLink* is the link whose Enough attribute is being changed.  
*LogicalEnoughExpression* is an expression that Stroboscope will evaluate every time it needs to know if the predecessor Queue can support the Combi.

The example used above indicates that *SoilToMove* is able to support *Load* only when the content of the Queue is larger than or equal to *HaulerSize* (a user-defined Variable).

It is very important to remember that a Draw Link’s Enough attribute is a logical expression. An Enough attribute that evaluates to 10,000 is no different than an Enough attribute that evaluates to 0.0001. They are both TRUE, and they both indicate that the predecessor Queue can support the Combi. It is also important to remember that:

- The value returned by the Enough attribute has no implication whatsoever upon the actual amount of resource that will flow through the link.
- For a Combi to start, all of the Queues that precede it must be able to support the Combi. In more precise terms, the Enough attribute of each and all of the links that enter the Combi must evaluate to TRUE.
- It is possible to define more than one link going from the same Queue to the same Combi. The Queue is only able to support the Combi if the Enough attributes of all such links are TRUE.
- The Enough attribute of a link is evaluated as one of the steps that determine if a Combi will or will not be able to start. Thus, when a Draw Link’s Enough attribute is evaluated, the successor Combi attempting to start is not in context.
(i.e., the Combi’s instance variables are not accessible because no instance of the Combi has been created yet).

6.2 Combi Resource Acquisition

Once a Combi can start, Stroboscope creates an instance of the Combi. The instance obtains its resources from the Queues that precede the Combi. A Combi removes resources from the preceding Queues through the Draw Links that enter the Combi.

The process by which a Combi obtains resources from the Queues that precede it is called Drawing. The drawing process consists of one or more (or none) draws through every link that enters a Combi.

6.2.1 Acquiring Resources in One Step

The method by which bulk and discrete resources are drawn from a Queue to a Combi are different. Discrete resources have not been examined in any detail yet. Bulk resources have been used in integer amounts as a substitute.

The Draw Link through which a generic (bulk) resource flows is called a Generic Draw Link. The amount of resource that a Generic Draw Link moves from the predecessor Queue to the successor Combi depends on the link’s DrawAmount attribute. The DrawAmount attribute is an expression that the link evaluates just before it moves resources from the Queue to the Combi. The link attempts to move an amount of resource equal to the result of the expression.

Sometimes the result of the DrawAmount attribute is larger than the current contents of the predecessor Queue. In these cases, it is not possible to remove that amount from the Queue. Stroboscope will remove as much as possible, which is the entire content of the Queue.

The default DrawAmount returns the value 1. It can be changed with the DRAWAMT statement:

Syntax: `DRAWAMT GenDrawLink DrawAmountExpression;`
Example: `DRAWAMT SL1 HaulerSize*Load.Haulers.Count;`
GenDrawLink is the name of Generic Draw Link whose DrawAmount attribute is to be changed. DrawAmountExpression is an expression that the link will evaluate just before (and every time) it draws.

Stroboscope evaluates a Generic Draw Link’s DrawAmount attribute when the successor Combi is starting. Thus, when the DrawAmount attribute is evaluated the Combi is in context (i.e., the Combi’s instance variables are accessible because the Combi instance is being created). The example uses the pre-defined instance variable Load.Haulers.Count, which returns the number of haulers held by the instance in context of the Load Activity (the instance that is starting).

The example implies that when the Combi is acquiring soil through SL1 it has already acquired the haulers from another Queue. Furthermore, it implies that the number of haulers acquired is not necessarily one. For this to be the case, the Combi needs to draw haulers from HaulersWait through HL1, before it draws soil from SoilToMove through SL1. Thus, the exact order in which a Combi draws from the Queues that precede it is important (even though all the draws from preceeding Queues appear to be performed at the same point in simulated time). The default order that Stroboscope uses to successively draw from links depends on the order in which the links are defined in the source file. In the example above, link HL1 must be defined before link SL1. Otherwise, at the time of evaluation of SL1’s DrawAmount, the value of Load.Haulers.Count will be zero (no haulers have been acquired because link HL1 has not been processed yet).

When the order in which links are processed (to draw the resources for a certain Combi) is important, this order can be shown on the network drawing by using superscripts. In the above example, the links that enter Load could be labeled HL1^1, LD1^2, and SL1^3. This would indicate that Load first acquires the hauler, then the loader, and finally the soil.

6.2.2 Acquiring Resources in Several Steps

The example used in the previous section removes all the soil required to fill a hauler at once. In most cases, the bucket of a loader is smaller than the size of a hauler. Thus, it is necessary to use more than one scoop of soil to fill a hauler. Typically, soil
will be moved from a pile to a hauler one scoop at a time until the hauler is full. Many other processes in construction behave in this manner. The drawing mechanism in Stroboscope was designed with this in mind. In order for a Combi to further control the draws through a link, it makes use of the link’s DrawUntil attribute.

A Draw Link’s DrawUntil attribute is a logical expression that Stroboscope evaluates **before** it moves resources from a Queue to a Combi. Drawing stops when the expression returns TRUE (non-zero).

The default DrawUntil attribute is the expression ‘**LinkName**.nDraws’, where 

**LinkName**.nDraws is a pre-defined system-maintained instance variable that applies to Draw Links. This variable returns the number of times resources have been drawn through the link from the preceding Queue to the current instance of the successor Combi. Before a Combi attempts to draw through a link for the first time, **LinkName**.nDraws is zero (FALSE). Thus, the Combi draws resources from the Queue.

The second time a Combi attempts to draw through a link, **LinkName**.nDraws is one (TRUE). Thus, the second draw through this link is abandoned and the Combi goes on to the next link (if any). Thus, the default DrawUntil allows a Combi to draw only once through each of its incoming links.

A Draw Link’s DrawUntil attribute can be changed with the DRA\text{WUNTIL} statement:

**Syntax**: DRA\text{WUNTIL} DrawLink LogicalExpressionThat\text{StopsTheDrawing};

**Example**: DRA\text{WUNTIL} SL1 Load.Soil.Count\geq Load.Haulers.Count\ast HaulerSize;

*DrawLink* is the name of the link whose DrawUntil attribute will be changed. 

*LogicalExpressionThat\text{StopsTheDrawing}* is the expression that Stroboscope will use to determine when to stop drawing from the link.

The following code should illustrate the use of the Enough, DrawAmount, and DrawUntil attributes.

```plaintext
/ 3 m3 buckets, 15 m3 haulers, ---> 5 buckets to fill a hauler
VARIABLE ScoopSize 3;
VARIABLE HaulerSize 15;

/ Load will only start if there is enough soil to fill a hauler
/ (assuming this is what we want to model)
ENOUGH SL1 SoilToMove.CurCount\geq HaulerSize;
```

Copyright © 1996 Julio C. Martínez
/ Keep drawing until the hauler is full
DRAWUNTIL Load.Soil.Count>=HaulerSize;

/ Remove a bucket-full each time
DRAWAMT SL1 ScoopSize;

The above code will only allow the Load Activity to start (as far as sufficiency of soil is concerned) when there is at least enough soil to fill a hauler completely. If the Activity does start, it will remove 3 m$^3$ of soil 5 times from SoilToMove, for a total of 15 m$^3$.

The Load.Soil.Count pre-defined system-maintained instance variable starts out with a value of zero before the first draw through link SL1. As each draw is performed, the variable increases in value by ScoopSize (3). After five draws, Load.Soil.Count will have a value of 15. The DrawUntil attribute will then return TRUE and the drawing through link SL1 will stop.

The Enough, DrawAmount and DrawUntil attributes of a link must be coordinated to achieve the intended behavior (and to avoid undesired secondary effects). Assume that the Enough attribute for link SL1 in the above example was as follows:

/ Load will start when SoilToMove has at least 10 m3
/ (but HaulerSize is 15, which is greater than 10)
ENOUGH SL1 SoilToMove.CurCount>10;

If the DrawUntil attribute of the link remains as before, it could be possible to enter an infinite drawing loop. This would happen if the content of SoilToMove is greater than ten, but less than fifteen. The Enough will indicate that Load can start, but the DrawUntil will never return TRUE because the maximum amount of soil that can be moved is the amount that was in the Queue before drawing started. Stroboscope prevents such infinite loops from occurring because it implicitly appends ‘ | !Queue.CurCount’ to the DrawUntil expression (i.e., draw until the DrawUntil expression is TRUE or the Queue is empty). Thus, the DrawUntil attribute returns TRUE in the cases in which the predecessor Queue is empty, even if the expression used to set the DrawUntil attribute returns FALSE.

Although Stroboscope has some built-in safety checks as the one described above, the DrawUntil attribute must be specified with care so as to not enter an infinite loop.
The logic of the process should be such that the DrawUntil attribute will eventually return TRUE.

### 6.2.3 Detailed Control of Combi Durations

It can be assumed that the Load Combi described above, which performs several draws from SoilToMove, encompasses several sub-tasks. First, the loader and hauler must get ready, or setup, to work together. After they are ready, the loader will start to get soil from SoilToMove, one scoop at a time, until the hauler is full.

The duration of an instance of an Activity is determined by the Duration attribute of the Activity. When several draws are performed through one or more of the links that enter a Combi, the expression that determines the duration of the Combi may need to access information related to the draws.

In the earth-moving example the duration of Load instances may be based on the durations of the draws performed and on the setup time. The time to draw the loader through link LD1 could represent the loader’s setup time; and the time to draw the hauler through link HL1 could represent the hauler’s setup time. Actual loading starts after both the hauler and the loader have been drawn. If the loader and hauler set-up simultaneously, the initial or setup time for the Load Activity is the maximum of the drawing times for the hauler and loader. Each one of the soil draws through link SL1 may also have a duration. Load’s actual duration (not including the hauler and loader setup time) could be set to the sum of the durations of each draw through SL1.

Draw Links have an attribute called DrawDuration that is useful for situations such as the one described above. The DrawDuration attribute is an expression that Stroboscope evaluates to collect information about the durations of draws through a link. The default DrawDuration is an expression that returns zero.

A Draw Link’s DrawDuration attribute can be changed with the DRAWDUR statement:

**Syntax:**

```
DRAWDUR DrawLink DrawDurationExpression;
```

**Example:**

```
DRAWDUR SL1 Pert[0.7,1.1,1.5];
```
**DrawLink** is the name of the link whose DrawDuration attribute is to be changed. **DrawDurationExpression** is an expression that Stroboscope evaluates after each draw, and collects statistics on the result.

The draw duration statistics are accessible through the following pre-defined system-maintained instance variables:

\[ \text{LinkName}.\text{AveDrawDur} \]
\[ \text{LinkName}.\text{MaxDrawDur} \]
\[ \text{LinkName}.\text{MinDrawDur} \]
\[ \text{LinkName}.\text{nDraws} \]
\[ \text{LinkName}.\text{SDDrawDur} \]
\[ \text{LinkName}.\text{SumDrawDur} \]

The following code sample illustrates the use of draw durations:

```
/Time to get a hauler
(other attributes of HL1 are unchanged from default - HL1 draws once)
DRAWDUR HL1 Pert[0.1,0.4,0.5];

/Time to get a loader
(other attributes of LD1 are unchanged from default - LD1 draws once)
DRAWDUR LD1 Pert[0.1,0.3,0.4];

/Time to get one scoop of soil
/ (other attributes of SL1 make SL1 draw several times)
DRAWDUR SL1 Pert[0.7,1.1,1.5];

/Duration of the Load Activity
```

The above sample instructs Stroboscope to calculate the durations of **Load** instances based on the durations of the draws. The setup time is the maximum of the time to get a hauler and the time to get a loader. Soil loading time is the sum of the times for all scoops. The duration of the instance is the setup time plus the soil loading time.

The only purpose of the DrawDuration link attribute is to provide statistics on the values (returned after each draw), through the system-maintained pre-defined instance variables mentioned above. In reality, all of the drawn resources are moved from the Queues to the Activity at the simulation time at which the Activity starts.

Sometimes draw durations depend on the corresponding amount of resource drawn through a link. Stroboscope provides access to this amount through the **LinkName.LastAmtDrawn** pre-defined system-maintained variable. When evaluated, this variable returns the amount of resource that was last moved through the link. In order to
allow the dependence of time on the amount drawn, Stroboscope determines draw
durations immediately after each draw. The following sample illustrates the use of the
variable:

/ Amount of soil in a scoop is not deterministic
DRAWAMT SL1 Max[0.75,Normal[3,0.5]];

/ Time to load a scoop is proportional to the amount of soil drawn
DRAWDUR SL1 Pert[0.7,1.1,1.5]*SL1.LastAmtDrawn/ScoopSize;

6.3 Controlling the Amount of Resources Released

When an Activity instance ends it releases resources to its successor nodes. The
resources are released through links called Release Links. Release Links for bulk
resources are called Generic Release Links. It is the responsibility of the Generic Release
Links to determine the amount of resource released when an Activity instance ends.

Generic Release Links use their ReleaseAmount attribute to determine the amount
of resource that flows through them. The ReleaseAmount attribute is an expression that
the link evaluates when the predecessor ends. The link creates an amount of resource
equal to the result of the expression. The amount is then passed to the successor. The
amount of resource held by the instance that is terminating is not affected.

The default ReleaseAmount attribute is the expression
‘Predecessor.GenType.Count’, where Predecessor is the name of the predecessor
Activity, and GenType is the generic resource type that flows through the link. Thus, by
default, an Activity instance releases through a link as much resource of the type as the
Activity instance acquired.

It is important to remember that the amount of resource released is created by the
link, and that the amount of resource held by the terminating Activity instance is not
altered. For this reason, if two links of the same generic type come out of the same
Activity, each such link will release (create) the same amount (i.e., the value of
‘Predecessor.GenType.Count’ is the same before and after a link releases).
It is also important to note that if the terminating instance did not acquire resources of a certain type, the default amount of resource of the type that will be released is zero.

The ReleaseAmount attribute of a Generic Release Link can be changed with the RELEASEAMT statement:

Syntax:  RELEASEAMT GenReleaseLink ReleaseAmtExpression;
Example: RELEASEAMT SL2 0.95*Load.Soil.Count;

*GenReleaseLink* is the name of the link whose ReleaseAmount attribute will be changed. *ReleaseAmtExpression* is the expression that the link will use to determine how much resource to create and release. This expression can be as complex as required, but must not return a negative number. The expression can return zero, and in such a case a generic resource of amount zero will be created and released (this is not the same as releasing nothing). In the example given, only 95% of the soil acquired by the *Load* Activity is released to the *Haul* Activity.

Note that unless resources are generated or destroyed by Activities, it is not necessary to alter a Generic Release Link’s default ReleaseAmount. The amount of soil drawn through link *SL1* will be released through links *SL2*, *SL3*, and *SL4* by default.

As with Draw Links, the order in which links release may be important. The order Stroboscope uses depends on the order in which the links are defined in the source file. The relative order can be shown in a network drawing using superscripts.

When generic resources are released through links, some information about the amount released may be useful for other parts of a model. Stroboscope provides access to the last amount released by a link and to the last amount received by a Queue. The last amount of resource released through a link is accessible through the pre-defined system-maintained variable *GenReleaseLink.LastAmtReleased*, where *GenReleaseLink* is the name of the link. The last amount of resource to enter a Generic Queue is accessible through the pre-defined system-maintained variable *GenQueue.LastAmtReceived*, where *GenQueue* is the name of the Queue.
6.4 Recap

- A Queue can support a Combi when the Enough attribute of all the links that go from the Queue to the Combi evaluate to TRUE.

- A Combi will not start unless all the Queues that precede the Combi can support it (i.e., the Enough attribute of each of the Draw Links that enter the Combi must evaluate to TRUE).

- A Draw Link’s DrawAmount attribute determines the amount of resource that a Combi attempts to draw through the link.

- A Combi draws through a link until the DrawUntil attribute of the link evaluates to TRUE. This can happen even before the first draw.

- A Draw Link’s DrawDuration attribute is useful for the collection of statistics about draw durations (or any other per-draw values) for a particular instantiation, but it has no direct effect on the duration of Activity instances.

- The order in which a Combi considers its incoming links for drawing may be important. It is determined by the order in which the links are defined in a source file, and is shown in a network drawing using superscripts.

- The amount of resource released through links by terminating Activity instances depends on the Release Link’s ReleaseAmount attribute.

- The amounts of generic resources held by a terminating Activity instances are not altered by releases because Generic Release Links create resources.

- The order in which an Activity considers its outgoing links for releases may be important. It is determined by the order in which the links are defined in a source file, and is shown in a network drawing using superscripts.
Chapter 7
More Detailed Control of Combi Instantiations

Combis are able to start when all the Queues that precede them are capable of supporting the task represented by the Combi. This chapter discusses other issues of importance regarding Combi instantiations. The first issue concerns competition for scarce resources between two or more Combis. The second issue deals with non-resource related conditions that may affect the instantiation of a Combi. The third issue is about optimizations that are built into the Stroboscope simulation engine, and how these optimizations may affect Combi instantiations and model logic.

7.1 Combi Priorities

Sometimes the resources available in a Queue are sufficient to support only a few of the Combis that succeed it. In such a situation, the first Combis to attempt instantiation will be able to start and remove resources from the Queue. Subsequent Combis that attempt instantiation will see that the Queue contains fewer resources, perhaps not enough to support the Combis’ task. Consequently, some Combis will not be able to start because they are in an unfavorable position in the pecking order.

The order in which Combis attempt instantiation may be important for other reasons. Recall that all the Combis that start during a given Combi Instantiation Phase do so at the same point in simulated time. Every startup also involves several steps that include the evaluation of the Enough, DrawAmount, DrawDuration, and DrawUntil attributes of Draw Links; the Duration attribute of Activities; and other attributes of other
elements not yet discussed. The instantiation of each Combi, and even the steps performed during the instantiation of each Combi, change the state of the simulation (e.g., contents of Queues change, current and total instance counts of Combis change, duration and interval statistics about Activities change). A change in the state of the simulation can affect the result of expressions that use system-maintained pre-defined variables that access the changed state. It may be necessary to control the order in which Combis start so that the model’s logic is consistent with the dynamic changes in the state of the simulation.

Stroboscope determines the order in which it attempts to instantiate a Combi by evaluating the Combis’ Priority attribute. Combis attempt instantiation in decreasing order of their Priority. Stroboscope resolves ties in Priority based on the order in which the Combis are defined in the source file. The default Priority of a Combi is the expression ‘0’ (Priorities can be negative). A Combi’s Priority attribute can be changed with the PRIORITY statement:

Syntax: PRIORITY CombiName PriorityExpression;

Example: PRIORITY Load ‘SoilToMove.CurCount>TooMuchSoilInPile ? 10 : 0’;
Example: PRIORITY Load 10;

CombiName is the name of the Combi whose Priority attribute is changed. PriorityExpression is the expression that Stroboscope will use to determine the Combi’s priority. The expression can return any real value (positive, negative, or zero).

The first example sets Load’s priority to 10 (higher than the default value of 0) when the content of SoilToMove is larger than TooMuchSoilInPile, and to zero otherwise. The second example sets Load’s priority to 10 regardless of the content of SoilToMove or of any other condition. In either case, when Load has a high priority (10), it has first chance at attempting instantiation (assuming all other Combis have a priority of zero).

Obviously, the priority of a Combi is irrelevant in a model that contains only one Combi. In models that contain several Combis, only the relative values of priorities matter. The network in Figure 11 below, an extension of the earth-moving example of Figure 7 on page 27, illustrates priorities because two Combis compete for the same resources.
The example in Figure 11 includes haulers of two different sizes. Haulers of different size follow different cycles. The Combis that load haulers with soil, \textit{LoadSmHlr} and \textit{LoadBigHlr}, compete for the resources in \textit{LoadersWait} and \textit{SoilToMove}. When soil or loaders are scarce, and both big and small haulers are waiting to load, the relative priorities of \textit{LoadSmHlr} and \textit{LoadBigHlr} determine which size of hauler will be loaded.

Since Activities for loading, hauling, dumping, and returning of each size of hauler are different, Activities can have durations that depend on hauler size. Similarly, the link attributes that control sufficiency and drawing of soil can and must be specified differently for each hauler size (otherwise there would be no need for separate hauler cycles).

Both the small and the big haulers can be represented with a single generic resource type, \textit{Hauler}. Alternatively, two generic resource types, \textit{BigHauler} and
SmallHauler, can be used to represent haulers of each size. Either will work well in this case.

If PRIORITY statements are not used, then the Combi that appears first in the source code will also be the first to attempt to start (assuming haulers of both sizes are waiting to be loaded). If the following statement is included:

PRIORITY LoadBigHlr 1; / give preference to big haulers

then big haulers will be served first regardless of the order in which LoadBigHlr and LoadSmHlr are defined in the source code.

If the following statement is included:

PRIORITY LoadBigHlr -1; / give least preference to big haulers

then big haulers will be served only after the small haulers have been served.

If the following statements are included:

PRIORITY LoadBigHlr 100; / big haulers have high priority
PRIORITY LoadSmHlr 100; / small haulers also have high priority

then the Combi that appears first in the source file has priority because the priorities of the two Combis are equal.

The examples above set Priority attributes with expressions that always return the same value (1, -1, or 100). In these cases the priority values are constant. Thus, Stroboscope sorts Combis once at the beginning of the simulation. It then remembers their relative order throughout the remainder of the simulation.

Combi priorities can also be defined as follows:

/ Give preference to the longest hauler Queue (the real physical queue)
PRIORITY LoadBigHlr 'BigHlrsWait.CurCount>SmHlrsWait.CurCount ? 1 : -1';

In this case, the LoadBigHlr's priority depends on the state of the simulation (i.e., the contents of BigHlrsWait and SmHlrsWait). If the number of big haulers waiting is greater than the number of small haulers waiting, then big haulers are served. Otherwise, small haulers are served.
When priorities are dynamic, as in the last example, Stroboscope needs to sort the Combis in the model very often. When a model contains more than a few Combis, this sorting can consume a significant amount of computing resources.

In order to achieve a reasonable level of efficiency in these situations, Stroboscope compromises the dynamic behavior of Combi priorities - it sorts Combis only at the beginning of every Combi Instantiation Phase (CIP).

Although this is adequate in most cases, it has implications that may not be initially evident. For example, the above dynamic priority for \textit{LoadBigHlr} depends on the contents of Queues that change every time the Combis in the model start. Recall that a Combi will continue to start until the Queues that support it can no longer provide it with resources (as determined by the Enough attributes of the links that go from the Queues to the Combi). Assume that \textit{LoadBigHlr} has a high priority at the beginning of the CIP and will be processed before \textit{LoadSmHlr}. As instances of \textit{LoadBigHlr} are created, the content of \textit{BigHlrsWait} decreases. The value of \textit{BigHlrsWait.CurCount} may decrease to be smaller than or equal to \textit{SmHlrsWait.CurCount}. If performed, a re-sort of the Combis according to their priorities would change their relative processing order. Stroboscope, however, does not re-sort the Combis until the start of the next CIP. \textit{LoadBigHlr} will continue to attempt instantiations even though conceptually it should have yielded to \textit{LoadSmHlr}.

The above discussion indicates that \textit{LoadBigHlr}’s dynamic priority must be interpreted as “serve as many haulers as possible, of the size whose physical queue is longest before serving starts”.

Due to the dynamic and probabilistic nature of construction processes, situations such as the one described above seldom occur. If, however, it is necessary to model “dynamic priorities to the extreme,” some of the Combis in the model must be substituted with zero-duration Combis followed by Normals. The instantiation of a zero-duration Combi will end the CIP (Combi Instantiation Phase). The instance will be terminated immediately and during its termination Stroboscope will instantiate the succeeding Normal (which now represents the task and has a duration). The CIP will start again, forcing a re-sort of the Combis according to their priorities.
The example of Figure 11 uses generic resources as a substitute for discrete resources. Furthermore, it uses duplicate Activity cycles (one for each type of hauler) in order to distinguish between the different hauler sizes. With Stroboscope, it is not necessary to define more nodes and cycles than really necessary. The characterized resources that will be discussed later provide a much more efficient and elegant way of modeling situations of this kind.

### 7.2 Semaphores

Sometimes tasks are not performed even if all the resources required to perform it are available. During breaks, for example, haulers may not be loaded with soil even if soil, loader, and hauler are available.

This problem can be handled by introducing additional resources in the model. For example, the operator for the loader could be modeled as a resource. During breaks the operator will be captured by a high priority Combi and will not be released until the break is over. Load will not be able to start during breaks because the Queue holding the operator will be empty. While such constructs can be effective, they complicate a model with extra resources that are otherwise uninteresting.

Every Combi has an attribute called Semaphore that is useful for non-resource related control of Combi instantiations. The Semaphore is a logical expression that must return TRUE when it is acceptable to attempt an instantiation of the Combi, and FALSE otherwise. The default Semaphore is the expression ‘1’, which is always TRUE. Thus, by default, Stroboscope will always attempt to instantiate a Combi, and it will succeed whenever the Enough attributes of all incoming links evaluate to TRUE. The Semaphore attribute of a Combi can be changed with the SEMAPHORE statement:

**Syntax:**

```
SEMAPHORE CombiName SemaphoreLogicalExp;
```

**Example:**

```
SEMAPHORE Load ‘WorkingHours & !Weekend’;
```

*CombiName* is the name of the Combi whose Semaphore will be changed.

*SemaphoreLogicalExp* is an expression that Stroboscope will evaluate as the first step in an attempt to instantiate the Combi. If the expression returns FALSE, Stroboscope will abort the attempt to instantiate the Combi. If the expression returns TRUE (the default
Semaphore always returns TRUE), Stroboscope continues the instantiation attempt by evaluating the Enough attribute of all incoming links (thus, the attempt can still fail).

The example above tells Stroboscope to attempt to instantiate Load only if WorkingHours is TRUE and Weekend is FALSE. If WorkingHours is FALSE or Weekend is TRUE, Stroboscope aborts the attempt to instantiate the Combi and goes on to attempt the instantiation of the next Combi, if any.

If a Semaphore returns FALSE, Stroboscope will not automatically attempt to start the Combi as soon as the expression becomes TRUE. The Semaphore will be evaluated only during the Combi Instantiation Phase, and then only after all other Combis with higher Priority have attempted instantiation.

As an example, let us assume that simulation time in the earth-moving simulation is expressed in minutes, that the simulation starts at 8 AM on a Monday (i.e., day 0), and that WorkingHours and Weekend are defined as follows:

\[
\text{VARIABLE CurrentHour Mod[8+SimTime/60,24];}
\]
\[
\text{VARIABLE WorkingHours 'CurrentHour>=8 & CurrentHour<=16';}
\]
\[
\text{VARIABLE DayOfWeek Mod[Int[(8+SimTime/60)/24],7];}
\]
\[
\text{VARIABLE Weekend DayOfWeek>=5;}
\]

Let us also assume that Load’s Semaphore is defined as follows:

\[
\text{SEMAPHORE Load 'WorkingHours & !Weekend';}
\]

During the first 8 hours of the simulation, the Semaphore for Load will return TRUE, and the model will behave just as if it had the default Semaphore (which always returns TRUE). When Load attempts instantiation after these 8 hours have passed, Load’s Semaphore will return FALSE and the attempt will be aborted. In fact, the Semaphore will not return TRUE until the simulation has run for a total of 24 hours and the WorkTime variable has a value of TRUE again. As soon as all the haulers accumulate in HaulersWait, the FEL will be empty. The simulation time will then be a few minutes after 8 hours (i.e., a little after 4 PM), and the simulation will stop.

The simulation stops because the FEL is empty and no Combi can start. Stroboscope is not smart enough (it can’t afford to be) to realize that if it advances the simulation clock to time 24*60, Load’s Semaphore will again be TRUE (at which time it could attempt to instantiate Load and possibly succeed). When the simulation stops
because the FEL is empty and no Combi can start, but the Semaphore of at least one of the Combis in the model is FALSE, the simulation stops due to a “False Semaphore” (and not due to a “lack of resources”).

Obviously, the intention is not to have the simulation stop due to a False Semaphore as soon as the first work day is over. The intended behavior can be achieved by adding some elements to the network that create events in the FEL at the same simulated times at which Semaphores change value. Because work days in this example are 8 hours long, and the 24 hours in a day are a multiple of 8, an event needs to be created every 8th hour of the simulation. This can be achieved by adding the network fragment in Figure 12 to the original network,

![Figure 12 - Eight Hour Clock](image)

and including the following lines of code in the source file (these lines describe Figure 12):

```plaintext
GENTYPE Tick; /just some generic resource to keep things going
QUEUE Ticks Tick; /as soon as a Tick comes in, EightHours will start
COMBI EightHours;
LINK T1 Ticks EightHours;
LINK T2 EightHours Ticks;
DURATION EightHours 8*60;
INIT Ticks 1; /this Tick will circulate constantly
```

The simulation will never end due to a lack of resources or False Semaphore because *EightHours* will be continuously starting and ending. This will create an event every 8 hours. At the simulation time that corresponds to hour 24, an instance of *EightHours* will be ending and another will be starting. At this point, the Semaphore for *Load* will return TRUE, *Load* will be able to start, and the model will behave as intended.

Note that this small network fragment is completely independent of the network that models earth-moving. Its only purpose is to create an event every 8 hours.
The Semaphore is useful for many other situations where Combi instantiations must be controlled for issues not related to resource availability. In fact, the Semaphore allows the definition of Combis with no Queues as predecessors (such a Combi without a Semaphore would start an infinite number of times at the beginning of the simulation).

The following fragment could be used instead of the small network fragment shown above:

```
COMBI EightHours; / no links come in or out of this Combi
SEMAPHORE EightHours !EightHours.CurInst; / one instance at a time!
DURATION EightHours 8*60;
```

Whenever a Combi has no incoming links, it must have a Semaphore that controls the instantiations of the Combi completely.

Both methods shown above create events every 8 hours. This, combined with the definition of the `WorkTime` and `Weekend` variables, makes it very easy to allow a Combi to start only during regular working hours (assuming that shifts start at 8 hour multiples of the beginning of the simulation). If shifts are more irregular, with one or more breaks in the middle of the day, the above approach could be used with some modifications. `EightHours` must be renamed and its duration must be changed to some other convenient duration (it could even be one minute or one second). Furthermore, additional variables may be required to provide relevant information.

Alternatively, a more elaborate network fragment can act as a clock. An annotated example is shown in Figure 13 below (all relevant information can be obtained from the drawing).

Clocks can be as simple or complex as needed for the model. The tradeoff between detailed clock networks with simple variables, and simple clock networks with more complex variables is a matter of style. In any case, clocks are highly reusable and can be plugged into any network easily. There is no need to connect the initial network to the clock.

A clock and Semaphores are heavily used in the Stroboscope model for the operations of a quarry presented in detail in (Martinez and Ioannou 1995). Although the model presented in the paper uses material that will be discussed in chapter 10, the
7.3 The MCS Flag

Stroboscope networks can be as large as required. The language imposes no limitations. When a network contains many Combis the time spent scanning Combis for possible startup can be substantial. Stroboscope has a built-in optimization method that significantly reduces the number of times it needs to scan Combis. It is important to understand how this works to be aware of the possible secondary effects. An understanding of the optimization enables the design of more effective models.

Every Combi in a network has a flag called MCS (“Maybe Can Start”). This flag is either turned on or turned off. When the flag is turned on Stroboscope assumes that it is worthwhile to attempt instantiation of the Combi during the CIP (thus, the Combi has a chance of starting). When the flag is turned off Stroboscope assumes that trying to instantiate the Combi is a waste of time, and therefore does not even attempt to do so.

At the beginning of the simulation every Combi’s MCS flag is turned on. When Stroboscope is attempting to instantiate a Combi, and the Enough attribute of one of the Draw Links that enter the Combi returns FALSE, Stroboscope turns the Combi’s MCS
flag off. The MCS flag for the Combi remains off until one of the Queues that precede the Combi receives new resources.

The rationale behind this optimization is that when a Combi fails instantiation due to a FALSE Enough, it is because the Queues preceding the Combi are not capable of supporting the Combi (i.e., they do not contain enough resources). This situation will not change until some new resources enter the Queues that precede the Combi. When the flag is off, Stroboscope does not attempt to instantiate the Combi because it assumes that the preceding Queues cannot support it. When a Queue receives resources, it turns on the MCS flag of all the Combis that succeed it (maybe those Combis will be able to start now that more resources are available).

When Stroboscope cannot instantiate a Combi because the Combi’s Semaphore returns FALSE, Stroboscope does not turn off the Combi’s MCS Flag. Stroboscope assumes the Combi cannot start due to a non-resource related reason (the preceding Queues may be able to support the Combi). During the next CIP, Stroboscope will attempt to instantiate the Combi again.

Stroboscope allows the definition of Enoughs and Semaphores with any expression. Stroboscope is designed under the assumption that the Enough attributes depend on the resources available in the Queues that precede the Combi, and that Semaphores do not depend on those resources. The MCS Flag optimization method works well in these situations.

There may be occasions in which some Draw Link’s Enough attribute is defined with an expression that is not related to the contents of Queues. It is important to understand MCS optimization in order to understand the consequences of specifying such Enoughs. If such an Enough attribute returns FALSE, Stroboscope will turn off the Combi’s MCS Flag. Stroboscope will not turn on the Combi’s MCS Flag until one of the Queues that precede the Combi receives new resources. In the meantime, Stroboscope will not try to instantiate the Combi during the CIPs (even though the Enough that previously returned FALSE could now return TRUE).

Another interesting point is that when a Combi’s Semaphore is defined with an expression that depends on the resources available to the Combi, simulation speed may be slower. Stroboscope will continue to attempt instantiation of the Combi, even if the
resources available to start it are the same as before. This in fact turns off MCS optimization, but is safe from a logical standpoint.

7.4 Multiple CIP Passes

Assume that a Combi cannot start because its Semaphore returns FALSE. Other Combis with lower Priorities then start and change the state of the simulation in such a way that the Semaphore of the first Combi now returns TRUE. In this case, it is wise to attempt to start the first Combi again during the current CIP. For this reason, when Stroboscope finishes processing the list of Combis, which is sorted in descending priority order, it starts processing the list of Combis again without advancing the clock (i.e., Stroboscope does a second pass). Stroboscope continues to do Combi instantiation passes until none of the Combis can be instantiated (i.e., until there is no change in the state of the simulation that can make a previously FALSE Semaphore become TRUE).

Stroboscope does not re-sort the list of Combis between passes. The list is only re-sorted before the first pass. The order in which instantiations are attempted remains the same between passes.
Chapter 8

General Programming & Statistical Objects

In Stroboscope, general programming and statistical objects are used to maintain our own statistics and variables, as well as to produce formatted output. The system defines the classes of objects available, as well as a few specific objects. Most objects are user-defined from the object classes provided by Stroboscope. This chapter introduces general programming object classes and the mechanism with which objects of these classes are used in Stroboscope.

8.1 Class Categories

The general programming classes in Stroboscope fall into three categories: “statistics collection” classes, “value storage” classes, and “output file” classes.

Statistics Collection Classes

Stroboscope automatically collects various statistics about modeling elements. For example, Stroboscope keeps statistics on the content of Queues or the duration of Activities. Some of these statistics are included in the standard output produced by the REPORT statement.

It is often necessary to obtain statistics beyond those automatically collected by the system. For example, the network for a simulation model might include several subsystems (each made up of several nodes), and statistics may be required about the number of resources in each subsystem.
Stroboscope provides 4 classes of objects collectively called “statistics collectors” whose purpose is to collect data and provide statistics on that data. Due to Stroboscope’s programmability, there is virtually no limit as to the data that collectors can accumulate. The classes in this category allow us to collect statistics on a set of discrete samples, as well as continuous statistics on values that are associated with user-specified weights or that are weighted by simulation time.

**Value Storage Classes**

Stroboscope Variables (defined with the VARIABLE statement) are like locked formulas in a spreadsheet (the formulas that define the cells cannot be changed). References to these Variables are always resolved by a reevaluation of the corresponding formula. As a result, Variables cannot be used as to store numbers whose values may be determined or may change at simulation runtime. A Variable can be used as a value storage location if its formula is simply a constant. An example of this is defining the Variable PI with the formula ‘3.1415926’ (notice that strictly speaking this is a formula and not a number). Even in this case, however, the value of the Variable PI cannot be changed from 3.1415926 and replaced with another value.

Often, however, it is necessary to have a variable that retains its value until another value is explicitly assigned to it.

This is accomplished by Stroboscope’s “value storage” classes. As their name implies, the purpose of objects of these classes is to act as storage locations whose content is controlled by the simulation model. Objects of the value storage classes can represent a single variable, one-dimensional and two-dimensional arrays.

**Output Files**

By default, Stroboscope’s output is provided to the user in what is called the standard output device. The standard output device can be a window on the screen, the console, or a disk file, depending on the environment in which the simulation engine is running. For example, the REPORT statement directs its output, by default, to the standard output device. The standard output device can also receive simple unformatted output through the DISPLAY statement.
Some situations require that output be directed to additional devices such as disk files. Other situations require that output be formatted and customized beyond the capabilities of the DISPLAY statement.

Stroboscope provides classes of objects called Output Files that represent output devices and that can format output according to very detailed specifications.

A detailed description of the general programming object classes appears in section 8.4 on page 118. A summary appears in the “action targets” section of the reference on page 427.

### 8.2 Actions

Objects do their job when “arguments” are applied to them. The number and type of arguments that can be applied to an object are determined by the object’s class. The response of the object to the arguments applied to it is also determined by the object’s class.

An “action” is the application of arguments to an object. The arguments being applied are called “action arguments”. The object to which the arguments are applied is called the “action target”.

One of the value storage classes is the `SaveValue` class. A `SaveValue` expects only one argument, which must be an expression. The argument cannot be a string nor a Stroboscope keyword nor the name of a modeling element that cannot be interpreted as a value. A `SaveValue` responds to the application of an argument by evaluating the expression and storing the resulting number (thus replacing the previously stored value). The application of an argument to a `SaveValue` results in an “assignment” of a number to the `SaveValue`.

In general, the response of a Stroboscope object depends entirely on the number and type of arguments applied. This response is fixed and determined by the class of the object. Because there is a one to one correspondence between the combination of arguments applied to an object, and the object’s response to the arguments, it is not necessary to specify how an object should respond. The response is determined by the number and type of arguments.
In the context of SaveValues, for example, it is not necessary to indicate that the argument (a number) will be “assigned” to the SaveValue. There is no alternate object response. The only thing that a SaveValue knows how to do when it receives an expression, is to assign itself the result of evaluating the expression.

Given a different combination of type and number of arguments, an object could respond differently. Most of the classes of objects currently offered by Stroboscope take only one combination. The classes of objects that take more than one combination are Arrays (one of the value storage classes) and the output files.

8.3 Timing of Actions

Actions, when performed in the appropriate sequence, resemble the processing done by a conventional programming language (where every line of a program can represent an action).

In a conventional programming language, the sequence in which lines of code are executed is determined by the flow control structures of the language (e.g., do loop, while loop, procedure call, goto).

In Stroboscope, the world of actions is divided in two:

1. Those actions that are performed during the execution of “action control statements”. These are control statements whose purpose is to perform an action as soon as they are encountered during the processing of the simulation input file. The execution of such a statement, for example, could print a “banner” with the model title before the simulation starts.

2. Those actions that are performed while the simulation is running (i.e., during the execution of a SIMULATE or SIMULATEUNTIL statement). These actions are performed at specific points within the logic of a simulation run called “action events”.

These two types of actions are described below.
8.3.1 Action Control Statements

It is often desired to perform actions that display formatted input information or that do preliminary calculations before the simulation runs. When the simulation finishes it may be necessary to perform actions that do further calculations and that display formatted results. These actions are performed either before or after the simulation runs, but not while it is running (i.e., not during the execution of a SIMULATE or SIMULATEUNTIL statement). Stroboscope performs these actions when it executes “action control statements”.

Action control statements are similar to statements in a programming language such as BASIC. Their purpose is to perform an immediate action on an object. The action is performed as soon as the statement is executed. Three action control statements are available in Stroboscope: ASSIGN, COLLECT, and PRINT. The syntax for these statements is similar, as shown below:

ASSIGN ActionTarget [PRECOND LogicalExp] ActionArguments;
COLLECT ActionTarget [PRECOND LogicalExp] ActionArguments;
PRINT ActionTarget [PRECOND LogicalExp] ActionArguments;

*ActionTarget* is the object to which the action arguments are applied. The number and type of the action arguments, *ActionArguments*, is variable and depends on the class of *ActionTarget*. PRECOND is an optional Stroboscope keyword that must be typed as is. If PRECOND is used, it is followed by the logical expression *LogicalExp*.

When Stroboscope is processing the input source file and executes an action control statement that does not include PRECOND, it applies *ActionArguments* to *ActionTarget*. If the statement includes PRECOND, Stroboscope first evaluates *LogicalExp*. If the result is TRUE, Stroboscope applies *ActionArguments* to *ActionTarget*, otherwise Stroboscope does nothing.

The outcome of the execution of an action control statement depends entirely on *ActionTarget*, *LogicalExp*, and *ActionArguments*. This means that the specific action control statement that is used (e.g., ASSIGN, COLLECT, or PRINT) to carry out the particular action is irrelevant. Thus, the statements ASSIGN, COLLECT, and PRINT are interchangeable.
Despite the fact that ASSIGN, COLLECT, and PRINT are synonyms, it is convenient to use the one that best indicates the action being specified. This practice makes simulation input files easier to understand. Thus, for example, the action control statement that prints formatted text to an output file should logically be the PRINT statement; the action control statement that assigns a value to an element of an array should be the ASSIGN statement; and the action control statement that sends a new data point to a statistics collector should be the COLLECT statement.

The Stroboscope keywords ASSIGN, COLLECT, and PRINT are also called “action verbs”. Thus, the syntax for action control statements can be generalized as follows:

Syntax: \[ \text{ActionVerb ActionTarget [PRECOND LogicalExp] [ActionArguments...]}; \]

Example: ASSIGN SimEndTimeInHours SimTime/60;
Example: PRINT StdOutput PRECOND SimEndTimeInHours>200 "Long simulation, %.2f hours!" SimEndTimeInHours;

For the two examples above to make sense, they should be placed sequentially in a simulation source file after a SIMULATE or SIMULATEUNTIL statement (otherwise SimTime would be zero). The first example performs an action that assigns the current simulation time (the value of the simulation clock at the time of execution of the ASSIGN statement) in hours to the \( \text{SimEndTimeInHours} \) SaveValue. The second statement prints text, such as “Long simulation, 243.42 hours!”, to the standard output device. The text is only printed if the simulated time was longer than 200 hours.

The action target in the first example is the \( \text{SimEndTimeInHours} \) SaveValue (which must have been defined previously). The action argument applied to \( \text{SimEndTimeInHours} \) is the expression ‘SimTime/60’. Thus, once this statement is executed, the new value of \( \text{SimEndTimeInHours} \) will be the current simulation time in hours (assuming that the time unit for the simulation is a minute).

The action target in the second example is StdOutput. StdOutput is a pre-defined object of the OutFile class. It represents the standard output device (i.e., it can be a file, a window, or the console, depending on the environment in which the Stroboscope engine is running). The OutFile class will be described in detail later. OutFile action-targets take
a variable number of arguments. The first argument is a “format string” that may contain embedded “format specifications”. The number of remaining arguments must match the number of format specifications embedded in the format string. The remaining arguments, if any, are expressions whose results replace the format specification(s) embedded in the format string.

The first step in processing the second example is to evaluate ‘CurTmInHours>200’. If the result is FALSE nothing happens. If the result is TRUE, Stroboscope applies the two action arguments (“Long simulation, %.2f hours!” and ‘CurTmInHours’) to the action target (StdOutput). Within the first argument, the characters “%.2f” are a format specification that will be replaced by a numerical value using 2 decimal places. The number to use is given by the result of the second argument. The resulting string is printed to the standard output device. If, for example, the value of CurTmInHours is 319.32765, Stroboscope will print to the standard output device the string “Long simulation, 319.33 hours!”

Note that the examples use “ASSIGN” and “PRINT” as the action verbs. Thus, they provide a clue that some value will be assigned and that something will be printed. Other action verbs could have been used without altering Stroboscope’s execution of the statements, but they would not have been as clear to someone trying to follow the code.

### 8.3.2 Action Events

Actions must also be performed while the simulation is running (i.e., during the execution of a SIMULATE or SIMULATEUNTIL statement). Such actions, for example, could collect supplemental statistics, print text to trace files for use by an animation program, or selectively update the value of SaveValues.

With respect to timing, actions may assign values, collect statistics, or print formatted output every time a specific Activity starts. The “start of a specific Activity” is a distinct “point” within the logic of a simulation that must be defined precisely. There are many possible interpretations, such as the “point before the Activity acquires resources”, “after resources are acquired but before the Activity determines its duration”, or “after it has determined its duration”.
The number of distinct points within the logic of the simulation engine at which actions could be taken is indeed very large. Stroboscope has been designed with a set of such points that are conveniently selected and precisely defined. These predefined points in the simulation logic are called “action events”. Stroboscope can perform specific actions at the occurrence of any one of these events.

In order to be precise, action events must be tied to specific modeling elements. Thus, for example, it is not sufficient to specify that an action should be performed at the start of “an” Activity. It is necessary to specify which Activity.

The following table lists some of Stroboscope’s action events (some action events are not listed because they apply to aspects of the simulation logic that have not been described yet). In general, action events have been chosen so that they cover the entire spectrum of “points” within the simulation logic where actions may need to be performed. The list below presents action events in the order in which they would occur from the perspective of a Combi.

**Table 7 - Action Events**

<table>
<thead>
<tr>
<th>Action Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEFOREDRAWS <strong>CombiName</strong></td>
<td>The event occurs after the <strong>CombiName</strong> Combi is known to be able to start, but before the Combi draws through its incoming links.</td>
</tr>
<tr>
<td>ONDRAW <strong>DrawLinkName</strong></td>
<td>The event occurs when a resource is moved from a Queue to a Combi through link <strong>DrawLinkName</strong>. When this event occurs, the resource has already left the Queue but has not yet arrived at the Combi. The Draw Amount and Draw Duration have already been determined.</td>
</tr>
<tr>
<td>ONSTART <strong>ActivityName</strong></td>
<td>The event occurs just after an instance of the <strong>ActivityName</strong> Activity is created. All the resources that will be part of this instance have already been received and the duration has been determined.</td>
</tr>
<tr>
<td>BEFOREEND <strong>ActivityName</strong></td>
<td>The event occurs just before an instance of the <strong>ActivityName</strong> Activity is created. All the resources that will be part of this instance have already been received and the duration has been determined.</td>
</tr>
<tr>
<td>Action Event</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>ActivityName</strong></td>
<td>Activity is terminated, before the instance releases resources through its outgoing links.</td>
</tr>
<tr>
<td><strong>ONRELEASE RelLinkName</strong></td>
<td>The event occurs when a resource is released from an Activity through link RelLinkName. If this link is for generic resources, the resource has already been created by the link but has not yet arrived at the successor. (If this link is for characterized resources, the characterized resource has already left the Activity but has not yet arrived at the successor.)</td>
</tr>
<tr>
<td><strong>ONENTRY QueueName</strong></td>
<td>The event occurs when a resource enters the Queue QueueName. The resource that entered the Queue is already part of the Queue.</td>
</tr>
<tr>
<td><strong>ONEND ActivityName</strong></td>
<td>The event occurs when an instance of the ActivityName Activity is terminated, after the instance releases resources through its outgoing links, but before it destroys any unreleased resources.</td>
</tr>
</tbody>
</table>

### 8.3.3 Action Definition Statements

Action control statements, like all control statements, are performed when the statement is *executed*. In contrast, actions tied to action events are not specified via control statements. They are defined with “action definition statements”. Thus, **these actions are not performed when the statements that define them are executed**. Instead, Stroboscope takes note of the actions and performs the actions every time the corresponding action event occurs during simulation.

The general syntax for action definition statements is:

**Syntax:** `ActionEvent [ActionVerb] ActionTarget [PRECOND LogicalExp] [ActionArguments ...];`
Example: ONSTART Load PRINT StdTrace
   “Instance %.0f of Load had a duration of %.2f”
   Load.Instance Load.Duration;

Example: ONSTART Load StdTrace PRECOND Load.Duration>15
   “Instance %.0f of Load had a duration of %.2f”
   Load.Instance Load.Duration;

**ActionEvent** is any of the action events listed in Table 7 or other action events that will be introduced in later chapters. Action events are composed of two words. The first word is the generic name of the action event. The second word is the specific modeling element to which the action event is tied. **ActionVerb** is optional and, if used, must be one of: ASSIGN, COLLECT, or PRINT. The remaining arguments are the same as for action control statements described in section 8.3.1.

In both examples, the action event is “ONSTART Load”. The first example uses the optional argument PRINT (i.e., an **ActionVerb**) to emphasize that the purpose of the action is to print. In both cases the action target is StdTrace, an object pre-defined by Stroboscope that belongs to the OutFile class. The second example uses the PRECOND keyword to indicate that the action will not be performed every time that the Load Activity starts. It will be performed only when the duration of the instance that just started is larger than 15. In both examples, the action prints to the standard trace device a string such as “Instance 217 of Load had a duration of 23.44”.

It is possible to perform many actions at the same action event. Stroboscope performs these actions in the order in which the statements that define the actions were executed while processing the simulation input file.

### 8.4 Classes of Action Targets

The following classes of action targets are available in Stroboscope:

- **Value Storage Classes**:
  - Arrays
  - **SaveValues**
  - Time-Weighted Collectors
• Statistics Collectors:
  • Collectors
  • Moving Average Collectors
  • Time-Weighted Collectors
  • Weighted Collectors

• Output Files:
  • OutFiles
  • AppFiles

These classes are discussed in detail in the following subsections.

In the discussion of the several classes of action targets, “EVENT” can be an action control statement, or an action definition statement (optionally followed by an action verb). Thus, in the case of an action control statement, EVENT can be “ASSIGN”, “COLLECT”, or “PRINT”. In the case of action definition statements, any of the following examples can take the place of EVENT: “ONEND Load”, “ONEND Load COLLECT”, “ONEND Load PRINT”, “ONDRAW PL1”, etc.

8.4.1 SaveValues

A SaveValue in Stroboscope is the same as a variable in a conventional programming language. Once a SaveValue is defined, values can be stored in it and referenced in expressions. SaveValues are defined with the SAVEVALUE statement:

Syntax:  SAVEVALUE SaveValueName InitialValueExp;
Example: SAVEVALUE BlastingIsAnnounced 1; / initially TRUE

When this statement is executed while Stroboscope is processing the simulation input file, Stroboscope creates a SaveValue and gives it the name SaveValueName. The initial value of SaveValueName is determined by InitialValueExp. Contrary to Variables (i.e., Stroboscope Variables, defined with the VARIABLE statement), InitialValueExp is never evaluated again. Thus, a SaveValue is not associated with any particular
expression. The value of a SaveValue does not change unless the SaveValue is the target of an action.

The example above defines a SaveValue called `BlastingIsAnnounced` and sets its initial value to 1. `BlastingIsAnnounced` will retain its value of 1 until it is the target of an action that gives it a new value.

SaveValues can be used in expressions as if they were numbers. The following example illustrates this:

```plaintext
/ The ClearLoader Combi cannot start if BlastingIsAnnounced is FALSE
SEMAPHORE ClearLoader BlastingIsAnnounced;
```

Actions on SaveValues take one argument, an expression that is evaluated to determine the new value of the SaveValue:

```plaintext
Syntax: EVENT SaveValueName [PRECOND LogicalExp] NewValueExp;
Example: ASSIGN ReplicationsToGo ReplicationsToGo-1;
Example: ONSTART AnnounceBlast ASSIGN BlastingIsAnnounced 1; /*TRUE
Example: ONEND Inspect BlastingIsAnnounced 0; /*FALSE
```

`SaveValueName` is the name of the SaveValue. `NewValueExp` is an expression that Stroboscope evaluates to determine the new value of the SaveValue. Upon execution of the first example, Stroboscope will decrement the value of the `ReplicationsToGo` SaveValue. The second example instructs Stroboscope to assign the value 1 to `BlastingIsAnnounced` every time the `AnnounceBlast` Activity starts. The third example instructs Stroboscope to assign the value 0 to `BlastingIsAnnounced` every time the `Inspect` Activity ends.

### 8.4.2 Arrays

Arrays are indexed (zero-based) collections of SaveValues. The elements of one-dimensional arrays are accessed with one index, and the elements of two-dimensional arrays with two indexes. Arrays are defined with the ARRAY statement:

```plaintext
Syntax: ARRAY OneDArrayName Size [{InitExp1 InitExp2 ...}];
Example: ARRAY AccidentsAtStation 4;
Example: ARRAY StationHeights 4 {3.28 4.16 1.17 8.03};
```
Syntax: \[ \text{ARRAY TwoDArrayName Rows Columns \{InitExp1 InitExp2 \ldots \}}; \]
Example: \[ \text{ARRAY RoomStage 3 5;} \]
Example: \[ \text{ARRAY RoomAreas 3 5} \]
\begin{verbatim}
{  4.29  3.17  44.21  12.00  14.50 \\
  7.24  8.14  9.54  90.47  7.12 \\
  34.17 12.00  20.72  4.50  13.50};
\end{verbatim}

The first form is used to define one-dimensional arrays. The second form is used to define two-dimensional arrays. \textit{OneDArrayName} and \textit{TwoDArrayName} are the respective names of the arrays. \textit{Size} is the number of elements in the one-dimensional array. \textit{Rows} and \textit{Cols} are the number of rows and columns in the two-dimensional array. \textit{Size}, \textit{Rows}, and \textit{Cols} are expressions that Stroboscope evaluates and truncates to integers when an array is defined. They are not used later.

In both cases, the square brackets in the syntax indicate that the arguments enclosed in the brackets are optional. The brackets themselves are not part of the statement. The curly braces are required if the option within the square brackets is used. The optional arguments indicate the initial values of the elements in the array. When the optional arguments are not used, all the elements in the array are initialized with the value 0. When the arguments are used, Stroboscope evaluates the expressions to determine the initial value of each element — there must be an expression for every element in the array. The initial values for two-dimensional Arrays must be specified in row-order. There is no special delimiter between rows.

The first one-dimensional example defines the \textit{AccidentsAtStation} Array. It contains 4 elements, all of which are initialized to 0.

The second one-dimensional example defines the \textit{StationHeights} Array. It contains 4 elements. The first element is initialized to the value 3.28; the second element is initialized to the value 4.16; etc.

The first two-dimensional example defines the \textit{RoomStage} Array. It contains 15 elements arranged in 3 rows of 5 columns. All the elements are initialized to 0.

The second two-dimensional example defines the \textit{RoomAreas} Array. It also contains 15 elements arranged in 3 rows of 5 columns. The first element (row 0, column 0), is initialized to the value 4.29. The element at row 2 (third row), column 3 (fourth column), is initialized to the value 4.50, etc.
When an Array is defined, Stroboscope creates a function with the same name as the Array. The function allows access to the values of the elements in the Array. The number of arguments that the function takes depends on the number of dimensions in the array.

The function is zero-based. Thus, ArrayName[0] accesses the first element of array ArrayName, and ArrayName[Size-1] accesses the last element of ArrayName (assuming the size of the array is Size).

For example, the following statements display values from the Arrays defined above:

DISPLAY StationHeights[2]; / will display 1.17
DISPLAY RoomAreas[2,3]; /will display 4.50

Actions on Arrays take arguments in different formats depending on the number of dimensions, and depending on whether the action targets a single element within the Array or all the elements in the Array.

Statements in the following format specify actions that change the value of a single element in a one-dimensional Array:

Syntax: EVENT OneDArrayName [PRECOND LogicalExp] IndexExp NewValExp;
Example: ASSIGN StationHeights 3 Int[Rnd[]*100];
Example: ONSTART InjuryAtStation ASSIGN AccidentsAtStation CurStation AccidentsAtStation[CurStation]+1;

OneDArrayName is the name of the Array. IndexExp is an expression that Stroboscope evaluates and truncates to determine the index of the element in OneDArrayName that will take a new value. IndexExp must return a non-negative value smaller than the size of the array, otherwise Stroboscope issues an error. NewValExp is an expression that Stroboscope will evaluate to determine the new value of the Array element.

The first example assigns a random integer between 0 and 99 to the fourth element in the StationHeights array, as soon as the ASSIGN statement is executed.

The second example instructs Stroboscope to increment the element indexed by CurStation (a variable defined elsewhere) in the AccidentsAtStation Array, every time that the InjuryAtStation Activity starts.
To specify actions that change the value of a single element in a two-dimensional array, statements with the following format are used:

Syntax: EVENT TwoDArrayName [PRECOND LogicalExp]
RowExp ColExp NewValueExp;
Example: ASSIGN RoomAreas 2 3 38.25;
Example: ONEND PaintWalls ASSIGN RoomStage CurFloor CurRoom Painted;

TwoDArrayName is the name of the Array. RowExp and ColExp are expressions that Stroboscope evaluates and truncates to determine the row and column of the element in TwoDArrayName that will take a new value. NewValExp is an expression that Stroboscope will evaluate to determine the new value of the Array element.

The first example assigns the value 38.25 to the element at the third row and fourth column of the RoomAreas Array, as soon as the ASSIGN statement is executed.

The second example instructs Stroboscope to assign the value Painted to the element at row CurFloor and column CurRoom of the RoomStage Array, every time that the PaintWalls Activity ends.

Statements in the following format specify actions that change the value of an entire Array:

Syntax: EVENT ArrayName [PRECOND LogicalExp]
{Expression ...};
Example: ASSIGN AccidentsAtStation {0 0 0 0};/zero out all elems
Example: ONEND InspectPaint1stFloor ASSIGN RoomStage
{ Inspected Inspected Inspected Inspected Inspected
Painted Painted Painted Painted Painted
Erected Erected Erected Erected Erected};

ArrayName is the name of the Array. The curly braces must enclose one Expression for each element in ArrayName. The Expressions are evaluated to determine the values that will be assigned to the elements in the Array.

The first example sets all elements in the AccidentsAtStation Array to zero.

The second example sets the value of the elements of the RoomStages Array to the Inspected, Painted, or Erected values. This happens every time the InspectPaint1stFloor Activity ends.
8.4.3 Collectors

Collectors are action targets that keep statistics about the numbers they receive. A Collector is defined with the COLLECTOR statement:

Syntax:  
COLLECTOR CollectorName;

Example: COLLECTOR BlastingTimeStats;

CollectorName is the name of the Collector. The example defines a Collector named BlastingTimeStats.

Conceptually, Collectors are similar to dedicated statistical calculators. The following pre-defined system-maintained variables access statistics about the values received by a Collector:

- CollectorName.AveVal (average of the values received by CollectorName)
- CollectorName.MaxVal (maximum of the values received by CollectorName)
- CollectorName.MinVal (minimum of the values received by CollectorName)
- CollectorName.nSamples (number of values received by CollectorName)
- CollectorName.SDVal (standard dev. of the values received by CollectorName)
- CollectorName.SumVal (the sum of the values received by CollectorName)

Collectors receive data that are sent to them by actions. These actions take one argument:

Syntax:  
EVENT CollectorName [PRECOND LogicalExp] SampleExp;

Example: COLLECT BlastingTimeStats 23.22;
Example: ONRELEASE SC7 COLLECT BlastingTimeStats SimTime-AnnounceBlast.LastStart;

CollectorName is the name of the Collector. SampleExp is an expression that Stroboscope evaluates to determine the value that the Collector receives. Collectors treat the values they receive as statistical samples.

The first example instructs Stroboscope to send the value 23.22 to the BlastingTimeStats Collector. This happens as soon as Stroboscope executes the COLLECT control statement.
The second example instructs Stroboscope to send a value equal to the time elapsed since the last start of the AnnounceBlast Activity, to the BlastingTimeStats Collector. This will happen every time a resource is released through link SC7.

The following code uses a Collector to obtain statistics about some numbers applied to the Collector with the COLLECT control statement:

```
/ define a Collector
COLLECTOR aCollector;

/ apply some values to the Collector
COLLECT aCollector 18.5;
COLLECT aCollector 14;
COLLECT aCollector 10;
COLLECT aCollector 12/2.75;
COLLECT aCollector Log[100]^3;

/ display information about the collected values
DISPLAY "Number of Samples : " aCollector.nSamples;
DISPLAY "Average Value     : " aCollector.AveVal;
DISPLAY "Standard Deviation: " aCollector.SDVal;
DISPLAY "Maximum Value     : " aCollector.MaxVal;
DISPLAY "Minimum Value     : " aCollector.MinVal;
DISPLAY "Sum of the Values : " aCollector.SumVal;
```

Running a source file with the above code produces the following output:

```
Number of Samples : 5
Average Value     : 10.972727
Standard Deviation: 5.4609432
Maximum Value     : 18.5
Minimum Value     : 4.3636364
Sum of the Values : 54.863636
```

Notice that this example does not run a simulation.

### 8.4.4 Moving Average Collectors

A Moving Average Collector is a specialized version of the standard Collector described in the previous section. The difference is that a Moving Average Collector computes statistics by considering, at most, the last “n” values it has received. Here “n” is the “memory” of the Collector. A Moving Average Collector is defined with the MVAVGCOLLECTOR statement:

```
Syntax: MVAVGCOLLECTOR CollectorName MaxSamplesExp;
Example: MVAVGCOLLECTOR BlastingTimeStats 50;
```
CollectorName is the name of the Collector. MaxSamplesExp is an expression that Stroboscope evaluates and truncates to determine the maximum number of samples to consider when computing statistics. If the number of values applied to a Moving Average Collector is less than its maximum number of samples, it is no different from a standard Collector. When more than the maximum number of samples are applied to a Moving Average Collector, the Collector only considers the last “n” (where “n” is the maximum number of samples) values it has received and discards the rest.

Actions are performed on Moving Average Collectors in the same way they are performed on standard Collectors. In addition, Stroboscope offers the same pre-defined system-maintained variables about Moving Average Collectors, as it does on standard Collectors. Note that the maximum value that the MvAvgCollector.nSamples variable can return is the maximum number of samples for the Collector. The following sample code differs from the code in the previous section only in the definition of the Collector:

```
/ define a Collector
MVAVGCOLLECTOR aCollector 3; / three samples maximum

/ apply some values to the Collector
COLLECT aCollector 18.5;
COLLECT aCollector 14;
COLLECT aCollector 10;
COLLECT aCollector 12/2.75;
COLLECT aCollector Log[100]^3;

/ display information about the collected values
DISPLAY "Number of Samples  : " aCollector.nSamples;
DISPLAY "Average Value      : " aCollector.AveVal;
DISPLAY "Standard Deviation : " aCollector.SDVal;
DISPLAY "Maximum Value      : " aCollector.MaxVal;
DISPLAY "Minimum Value      : " aCollector.MinVal;
DISPLAY "Sum of the Values  : " aCollector.SumVal;
```

Running a source file with the above code produces the following output:

```
Number of Samples  : 3
Average Value      : 7.4545455
Standard Deviation : 2.857497
Maximum Value      : 10
Minimum Value      : 4.3636364
Sum of the Values  : 22.363636
```

An examination of the above output indicates that the first two values applied to the Collector, 18.5 and 14, are not included in the statistics.
Moving Average Collectors are useful for determining whether a simulation has reached a steady state condition (Law and Kelton 1991). They consume more memory than standard Collectors because they must keep a list of all the values to be used for statistics. The maximum number of samples in a Moving Average Collector determines the size of the list. The largest value that can be specified as the Maximum Number of Samples depends on the memory available in the computer - Stroboscope imposes no limits.

### 8.4.5 Weighted Collectors

Standard Collectors and Moving Average Collectors collect statistics on a discrete number of samples (i.e., the values received by the Collector). Each sample has equal weight and the standard deviation obtained from these collectors is the sample standard deviation.

The following is a graph of the data used in the example for the standard Collector:

![Graph example](image)

Weighted Collectors are not based on discrete samples as described above. They are based on a continuous sampling process similar to a cardiogram. Continuous statistics are analogous to the area under a graph. The x axes are the weights and the y axes are the values. The average value is the average height of the graph (i.e., the area of the graph divided by its width). The standard deviation gives an indication of graph’s height variability. The maximum value is the highest peak in the graph. The minimum is the lowest. The width of the graph is the total weight. The same values as before, but using continuous statistics and giving each value a weight of one, produces the following graph:

![Continuous graph example](image)
In this case the average, minimum, and maximum are the same as with discrete statistics. The number of discrete samples is the same as the total weight. Stroboscope uses the sample standard deviation formula to evaluate the standard deviation of a standard Collector:

\[ SD = \sqrt{\left( \sum V^2 - \left( \sum V \right)^2 / n \right) / (n-1)} \]

In the continuous case a definite integral is needed to compute the standard deviation of the heights:

\[ SD = \sqrt{\left( \sum WV^2 - \left( \sum WV \right)^2 / \sum W \right) / \sum W} \]

Weighted Collectors are interesting because they allow the assignment of weights to the different values sent to the Collector. These weights do not have to be integers, they can be any real value. Weighted Collectors are defined with the WGTCOLLECTOR statement:

Syntax: WGTCOLLECTOR CollectorName;
Example: WGTCOLLECTOR SoilHaulTime;

CollectorName is the name of the Collector. The example defines a Weighted Collector named SoilHaulTime.

The following pre-defined system-maintained variables access statistics about the values applied to a Weighted Collector:

- CollectorName.AveVal (weighted average value received by CollectorName)
- *CollectorName*.MaxVal (maximum value received by *CollectorName*)
- *CollectorName*.MinVal (minimum value received by *CollectorName*)
- *CollectorName*.SDVal (weighted std. dev. of values received by *CollectorName*)
- *CollectorName*.TtlWgt (total weight of the values received by *CollectorName*)

Actions on Weighted Collectors require two expressions as arguments (the value being collected and its weight):

**Syntax:**  
EVENT CollectorName [PRECOND LogicalExp] ValueExp WeightExp;

**Example:**  
COLLECT SoilHaulTime 45/60 14.28;

**Example:**  
BEFOREEND Haul COLLECT SoilHaulTime  
Haul.Duration Haul.Soil.Count;

*CollectorName* is the name of the Weighted Collector. *ValueExp* is an expression that Stroboscope evaluates to determine the value it will send to the Collector. *WeightExp* is an expression that Stroboscope evaluates to determine the weight of the value.

The first example instructs Stroboscope to send the value 45/60 with a weight of 14.28 to the *SoilHaulTime* Weighted Collector. This is equivalent to adding a rectangle with a height of 0.75 and a width of 14.28 to the graph that represents the statistics kept by the Weighted Collector. This will happen as soon as the COLLECT control statement is executed.

The second example applies to a situation where the *Haul* Activity represents soil being hauled. The amount of soil hauled in each instance of *Haul* varies. The duration of each instance of *Haul* also varies. The average duration of the *Haul* Activity gives equal weight to all instances of *Haul*, regardless of the amount of soil hauled in each instance. Thus, *Haul*.AveDur is not the average amount of time that soil is hauled, it is the average duration of a haul.

The *SoilHaulTime* Weighted Collector keeps statistics on the amount of time soil is hauled. This is accomplished by collecting statistics on the duration of *Haul*, weighted with the amount of soil hauled. This will happen every time Stroboscope terminates an instance of *Haul*, but before its resources are released. Assume the following details about the instances of *Haul* during a short simulation:
Statistics on the duration of *Haul* from the standard report would indicate $Haul.AveDur=3.27$ and $Haul.SDDur=1.13$. However, statistics kept by the *SoilHaulTime* Weighted Collector would indicate $SoilHaulTime.AveVal=2.99$ and $SoilHaulTime.SDVal=0.93$. The average duration of *Haul* is determined by dividing the sum of the durations (13.07) by the number of instances (4). The average value of *SoilHaulTime* is determined by dividing the area by the width of the following graph:

The two averages obtained are not the same. The average duration of *Haul* does not consider the amount of soil involved in *Haul*. The average value of *SoilHaulTime* reflects that the hauls involving more soil have more impact than the hauls involving less soil (i.e., the wider rectangles have more influence than the narrower ones). Thus, *SoilHaulTime.AveVal* is a better estimator than $Haul.AveDur$ for the amount of time soil is hauled.

The following example illustrates actions on Weighted Collectors through the COLLECT control statement, the values used are the same as in previous examples:
/ define a Collector
WGTCOLLECTOR aCollector;

/ apply some values to the Collector
COLLECT aCollector 18.5 1;
COLLECT aCollector 14 1;
COLLECT aCollector 10 1;
COLLECT aCollector 12/2.75 1;
COLLECT aCollector Log[100]^3 1;

/ display information about the collected values
DISPLAY "Total Weight : " aCollector.TtlWgt;
DISPLAY "Average Value : " aCollector.AveVal;
DISPLAY "Standard Deviation : " aCollector.SDVal;
DISPLAY "Maximum Value : " aCollector.MaxVal;
DISPLAY "Minimum Value : " aCollector.MinVal;
DISPLAY "Sum of the Values : " aCollector.TtlWgt*aCollector.AveVal;

Running a source file with the above code produces the following output:

Total Weight : 5
Average Value : 10.972727
Standard Deviation : 4.8844161
Maximum Value : 18.5
Minimum Value : 4.3636364
Sum of the Values : 54.863636

Note that the above output is almost exactly the same as the output produced by the example for the standard Collector. The difference is the standard deviation, which in this case is smaller by a factor of Sqrt[4/5]. Obviously, the real advantage of Weighted Collectors is the ability to specify different weights. In the example above, all weights were 1 so that the statistics could be compared easily to those produced by previous examples.

8.4.6 Time-Weighted Collectors

Often, Weighted Collectors are weighted by elapsed simulation time. Time-Weighted Collectors combine a SaveValue with a Weighted Collector.

After a Time-Weighted Collector is defined, it can be used as if it were a SaveValue. It can be assigned a value, and used as a variable in expressions. Additionally, a Time-Weighted Collector keeps statistics about the history of values it has acquired, weighted by the simulation time elapsed with each value. Thus, Stroboscope offers the same system-maintained pre-defined variables for Time-Weighted Collectors as it does for Weighted Collectors.
Time-Weighted Collectors are defined with the TMWGTCOLLECTOR statement:

Syntax: TMWGTCOLLECTOR CollectorName InitialValueExp;
Example: TMWGTCOLLECTOR TrucksAway 0;

CollectorName is the Time-Weighted Collector’s name. InitialValueExp is an expression that Stroboscope evaluates to determine the initial value of the Time-Weighted Collector. The InitialValueExp is never evaluated again. Notice that this statement is different from COLLECTOR and WGTCOLLECTOR. This is necessary because the continuous sampling process starts at the simulation time at which the Time-Weighted Collector is defined (time zero if the definition statement is executed before any simulation goes on). Thus, the value of the continuous sampling process must be initialized at that point in time.

The example above defines a Time-Weighted Collector called TrucksAway and sets its initial value to 0. TrucksAway will retain its value of 0 until it is the target of an action that gives it a new value.

Actions are performed on Time-Weighted Collectors the same syntax used for SaveValues and standard Collectors. In the case of Time-Weighted Collectors, both the ASSIGN and COLLECT control statements or action verbs make linguistic sense.

When an action assigns a new value to TrucksAway, Stroboscope will automatically collect statistics using the previous value, and will weigh it with the simulation time elapsed between assignments.

Let us assume that TrucksAway keeps statistics on the number of trucks that are away from the loading site in an earth-moving operation (i.e., trucks involved in Haul, Dump, or Return). At the beginning of the process no trucks are away, thus TrucksAway is initialized with the value 0. Every time Haul starts, the number of trucks that are away increases by 1. Every time Return ends, the number of trucks that are away decreases by 1. The number of trucks that are away only changes at the startup of Haul or the termination of Return. The following statements keep the value of TrucksAway up to date, and at the same time provide statistics on the number of trucks that have been away:
The two statements that follow could use COLLECT instead of ASSIGN
ONSTART Haul ASSIGN TrucksAway TrucksAway+1; /1 truck per instance
ONEND Return ASSIGN TrucksAway TrucksAway-1; /1 truck per instance

The statements above use TrucksAway as a variable to compute the new value (i.e., increment or decrement by one) of the action target (which is also TrucksAway). Thus, TrucksAway is used exactly as if it were a SaveValue. The added benefit is that the statistics on the history of values attained by TrucksAway are available through pre-defined system-maintained variables, and presented in the standard report. Thus, for example, TrucksAway.MaxVal returns the maximum number of trucks that have been away; and TrucksAway.AveVal returns the time-average number of trucks that have been away during the simulation.

The minimum, maximum, and current value of a Time-Weighted Collector can only change when the Collector attains new values. The average, standard deviation, and total weight, however, change continuously as simulation time advances. Every time the simulation engine advances the clock, Time-Weighted Collectors extend the weight of the last value they received by the increment in simulation time. Thus, Time-Weighted Collector statistics are always accurate.

The total weight of a Time-Weighted Collector is the simulation time elapsed since its definition. Since TrucksAway was defined before the simulation ran, TrucksAway.TtlWgt always returns the value of SimTime.

Some situations require a Time-Weighted Collector that does not start at the beginning of the simulation (i.e., when SimTime is still 0). In these cases the Time-Weighted Collector must be defined after the simulation has run for the appropriate amount of time. The following example starts collecting time-weighted statistics for TrucksAway after 200 dumps:

/ ..... Previous statements that do not use TrucksAway
SIMULATEUNTIL Haul.TotInst>=200;
/ now define TrucksAway and the action events that use it
/ define and initialize with the current number of trucks away

/ define the action events that maintain TrucksAway
ONSTART Haul ASSIGN TrucksAway TrucksAway+1;
ONEND Return ASSIGN TrucksAway TrucksAway-1;
It is interesting to note that Stroboscope keeps statistics on Queue contents using a Time-Weighted Collector (i.e., Queue.CurCount is essentially a Time-Weighted Collector). Whenever resources enter or leave a Queue, Queue.CurCount is updated. The pre-defined variables Queue.AveCount, Queue.MaxCount, Queue.MinCount, and Queue.SDCount, simply access the time-weighted statistics of Queue.CurCount.

### 8.4.7 OutFiles & AppFiles

OutFile is a class whose objects represent output devices. These would usually be disk files or windows containing text.

Stroboscope has three pre-defined OutFile Objects: StdOutput, StdError, and StdTrace. These OutFiles represent the standard output, error, and trace devices, respectively. In the Stroboscope Integrated Development Environment, StdOutput is the window that receives the main output of the simulation; StdError is a small window attached to the bottom of StdOutput; and StdTrace can be either a separate window, or be the same as StdOutput (it depends on whether the simulation is run using the “Trace Run” command or not).

OutFiles are defined with the APPFILE and OUTFILE statements:

**Syntax:**

APPFILE FileAlias DiskFileName;

**Example:**

APPFILE TmData “C:\Data\Timedata.sto”;

**Syntax:**

OUTFILE FileAlias DiskFileName;

**Example:**

OUTFILE TmData “C:\Data\Timedata.sto”;

*DiskFileName* is the name of a file on disk. *FileAlias* is the name of the OutFile in the model. The difference between the APPFILE and OUTFILE statements is that APPFILE will add to the existing contents of *DiskFileName*, whereas OUTFILE will overwrite it (i.e., OUTFILE opens a file in create mode and APPFILE opens a file in append mode). APPFILE and OUTFILE have the same effect if the file does not exist before the definition of the object.

Whenever Stroboscope performs an action on an OutFile, it simply sends formatted text output to the device or disk file represented by the OutFile. Actions on OutFiles take a variable number of arguments:
Syntax:  \[ \text{EVENT OutFileName \ [PRECOND LogicalExp] \ "Format String" \ [ExpForPlaceHolder1 ExpForPlaceHolder2 \ldots]; } \]

Example:  \[ \text{PRINT TmData} \]

\[ \text{"The current simulation time is \%.2f\n" SimTime; } \]

Example:  \[ \text{BEFOREEND LunchBreak PRINT TmData} \]

\[ \text{"\%.0f workers just finished a \%0.2f hour lunch!\n"} \]

\[ \text{LunchBreak.Workers.Count LunchBreak.Duration; } \]

\textit{OutFileName} is the name of the OutFile that will receive formatted output.

\textit{Format String} is a string that may contain format specifications and special symbols that indicate tabs, line breaks, etc. Format specifications are substrings that begin with the \% character, and end with any one of the letters e,E,f,F, or G. Format strings are discussed in detail further below. An expression must be supplied for each format specification in \textit{Format String}. When the action is performed, the results of the expressions substitute the format specifications.

The first example instructs Stroboscope to write text similar to “The current simulation time is 456.12” to the file represented by \textit{TmData}. This, of course, assumes that the current simulation time when the PRINT statement is executed rounds to 456.12. The “\n” indicates that subsequent output to the OutFile will start on a new line.

The second example instructs Stroboscope to write text similar to “17 workers just finished a 0.75 hour lunch!” to the file represented by \textit{TmData}. This will happen every time the \textit{LunchBreak} Activity ends (before it releases its resources). Obviously, every time the action is performed, the 17 and the 0.75 will be replaced by the number of workers and duration of the terminating instance.

\textbf{Format Specification}

The \textit{Format String} argument consists of ordinary characters, escape sequences, and (if other arguments follow \textit{Format String}) format specifications. The ordinary characters and escape sequences are copied to the OutFile in order of their appearance.

If any arguments follow the format string they must be expressions; and the format string must contain specifications that determine the output format for these arguments. Format specifications always begin with a percent sign (%) and are read left to right. When the first format specification (if any) is encountered, the value of the first argument after the format string is converted and output accordingly. The second format
specification causes the second argument to be converted and output, and so on. There
must be an equal number of extra arguments as there are format specifications.

**Format Specification Fields**

A format specification, which consists of optional and required fields, has the
following form:

```
%[flags] [width] [.precision] type
```

Each field of the format specification is a single character or a number signifying
a particular format option. The simplest format specification contains only the percent
sign and a type character (for example, %f). The fields in a format specification are
described in Table 8

**Table 8 - Format Specification Fields**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>Required character that determines whether the associated argument is interpreted as a decimal point number, a number in scientific format, or a combination of the two.</td>
</tr>
<tr>
<td>flags</td>
<td>Optional character or characters that control justification of output and printing of signs, blanks, and decimal points. More than one flag can appear in a format specification.</td>
</tr>
<tr>
<td>width</td>
<td>Optional number that specifies minimum number of characters output.</td>
</tr>
<tr>
<td>precision</td>
<td>Optional number that specifies the maximum number of characters printed for all or part of the output field.</td>
</tr>
</tbody>
</table>

If a percent sign is followed by a character that has no meaning as a format field,
the character is copied to the OutFile. For example, to print a percent-sign character, use
```
%%
```

**Type Field Characters**

The type character is the only required format field; it appears after any optional
format fields. The type character determines whether the associated argument is output in
decimal point notation, scientific notation, or a suitable choice between the two, as
indicated by Table 9.
Table 9 - Type Field Characters

<table>
<thead>
<tr>
<th>Character</th>
<th>Output format</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>Signed value having the form [-]dddd.dddd, where dddd is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number, and the number of digits after the decimal point depends on the requested precision.</td>
</tr>
<tr>
<td>e</td>
<td>Signed value having the form [-]d.dddd e [sign]ddd, where d is a single decimal digit, dddd is one or more decimal digits, ddd is exactly three decimal digits, and sign is + or -.</td>
</tr>
<tr>
<td>E</td>
<td>Identical to the e format, except that E, rather than e, introduces the exponent.</td>
</tr>
<tr>
<td>g</td>
<td>Signed value printed in f or e format, whichever is more compact for the given value and precision. The e format is used only when the exponent of the value is less than -4 or greater than or equal to the precision argument. Trailing zeros are truncated, and the decimal point appears only if one or more digits follow it.</td>
</tr>
<tr>
<td>G</td>
<td>Identical to the g format, except that E, rather than e, introduces the exponent (where appropriate).</td>
</tr>
</tbody>
</table>

Flag Directives

The first optional field of the format specification is flag. A flag directive is a character that justifies output and prints signs, blanks, and decimal points. More than one flag directive may appear in a format specification, as shown in Table 10.

Table 10 - Flag Directives

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Left align the result within the given field width.</td>
<td>Right align.</td>
</tr>
<tr>
<td>+</td>
<td>Prefix the output value with a sign (+ or -).</td>
<td>Sign appears only for negative signed values (-).</td>
</tr>
</tbody>
</table>
The second optional field of the format specification is the width specification. The width argument is a nonnegative decimal integer controlling the minimum number of characters printed. If the number of characters in the output value is less than the specified width, blanks are added to the left or the right of the values (depending on whether the - flag (for left alignment) is specified) until the minimum width is reached. If width is prefixed with 0, zeros are added until the minimum width is reached (not useful for left-aligned numbers).

The width specification never causes a value to be truncated. If the number of characters in the output value is greater than the specified width, or width is not given, all characters of the value are printed (subject to the precision specification).

The third optional field of the format specification is the precision specification. It specifies a nonnegative decimal integer, preceded by a period (.), which specifies the number of characters to be printed, the number of decimal places, or the number of
significant digits. Unlike the width specification, the precision specification can cause rounding of a floating-point value.

The interpretation of the precision value and the default when precision is omitted depend on the type, as shown in Table 11.

*Table 11 - Interpretation of Precision and Defaults*

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>e, E</td>
<td>The precision specifies the number of digits to be printed after the decimal point. The last printed digit is rounded.</td>
<td>Default precision is 6; if precision is 0 or the period (.) appears without a number following it, no decimal point is printed.</td>
</tr>
<tr>
<td>f</td>
<td>The precision value specifies the number of digits after the decimal point. If a decimal point appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.</td>
<td>Default precision is 6; if precision is 0, or if the period (.) appears without a number following it, no decimal point is printed.</td>
</tr>
<tr>
<td>g, G</td>
<td>The precision specifies the maximum number of significant digits printed.</td>
<td>Six significant digits are printed, with any trailing zeros truncated.</td>
</tr>
</tbody>
</table>

**Escape Characters**

*Format String* may include special sequences of characters, called *escape sequences*, to represent special symbols. The escape sequences listed in Table 12 are available in Stroboscope.
Table 12 - Escape Sequences

<table>
<thead>
<tr>
<th>Escape</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>writes a line-feed carriage-return pair to the OutFile</td>
</tr>
<tr>
<td>\t</td>
<td>writes a tab to the OutFile</td>
</tr>
<tr>
<td>\ddd</td>
<td>writes the character whose ASCII code is ddd to the OutFile. ddd must be an integer between 0 and 255.</td>
</tr>
<tr>
<td>\</td>
<td>writes a single “\” to the OutFile</td>
</tr>
</tbody>
</table>

Some Examples

Table 13 illustrates several of the possible format specifications.

Table 13 - Format String Examples

<table>
<thead>
<tr>
<th>Format String</th>
<th>Second Argument</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>“[%f]”</td>
<td>5</td>
<td>[5.000000]</td>
</tr>
<tr>
<td>“[% f]”</td>
<td>5</td>
<td>[ 5.000000]</td>
</tr>
<tr>
<td>“[%0.0f]”</td>
<td>4.55467</td>
<td>[5]</td>
</tr>
<tr>
<td>“[%0.2f]”</td>
<td>4.55467</td>
<td>[4.55]</td>
</tr>
<tr>
<td>“[%4.2f]”</td>
<td>4.55467</td>
<td>[4.55]</td>
</tr>
<tr>
<td>“[%12.2f]”</td>
<td>4.55467</td>
<td>[ 4.55]</td>
</tr>
<tr>
<td>“[%012.2f]”</td>
<td>4.55467</td>
<td>[000000004.55]</td>
</tr>
<tr>
<td>“[%-12.2f]”</td>
<td>4.55467</td>
<td>[ 4.55 ]</td>
</tr>
<tr>
<td>“[%+12.2f]”</td>
<td>4.55467</td>
<td>[ +4.55]</td>
</tr>
<tr>
<td>“[%+12.2f]”</td>
<td>-4.55467</td>
<td>[ -4.55]</td>
</tr>
<tr>
<td>“[%12.2e]”</td>
<td>-4.55467</td>
<td>[ -4.55e+000]</td>
</tr>
<tr>
<td>“[%12.2G]”</td>
<td>-4.55467</td>
<td>[ -4.6 ]</td>
</tr>
<tr>
<td>“[%12.2G]”</td>
<td>-0.000000000455467</td>
<td>[ -4.6E-011]</td>
</tr>
<tr>
<td>“[%1.2G]”</td>
<td>-0.000000000455467</td>
<td>[-4.6E-011]</td>
</tr>
<tr>
<td>“[%1.2E]”</td>
<td>-0.000000000455467</td>
<td>[-4.55E-011]</td>
</tr>
<tr>
<td>“[%12f]”</td>
<td>-0.000000000455467</td>
<td>[ -0.000000]</td>
</tr>
</tbody>
</table>
### OutFiles as argument to the REPORT statement

In addition to their use as action targets, OutFiles can be used as argument to the REPORT control statement:

**Syntax:**

```plaintext
REPORT [OutFile];
```

**Example:**

```plaintext
REPORT TmData;
```

When the REPORT statement has an argument, it directs the report to the specified OutFile. If the OutFile, which is optional, is not specified, Stroboscope uses StdOutput.
Chapter 9
Forks And Dynaforks

It is often necessary to make decisions about the routing of resources and the sequence of Activities in a simulation model. Other times it is necessary to model the routing of resources or the start of successor Activities based on a random selection process. This chapter introduces Forks and Dynaforks, which can be used to model decisions and probabilistic selection. They allow the modeling of situations that are conceptually similar to the decision and chance nodes found in decision trees.

Forks and Dynaforks are resource-type specific, auxiliary network nodes. Upon termination of an Activity instance, Dynaforks route resources, and Forks determine which of the successors to activate. In a network drawing, auxiliary nodes are drawn smaller than the regular nodes (e.g., Combis, Normals, and Queues). They are called auxiliary because resources never spend time in these nodes. Auxiliary nodes are essentially accessories to links. Forks and Dynaforks, auxiliary nodes that split one link into two or more links, are discussed below.

9.1 Forks

Forks are shown in a network drawing with a small circle that encloses a triangle. The name of the Fork is placed on top. Forks are defined in a simulation model source file with the FORK statement:

Syntax: FORK ForkName ResourceType [StreamToUse];
Example: FORK RouteHauler Hauler;
**ForkName** is the name of the Fork. **ResourceType** is the type of resource that can go through the Fork. **StreamToUse** is an optional argument that determines which random number stream the Fork uses to choose its successors. When **StreamToUse** is not specified, the Fork uses the default random number stream (i.e., stream 0). (See chapter 17 for details on using multiple streams.)

The example defines a Fork for Haulers called RouteHauler. The Fork uses the default random number stream to choose its successors.

Figure 14 shows a portion of a network that contains one Fork with three successors. The Fork, RouteHauler, is the successor to a Combi, Load. Three links come out of RouteHauler: HL3, HL4, and HL5. The three successors to RouteHauler are the HaulToA and HaulToB Normals and the HaulersWaitToFuel Queue. When a Load instance terminates, RouteHauler selects one of the three successors as the active successor. During the termination of that Load instance, the other two successors are completely ignored.

![Routing Haulers With a Fork Diagram](image)

*Figure 14 - Routing Haulers With a Fork*

Let us assume, for example, that RouteHauler selects HaulToA as the active successor during the termination of a Load instance. HaulToA will receive any haulers released through HL2. When Load is done releasing resources, Stroboscope will create a HaulToA instance. HaulToB and HaulersWaitToFuel will not receive resources. In fact, Stroboscope will not even create a HaulToB instance. During the termination of this Load instance, HaulToB and HaulersWaitToFuel are not considered as part of the model. LoadersWait is a direct successor to Load, it will receive any loaders released through LD2 regardless of RouteHauler’s selection.
Of course, RouteHauler could have selected HaulToB or HaulersWaitToFuel as the active successor. In any case, RouteHauler can select and activate only one of its successors.

Notice that RouteHauler will select one active successor every time a Load instance terminates. This selection remains the same throughout the complete termination and resource-releasing process for that instance. If during the termination of a specific Load instance, resources are released in several steps (this has not been explained yet, but it can be done), all such resources will be routed to the same active successor.

Forks select their active successor very early in the termination process of their predecessor Activity. The process by which a Fork selects its active successor is called the Fork’s resolution. After a Fork has selected its active successor, the Fork has been resolved.

Fork resolution occurs after:

• The BEFOREEND actions of the terminating Activity have been performed, and

• The current instance count of the terminating Activity has been decremented.

Fork resolution occurs before:

• Any resources are released through the links that come out of the terminating Activity.

### 9.2 Dynaforks

Dynaforks are shown in a network drawing with a small circle that encloses five rays. The name of the Dynafork is placed on top. They are defined in simulation model source file with the DYNAFORK statement:

**Syntax:**

```
DYNAFORK ForkName ResourceType [StreamToUse];
```

**Example:**

```
DYANFORK RouteHauler Hauler;
```

*ForkName* is the name of the Dynafork. *ResourceType* is the type of resource that can go through the Dynafork. *StreamToUse* is an optional argument that determines
which random number stream the Dynafork uses to choose its successors. When
StreamToUse is not specified, the Dynafork uses the default random number stream (i.e.,
stream 0). (See chapter 17 for details on using multiple streams.)

The example defines a Dynafork for Haulers called RouteHauler. The Dynafork
uses the default random number stream to choose its successors.

Dynaforks have more routing power than Forks, but they must activate all their
successors. Figure 15 is almost identical to Figure 14, the difference is that a Dynafork
has been substituted for the original Fork.

![Diagram of routing haulers with a Dynafork]

Figure 15 - Routing Haulers With a Dynafork

This network fragment is slightly different to that of Figure 15. RouteHauler is
now a Dynafork and not a Fork. In this case, when an instance of Load terminates,
RouteHauler activates all of its successors: HaulToA, HaulToB, and HaulersWaitToFuel.
When Load’s termination is complete, Stroboscope will have instantiated both HaulToA
and HaulToB.

The purpose of a Dynafork is to decide and route each of the resources it receives
(from the same terminating Activity instance). In the above example, RouteHauler routes
each hauler as it is received. Let us assume (and again, this has not been discussed yet)
that Load releases several haulers through HL2, one at time, during the termination of a
single instance. As each hauler reaches RouteHauler, RouteHauler chooses one of its
successors and sends the hauler there. Suppose, for example, that 6 haulers are released
one-by-one through HL2. At the end of the releasing process, 3 haulers could have been
sent to HaulToA, 2 haulers to HaulToB, and 1 hauler to HaulersWaitToFuel. It is also
possible for RouteHauler to have sent all 6 haulers to HaulToA or to HaulToB, or to
HaulersWaitToFuel. In any case, RouteHauler makes a new decision as to which successor to choose every time it receives and routes a hauler. Regardless of how many haulers are received by HaulToA and HaulToB, both HaulToA and HaulToB will start (even if no haulers reach HaulToA or HaulToB).

Dynaforks need to activate all their successors because Dynaforks cannot determine ahead of time the destination of the resources they will route. All the successors of a Dynafork must be “ready” to receive resources before the terminating Activity instance starts releasing them. Normals get “ready” by setting up a new instance to hold any resources that may be received. Therefore, a Normal will start regardless of whether it received any resources. Queues, on the other hand, are always “ready” to receive resources. If a Queue is activated, but does not receive any resources, it is as if nothing happened to the Queue.

If a Normal should start only when it receives resources, but the Normal is directly or indirectly preceded by a Dynafork, the Normal must be substituted with a Combi preceded by a Queue.

### 9.3 Fork Resolution

Links that come out of Forks and Dynaforks are called Branches. Forks and Dynaforks look at the Strength attributes of its Branches (outgoing) to determine how to route resources. The Strength attribute of a Branch is an expression that must return a nonnegative real number. When a Fork or Dynafork needs to choose a successor, it evaluates the Strength attribute of all the outgoing Branches. The Fork or Dynafork then normalizes these Strengths to convert them to probabilities (i.e., each Strength is divided by the sum of all the Strengths). The Fork or Dynafork then chooses one of the outgoing Branches according to these probabilities.

The default Strength of a Branch is the expression ‘1’. Thus, if all the outgoing Branches of a Fork or Dynafork have default Strength, each successor has an equal probability of being selected. The Strength of a Branch can be changed with the STRENGTH statement:
Syntax:  STRENGTH Branch StrengthExpression;
Example:  STRENGTH HL3 WeightOfA;
Example:  STRENGTH HL4 WeightOfB;

Branch is the name of the link whose Strength is to be changed.

StrengthExpression is the expression that will be used to determine the Strength of the
Branch every time that it is needed.

The examples set the Strength of links HL3 and HL4 to the expressions
‘WeightOfA’ and ‘WeightOfB’, respectively. It is assumed that WeightOfA and
WeightOfB are Variables or SaveValues defined elsewhere. There is no Strength defined
for HL5, so HL5 retains its default Strength of ‘1’ (not 1-WeightOfA-WeightOfB, nor
100-WeightOfA-WeightOfB).

Let us assume that at a time at which RouteHauler (be it a Fork or a Dynafork) is
being resolved, WeightOfA and WeightOfB return 100 and 150, respectively. The sum of
the Strength of all Branches is 100+150+1=251. The probability of selecting HL3 is
100/251=39.84%, the probability of selecting HL4 is 150/251=59.76%, and the
probability of selecting HL5 is 1/251=0.40%. RouteHauler will then obtain a random
number from its stream. If the random number is smaller than 0.3984, RouteHauler
selects HL3. Otherwise, if the random number is smaller than 0.3984+0.5976=0.996,
RouteHauler selects HL4. Otherwise, RouteHauler selects HL5.

The above is an example of the probabilistic resolution of a “chance node”. The
Strengths of Branches can also be specified so that the resolution is deterministic. In such
a case, the Fork or Dynafork represents a “decision” node. To achieve this, the Strengths
of all but one of the Branches must return the value 0. The following code illustrates such
a case:

/ we wish to Haul to the dumpsite that has the smallest contents
/ HaulToA is connected by HL3, it hauls to QueueA,
/ HaulToB is connected by HL4, it hauls to QueueB
STRENGTH HL3 QueueA.CurCount<=QueueB.CurCount;
STRENGTH HL4 QueueA.CurCount>=QueueB.CurCount;
STRENGTH HL5 0; / never go wait for fuel!

In the above example, RouteHauler will select HaulToA whenever the contents of
QueueA is smaller than the contents of QueueB. It will select HaulToB whenever the
contents of QueueB is smaller than the contents of QueueA. If the contents of QueueA
and $QueueB$ are the same, $HaulToA$ and $HaulToB$ have an equal chance of being selected.

### 9.3.1 Fork and Dynafork Chaining

Forks and Dynaforks can be successors to nodes other than Activities. They can be successors to other Forks or Dynaforks, or to other auxiliary nodes (to be examined in other chapters). Fork and Dynaforks cannot be successors to Queues, or predecessors to Combis (i.e., only Queues can precede Combis and only Combis can succeed Queues).

Stroboscope requires that an Activity and all its outgoing link chains form a directed acyclic graph. The root node of the graph is the terminating Activity. All auxiliary nodes are intermediate nodes. Any Normals or Queues are terminal nodes.

Figure 16 shows a hypothetical graph with several Forks and Dynaforks. The root of the graph is $RootNode$. The intermediate nodes are the $Int1$ and $Int7$ Forks, and the $Int2$ and $Int4$ Dynaforks. The terminal nodes of the graph are the $Terminal3$, $Terminal5$, and $Terminal6$ Queues, and the $Terminal8$ and $Terminal9$ Normals. All links that begin with “L” are part of the graph. When more than one link goes out of the same node, the relative order of definition in the corresponding simulation input file is indicated by the superscripts.

![Figure 16 - Release Graph With Forks and Dynaforks](image-url)
Forks and Dynaforks are resolved in the order in which they are encountered in a depth-first traversal of the graph. Ignoring Fork resolutions, a depth-first traversal of the above graph would visit the nodes in the following order: Int1, Int2, Terminal3, Int4, Terminal5, Terminal6, Int7, Terminal6, Terminal8, Terminal9. Notice that Terminal6 appears twice in the preceding list because it has two incoming links.

When an instance of RootNode terminates, and before any resources are released, the graph must be resolved. This is done by resolving the Forks (Dynaforks come into the picture later). Depending on the Strength of the Branches, Stroboscope can resolve the above graph in three possible ways:

1. Int1 chooses L2, then Int7 chooses L8:
   - RootNode activates Int1
     - Int1 is resolved, it chooses L2
     - Int1 activates Int2
       - Int2 activates Terminal3
       - Int2 activates Int4
         - Int4 activates Terminal5
         - Int4 activates Terminal6
     - Int2 activates Int7
       - Int7 is resolved, it chooses L8
       - Int7 activates Terminal6 (Terminal6 ignores the activation since it was already activated by Int4)
   - The active nodes, in the order of their activation, are Int1, Int2, Terminal3, Int4, Terminal5, Terminal6, Int7, and Terminal6.
   - Note that, in this case, no Activity instance starts as a result of the termination of RootNode.

2. Int1 chooses L2, then Int7 chooses L9:
• *RootNode* activates *Int1*
  
  • *Int1* is resolved, it chooses *L2*
  
  • *Int1* activates *Int2*
    
    • *Int2* activates *Terminal3*
    
    • *Int2* activates *Int4*
      
      • *Int4* activates *Terminal5*
      
      • *Int4* activates *Terminal6*
    
    • *Int2* activates *Int7*
      
      • *Int7* is resolved, it chooses *L9*
      
      • *Int7* activates *Terminal8*
  
  • The active nodes, in the order of their activation, are *Int1, Int2, Terminal3, Int4, Terminal5, Terminal6, Int7, and Terminal8*.
  
  • Note that, in this case, an instance of *Terminal8* starts as a result of the termination of *RootNode*.

3. *Int1* chooses *L10*:

• *RootNode* activates *Int1*
  
  • *Int1* is resolved, it chooses *L10*
  
  • *Int1* activates *Terminal9*
  
  • The active nodes, in the order of their activation, are *Int1, and Terminal9*.
  
  • Note that, in this case, an instance of *Terminal9* starts as a result of the termination of *RootNode*. An instance of the *Terminal8* Normal Activity will start later (perhaps at another simulation time), during the termination of the instance of *Terminal9* just created. Also note that in this case *Int7* does not need to be resolved.
Once the Forks in the graph are resolved, resources are released through the outgoing links of RootNode (Stroboscope knows the path through which resources are to be released except for branches that emanate out of Dynaforks). Depending on which nodes became active during Fork resolution, a resource released through \( L1 \) can end up in any of the \( \text{TerminalX} \) nodes. In fact, if links \( L2 \) and \( L8 \) were chosen during Fork resolution, a resource could end up in \( \text{Terminal6} \) through two different routes. Of course, it could also end up in \( \text{Terminal3} \) or \( \text{Terminal5} \). This is true for all resources released during the same termination.

### 9.4 New Programming Facilities

Different combinations of the terminal nodes can become active during the termination of an Activity instance whose outgoing link graph contains Forks. Those Normals that become active are said to be “in context”. Their instance variables are accessible and can be used in any expression that is evaluated while the Activities are “in context”. Queues do not have instance variables and are always “in context” (regardless of whether they are active or not). The logical pre-defined system-maintained variable \( \text{HeteroHolder.InContext} \) returns TRUE if \( \text{HeteroHolder} \) is “in context”. A HeteroHolder is a node that can hold resources of more than one resource-type. Activities are HeteroHolders because resources of different types can reside in them. Other nodes not yet discussed are also HeteroHolders.

If Stroboscope evaluates an expression that accesses an Activity’s instance data while the Activity is not in context, Stroboscope stops the simulation and issues a runtime error. \( \text{HeteroHolder.InContext} \) must be checked before accessing instance data for an Activity that may not be in context. The following example assumes that the “L” links in the above network carry resources of type \( \text{Airplane} \):

```stroboscope
/show in standard trace the number of planes released by RootNode that end up in Activities (as opposed to in Queues)
ONEND RootNode StdTrace "%.0f planes left RootNode for active work!\n"
```

Copyright © 1996 Julio C. Martínez
The InContext variables used above are necessary because it is not certain that $Terminal_8$, or $Terminal_9$, or neither of them, are in context. Accessing $Terminal_8.Airplane.Count$ or $Terminal_9.Airplane.Count$ when $Terminal_8$ or $Terminal_9$, respectively, are not in context, produces a runtime error. Notice that it is not possible for both $Terminal_8$ and $Terminal_9$ to be in context simultaneously because $Int1$ is a Fork and not a Dynafork.

Actions can be performed on action targets when resources flow through links other than Draw Links or Release Links (such as links out of Forks and Dynaforks). The ONFLOW statement must be used in these cases (instead of ONDRAW or ONRELEASE):

Syntax:  

\[
\text{ONFLOW LinkName ActionTarget [PRECOND LogicalExp] [TargetArg1 TargetArg2 ...];}
\]

Example:  

\[
\text{ONFLOW L6 StdTrace "A plane entered Terminal6 through L6\n";}
\]

Example:  

\[
\text{ONFLOW L8 StdTrace "A plane entered Terminal6 through L8\n";}
\]
Chapter 10
Consolidators

It is often necessary to accumulate resources of possibly different types until the resources accumulated meet a certain condition. When the condition is met, the resources can be routed to other nodes. For example, prefabricated concrete beams could be placed on a flatbed trailer. When the beams to transport have been placed in the trailer, the trailer and beams leave.

Similarly, it is often necessary to model a gate that blocks the flow of resources to the succeeding nodes until some condition is satisfied. An example of this is when road vehicles are blocked at a railway crossing until the train goes by and clears the tracks. All the vehicles can then go through.

Situations like these may be modeled by a Combi and a group of dedicated Queues that precede the Combi. The resources enter the different Queues and stay there until the Combi can start. The condition for the Combi to start is the condition the resources must meet. When the Combi starts, it removes all the resources from all the Queues that precede it, packs them into an instance of the Activity, and places the instance in the FEL. When the instance is terminated, the Release Links of the Activity route the resources to the desired destinations.

In cases such as these it would be convenient to merge a Combi and the Queues that precede it into a single node. Stroboscope provides a kind of Activity called a Consolidator, which can be used as a substitute for a Combi and its preceding Queues. In order for this substitution to make sense, however, the Combi must be the only successor
to the Queues that precede it. This is so that the entire set of nodes can be grouped into one (a Consolidator) without affecting other nodes in the network.

**10.1 Overview**

Consolidators are Activities that start and finish their instances depending exclusively on the resources they receive. Resources can enter Consolidator instances at any time during the lifetime of the instance. All the resources are released when the instance is terminated. When a Consolidator receives a resource, the reception can trigger the start of a new instance, or the end of an instance, or neither, or both. In any case, the resource is added to the instance that is starting, continuing, or ending. The details of exactly how a Consolidator reacts to the reception of a resource are discussed in section 10.2 below.

Consolidators are defined with the CONSOLIDATOR statement:

**Syntax:**

```
CONSOLIDATOR ConsolidatorName;
```

**Example:**

```
CONSOLIDATOR ContainerIsFull;
```

*ConsolidatorName* is the name of the Consolidator. The example defines a Consolidator named *ContainerIsFull*. In this case, the Consolidator’s name describes a condition for releasing the resources held. Perhaps a more descriptive name could be *WaitUntilContainerIsFullAndThenReleaseEverything*. Obviously, it is not practical to use such a name due to its length. Due to the multi-faceted nature of Consolidators, choosing an appropriate name is largely a matter of context and taste.

**10.2 Consolidator Reactions to Resource Receptions**

A Consolidator is always in one of two states: “consolidating”, or “not consolidating”. At the beginning of a simulation, a Consolidator is “not consolidating” and has no current instances.

When a Consolidator is in a “not consolidating” state and receives a resource, it creates an instance of itself. The number of current instances of the Consolidator
increases, the OnStart actions for the Consolidator are performed, and the Consolidator changes its state to “consolidating”.

The instance held by a Consolidator in a “consolidating” state is “open”. Any new arrivals to a Consolidator do not start a new instance. Instead, these resources become part of the “open” instance. The instance remains “open” until the consolidation condition is satisfied (that is, the instance is Consolidated) as explained below. The consolidation does not necessarily occur during the termination of the predecessor instance that started the Consolidator, or at the simulation time at which the instance started. Thus, the resources that become part of the instance can enter it at different simulated times and can come from various sources.

After a Consolidator receives a resource, including the resource that starts the “open” instance, the Consolidator evaluates its ConsolidateWhen attribute. If the attribute evaluates to FALSE, the Consolidator does nothing. If the attribute evaluates to TRUE, the Consolidator consolidates the open instance. At this point, the Consolidator will “close” the instance and set the instance’s end-time equal to the current value of the simulation clock. It is at this point that the duration of the instance becomes known and simply equals the simulation time elapsed between its start and its end. The Consolidator then inserts this instance into the FEL, and the Consolidator state changes to “not consolidating”. Since instances of Consolidators enter the FEL at their end-time, they are placed towards the front of the FEL behind any other instances whose end-time matches the current value of the simulation clock.

The ConsolidateWhen attribute of a Consolidator can be changed with the CONSOLIDATEWHEN statement:

Syntax:  
CONSOLIDATEWHEN ConsolidatorName LogicalConsolidationExp;

Example:  
CONSOLIDATEWHEN ContainerFull
 'ContainerFull.Container.Count & ContainerFull.SteelShape.Count >= ContainerCapacity';

*ConsolidatorName* is the name of the Consolidator whose ConsolidateWhen attribute is changed. *LogicalConsolidationExp* is the logical expression that will be used by the Consolidator to determine when it should consolidate. The example changes
ContainerFull’s ConsolidateWhen attribute. ContainerFull will consolidate whenever it contains a container and enough steel shapes to fill the container.

The default ConsolidateWhen expression is ‘1’, thus it always returns TRUE and is rarely used without being modified. However, it is instructive to understand how the default behavior works. A Consolidator with a default ConsolidateWhen will end as soon as it starts. If the termination of a single instance of a predecessor releases several resources to such a Consolidator, then a separate instance of the Consolidator will start and end for each resource. Each instance will have a duration of zero. All such Consolidator instances start and end at the same simulation time and are terminated one by one through the standard processing of the FEL. A Consolidator with a default ConsolidateWhen resembles a zero-duration Normal that, instead of starting once with all the resources it receives, can start many times (one start for each resource). Notice, that unlike a Normal, a Consolidator does not start if it does not receive resources from the terminating predecessor.

When referring to generic resources (the only kind of resources examined so far), “a resource” means any amount of a generic resource that is transmitted, including the amount zero. Generic Release Links release exactly once. What is released is “a resource”, whose amount can be 0 or any real number greater than zero. The number of releases is always one because there is no “ReleaseUntil” for Generic Release Links (although Characterized Release Links do have one). For this reason, unless some Forks or Dynaforks exist as auxiliary nodes in a graph in which a Consolidator is a terminal node (see Figure 16), a Consolidator will receive exactly one resource through each generic incoming link during the termination of an instance of a predecessor.

10.3 Consolidator Instances

Although a Consolidator can have several “current instances”, only one of them can be “open” at the same time. The time during which the instance is open is the duration of the instance. For this reason, and in contrast to instances of Normals and Combis, instances of Consolidators never overlap. Even if several zero-duration instances of a Consolidator start at the same simulation time, their lifetimes are still sequential.
When an “open” instance is “closed”, it is inserted in the FEL. Since instances of Consolidators enter the FEL at their end-time, which is by definition the same as SimTime, their presence in the FEL indicates that the FEL contains current events. These instances will be terminated as part of the current FEL processing. For more than one instance of the same Consolidator to be in the FEL, all but the front-most of these instances must be of zero-duration.

Consolidators are always ready to receive resources. As a result, they are always in context and their resource-content instance variables are always accessible.

Each of the current instances of a Consolidator can be in one of three states. One instance can be open, several instances can be in the FEL, and one instance can be in the termination process. This raises an interesting question. Which Consolidator instance is accessed by a variable such as “Consolidator.Soil.Count”? The instance variables of Activity instances that are in the FEL are never accessible, so these are not accessed. When an instance of a Consolidator is terminating, its instance variables should be accessible for use in the Consolidator’s BeforeEnd and OnEnd actions. They should also be available for use in any attributes of the outgoing Release Links. For this reason, during the termination of a Consolidator instance, the instance variables access the terminating instance, not the “open” instance. At all other times, a Consolidator’s resource-content instance variables access the “open” instance.

Consolidator instance variables not related to resource-content (i.e., Consolidator.Instance and Consolidator.Duration) are only available for terminating instances. If these instance variables are accessed while a Consolidator instance is not being processed for termination, Stroboscope issues a runtime error. The unavailability of these variables for the open Consolidator instance is not a problem because, for the open instance, Consolidator.TotInst and (SimTime-Consolidator.LastStart) can be used as substitutes.

10.4 Contrast to Other Nodes

In terms of starting new instances, Consolidators differ from Normals. Normals start their instances upon the termination of a predecessor instance (regardless of whether
any resources are received). Consolidators differ from Combis in that Combis start their instances when they are given a chance to start, and the conditions necessary for the Combi to start exist.

Consolidators differ from both Normals and Combis in the relationship between instance duration and finish time. Normals and Combis determine their duration from their Duration attribute. From the duration and start-time they determine their end-time (which is known at the time they start). Consolidator instances determine their end-time when they receive the last resource for the instance. The instance then determines its duration by subtracting the start-time from the end-time. Consolidator instances enter the FEL at their end-time because prior to that time their end-time is not known. This is in contrast to Combi and Normal instances, that know their end-time when they start and thus enter the FEL at their start-time.

Consolidators, like Normals, do not take an active role in obtaining resources. This is in contrast to Combis, which draw resources from the Queues that precede them. For this reason, Consolidators may not be preceded by Queues. Only other Activities (which includes other Consolidators) or auxiliary nodes may precede a Consolidator. Like all Activities, Consolidators are not type-specific nodes. Thus, the incoming and outgoing links for a Consolidator can be of any resource type.

Stroboscope makes no distinction between Combis, Normals, and Consolidators with regard to the statistics and system-maintained variables it keeps. The same is true with respect to the process of terminating instances. This is because once Activity instances are in the FEL (be they Combi, Normal, or Consolidator instances), they are treated in the same manner.

In some respects, Consolidators behave like Queues. An instance of a Consolidator Activity is similar to a Queue in that it stores resources that do not necessarily enter the instance at the same time. Furthermore, the time at which these resources leave the instance is not known until they are actually released (when the Consolidator instance is terminated during a Clock Advance Phase). Unlike Queues, Consolidator instances can contain resources of different resource types. Due to the similarities between Consolidator Activities and Queues, the left part of a Consolidator (symbolizing the resource receiving part) in a network drawing is round.
10.5 Examples

One of the operations involved in a quarry is to drill holes in the rock face. When a batch of 25 holes are drilled, the holes are loaded with dynamite one by one. The annotated network fragments shown below illustrate three alternative ways of modeling this situation (possible interactions with other resources and processes in the quarry have been ignored for simplicity).

Figure 17 contains two Queues for holes. *HolesDrilled* contains holes that are drilled but are not yet batched into a group of 25 holes. *HolesToLoad* contains holes that have already been batched and are ready to be loaded. *BatchHoles* moves the holes from one Queue to the other whenever enough holes to form a batch exist in *HolesDrilled*. This is accomplished through the attributes of link *HO2*.

![Diagram](image.png)

**Figure 17 - Consolidating with Two Queues and a Combi**

Figure 18 shows a solution that uses fewer nodes and accomplishes the task with a more elaborate Enough attribute in link *HO2*. *CompSetsDrilled* indicates the number of 25-hole batches that have been drilled. *CompSetsLoaded* indicates the number of 25-hole batches that have left *HolesDrilled*. *HO2*’s Enough uses these two variables to determine if *HolesDrilled* contains holes that have been batched. Holes that have not yet been batched into a group of 25 do not satisfy the Enough attribute. The solution of Figure 18 relies heavily on Stroboscope’s programmability. It is harder to understand and more prone to error than the solution of Figure 17.
Variables:
CompSetsDrilled = Int[HolesDrilled.TotCount/25]

Link attributes:
Enough = CompSetsDrilled > CompSetsLoaded & HolesDrilled.CurCount

Figure 18 - Consolidating With One Queue

Figure 19 shows a solution that uses a Consolidator in place of the HolesDrilled Queue and the BatchHoles Combi of Figure 17. The BatchHoles Consolidator accumulates holes until it contains 25 of them. The Consolidator then releases the 25 holes to HolesToLoad. This solution is simpler and more straight-forward than the other two solutions.

Figure 19 - Consolidating With a Consolidator and a Queue

Figure 20, below, shows a network that uses a Consolidator to model a variation of the earth-moving operation presented in earlier chapters. In this example, the setup of the truck and the loading of each scoop of soil are modeled as separate Activities. The loader is not committed to filling a truck completely in one step. If the loader were required by other Combis, it could leave a truck partially loaded and then come back to finish loading the truck later.

This network uses the Space resource to ensure that only one truck is loaded at the same time. This is achieved by initializing Spot with only 1 unit of Space (a Semaphore can also be used to model this). When SetupTruck starts, it removes the space from Spot and a truck from TrucksWait. When it finishes, it places the space and truck in TruckFull. This makes TruckFull start. TruckFull will not consolidate until it contains a truck and a truckload of soil:
The first part of the ConsolidateWhen becomes TRUE when \textit{TruckFull} receives a truck. The second part will become TRUE when \textit{TruckFull} has received enough soil from \textit{LoadScoop}.

\textit{LoadScoop} is a Combi that represents the loading of a single scoop of soil to the truck. In order for it to happen, (in addition to the default requirement of having a loader in \textit{LoadersWait} and soil in \textit{SoilToMove}), \textit{TruckFull} must already have a truck to receive the scoop:

\begin{verbatim}
SEMAPHORE LoadScoop 'TruckFull.Truck.Count';
\end{verbatim}

The Semaphore expression uses \textit{TruckFull} instance variables. This is acceptable because Consolidators are always in context.

As instances of \textit{LoadScoop} terminate and release soil to \textit{TruckFull}, \textit{TruckFull} evaluates its ConsolidateWhen attribute to see if it should consolidate. Consolidation occurs when the amount of soil in \textit{TruckFull} is greater than or equal to the capacity of the

\begin{verbatim}
CONSOLIDATEWHEN TruckFull
  'TruckFull.Truck.Count & TruckFull.Soil.Count>=TruckSize';
\end{verbatim}
truck. This marks the end-time of the current instance of TruckFull. The instance is then placed in the FEL and will be terminated after LoadScoop finishes terminating. TruckFull turns to a “not consolidating” state, and will remain like that until it receives a truck from SetupTruck. When the TruckFull instance is terminated, the Space will be placed in Spot, and the truck and soil will go to Haul.

In order to complete the model shown in Figure 20, it is necessary to specify the Duration attributes of the different Activities, to set SL1’s DrawAmount attribute, to initialize the remaining Queues, to run the simulation, and to print a report.

The model described above has the drawback that only one truck can be loaded at the same time. It has the advantage (or disadvantage, depending on the case) that the loader can be preempted from loading a truck. In order to model more than one simultaneous truck loading, the Consolidator must be substituted with a Combi and two dedicated Queues, one for soil and one for the truck (being loaded as opposed to waiting). Compound resources (to be introduced later) can be used as another alternative.

10.6 Recap

The following observations on Consolidators are worth noting:

- Consolidators are always in context.

- A Consolidator is in one of two states: “Consolidating” or “Not Consolidating”.

- The duration of a Consolidator instance spans from the time at which the Consolidator receives its “activating” resource, to the time of Consolidation.

- The duration of a Consolidator instance is not known until the instance is Consolidated.

- Consolidator instances of are placed in the FEL only after they are consolidated (at their end-time).

- Different resources can spend different amounts of time in the same Consolidator instance.
• The default ConsolidateWhen will cause a Consolidator to create a zero-duration instance of itself every time the Consolidator receives a resource.

• Many Consolidator instances can be created and inserted in the FEL during the termination of a single instance of a preceding Activity.

• The ConsolidateWhen expression is evaluated only when a Consolidator receives a resource. If it does not receive a resource it cannot end. If the ConsolidateWhen expression is not resource-dependent, there is no guarantee that the Consolidator will consolidate promptly and accurately.
Chapter 11

Characterized Resources

Resources can be bulk or discrete depending on their type. The distinction between bulk and discrete resources was discussed thoroughly in section 3.1 starting on page 19.

Preceding chapters were limited to “Generic” resources. Discrete resources had been represented by limiting generic resources to integer amounts. This chapter introduces discrete, uniquely identifiable resources, which in Stroboscope are called “Characterized” resources.

11.1 Characterized Types, Subtypes, and Resource Properties

Most real world processes involve discrete, uniquely identifiable resources. Some of these resources are completely different from others in that they represent different kinds of things. Such resources belong to different resource types. A steel shape, for example, is completely different from a bulldozer. The steel shape could be classified as a resource of type SteelShape while the bulldozer as a resource of type Tractor.

Resource types that represent discrete, uniquely identifiable resources, are called “Characterized Resource Types”. Individual resources of these types are called “Characterized Resources”.

Copyright © 1996 Julio C. Martínez
Characterized resources of the same type share common traits. For example, all resources of type *Loader* have a bucket of some size, a fuel tank of some capacity, some amount of fuel currently in the tank, an engine with a specific power rating, and a unique serial number. Individual loaders can be different from each other in that the size of their buckets, the capacity of their fuel tanks, the amount of fuel in their tanks, the power rating of their engines, or their serial numbers are different.

Characterized resources are “characterized” by their “properties”. Loaders, for example, may have properties named *BucketSize*, *TankCapacity*, *FuelLeft*, *Power*, and *SerialNumber*. All loaders will have properties with those names, but the values attached to each property may differ among loaders. The type of the resource determines which properties are borne by the resource, but not the specific property values.

The values of some resource properties do not change during the life of a resource. For example, once a loader is manufactured, its bucket size, tank capacity, engine power, and serial number do not change (ordinarily — in reality some of these properties can change). The values of other resource properties do change with time. For example, the amount of fuel left in a loader’s tank changes when the tank is filled or as fuel is consumed.

Characterized Resource Type properties can be classified as fixed (during the life of a resource) or changeable. They can be further classified as described below.

### 11.1.1 Characterized Types and SubTypes

Within a resource type, some groups of resources have common values for some of their properties. All loaders of a specific model, for example, have the same bucket size, tank capacity, and engine power. In order to capture this commonality within groups of resources, every characterized resource belongs to a specific “SubType”. Properties that have fixed, common values within a SubType, are called “SubType Properties”. The SubType determines the values of the “SubType Properties” for a characterized resource.
In the case of loaders, for example, SubTypes could correspond to loader models. 
$S916$ and $S926E$ could be SubTypes of the characterized type Loader. The power for all 
$S916$ loaders could be 63 kW, while the power for all $S926E$ loaders could be 82 kW.

The definition of a Characterized Resource Type includes the name of the 
resource type as well as the “SubType Properties”:

Syntax: \( \text{CHARTYPE \{TypeName\} \{STProp1\} \{STProp2\} \ldots \{STPropN\};} \)

Example: \( \text{CHARTYPE Loader BucketSize TankCapacity Power;} \)

\( \text{TypeName} \) is the name of the Characterized Resource Type. \( \text{STProp1, STProp2,} \)
and so forth until \( \text{STPropN} \) are the “SubType Properties” for resources of type 
\( \text{TypeName} \). The example defines \text{Loader} as a Characterized Resource Type that has 
“SubType Properties” named \( \text{BucketSize, TankCapacity, and Power.} \)

Characterized Resources \textbf{must} belong to a SubType that provides the values for 
the “SubType Properties”. SubTypes are defined as follows:

Syntax: \( \text{SUBTYPE \{TypeName\} \{SubTypeName\} \{Prop1\} \{Prop2\} \ldots \{PropN\};} \)

Example: \( \text{SUBTYPE Loader S916 1.5 123 63;} \)
Example: \( \text{SUBTYPE Loader S926E 1.9 150 82;} \)

\( \text{TypeName} \) is the name of the Characterized Resource Type. It must have been 
defined previously. \( \text{SubTypeName} \) is the name of the SubType being defined. \( \text{Prop1, Prop2,} \)
and so forth until \( \text{PropN} \) are the values of the “SubType Properties”. The number 
of property values must match the number of property names listed with the definition of 
the corresponding characterized resource type. If three property names were listed in the 
definition of \( \text{TypeName} \), then three property values must be listed in the definition of 
\( \text{SubTypeName} \). The property values may be defined with expressions, but the expressions 
are only evaluated once, when the SubType is defined. (VarProps, to be described later, 
are automatically recalculated based on the expressions used to define them.)

The examples define two Loader SubTypes: \( S916 \) and \( S926E \). If these are the 
only Loader SubTypes that have been defined, then any resource of type Loader that is 
created must be of one of these two SubTypes. In fact, as will be seen later, the SubType 
must be specified when a characterized resource is created. Loaders whose SubType is 
\( S916 \) have a 1.5 m$^3$ bucket, a 123 liter tank, and a 63 kW engine. \( S926E \) loaders have a 
1.9 m$^3$ bucket, a 150 liter tank, and an 82 kW engine.
The statements that define a Characterized Resource Type and the corresponding SubTypes are usually placed sequentially in a source file so that they look like a table. The examples above would typically be defined as follows:

\[
\begin{array}{llll}
\text{CHARTYPE} & \text{Loader} & \text{BucketSize} & \text{TankCapacity} & \text{Power} \\
/ & (\text{m}^3) & (\text{Liters}) & (\text{kW}) \\
\text{SUBTYPE} & \text{Loader} & S916 & 1.5 & 123 & 63 \\
& & S926E & 1.9 & 150 & 82 \\
\end{array}
\]

The property values defined by a SubType are fixed. The values do not need to be attached to a resource to have meaning. Hence, they are global in scope. The values are accessible through pre-defined system-maintained variables of the following form:

\[
\text{SubTypeName.PropertyName}
\]

Once the statement that defines SubType \textit{S926E} has been executed, for example, the variable \textit{S926E.Power} is available for use in any expression. It will always return the value 82.

The hierarchy of Characterized Resource Types in Stroboscope has a two-tier fixed depth (Type, SubType). A resource must be of a particular SubType and as such it automatically belongs to the corresponding Characterized Type. There is no third tier (a Sub-SubType) derivation. The role of the SubType is limited to the specification of fixed property values (SubTypes cannot provide new property names). Furthermore, Characterized Resource Types are “abstract” (i.e., a resource \textbf{must} belong to a SubType).

\subsection{11.1.2 SaveProps}

Some resource properties are not fixed throughout the lifetime of the resource. The values of these properties for a particular resource change as simulation proceeds. This is the case of the \textit{FuelLeft} loader property used as an example earlier. These properties are essentially SaveValues attached to a specific characterized resource. These properties are called “SaveProps”.

SaveProp names are defined at the resource type level with the \texttt{SAVEPROPS} statement:
Syntax: `SAVEPROPS TypeName [SaveProp1] [SaveProp2][....] [SavePropN];`

Example: `SAVEPROPS Loader FuelLeft AmountLoaded;`

Example: `SAVEPROPS Loader HoursOperated;`

`TypeName` is the characterized resource type for the SaveProps. `SaveProp1`, `SaveProp2`, and so on up to `SavePropN` are the names of the SaveProps being defined.

Several `SAVEPROPS` statements for the same resource type may appear in a simulation input file. SaveProps defined last do not substitute for those defined earlier; they extend the list of SaveProps for the corresponding resource type.

The examples define the `FuelLeft`, `AmountLoaded`, and `HoursOperated` SaveProps for resources of type `Loader`. After both statements are executed, every resource of type `Loader` that is created will possess these three properties (in addition to the “SubType Properties”). Every resource will have a separate value for these properties. As will be explained later, the SaveProp of a specific resource can obtain a new value when it is the target of an ASSIGN action. When a resource is created, all its SaveProps have a value of zero.

All the `SAVEPROPS` statements for a particular resource type must be executed before any resources of the corresponding type are created.

### 11.1.3 System-Maintained Properties

In addition to “SubType Properties” and “SaveProps”, all characterized resources have the following three properties: `BirthTime`, `ResNum`, and `TimeIn`.

`BirthTime` and `ResNum` are fixed properties. Their values are determined when the resource is created and do not change throughout the life of the resource. The value of the `BirthTime` property is the value of the simulation clock at the time at which the system creates the resource. The `ResNum` property is like a serial number that is unique within resources of the same type. When Stroboscope creates the first resource of a given type, it sets the resource’s `ResNum` property to the value 1. The `ResNum` property of the second resource of a given type to be created has a value of 2, etc.

`TimeIn` is a variable property that is updated every time the resource enters a Queue or an Activity. Its value is the simulation time at which the resource entered the
last node it visited. The \textit{TimeIn} property is useful in establishing resource disciplines based on arrival order, such as FIFO and LIFO.

\section*{11.1.4 Variable Properties}

Resource properties can be functions of other properties of the resource. Loaders, for example, can be characterized according to the amount of fuel left in their tanks expressed as a percentage of their capacity. A property called \textit{PctFuelLeft} can be used to characterize loaders in this way. Such a property depends on two other properties, \textit{FuelLeft} and \textit{TankCapacity}.

Properties that depend on the values of other properties are called “Variable Properties” or “VarProps”. These properties are defined with the following syntax:

\textbf{Syntax:} \textit{VARPROP} \textbf{TypeName} \textbf{VarPropName} \textbf{ExpressionInTermsOfOtherProps};

\textbf{Example:} \textit{VARPROP Loader PctFuelLeft FuelLeft/TankCapacity};

Once defined, a VarProp is like any other resource property. Whenever it is used, a VarProp returns an up to date value. A VarProp is much like a Variable (defined with the \textsc{VARIABLE} statement), except that it is attached to a resource and references other properties of the resource implicitly.

The example defines \textit{PctFuelLeft} as a loader VarProp. \textit{PctFuelLeft} always returns a value obtained by dividing the resource’s \textit{FuelLeft} property by its \textit{TankCapacity} property.

The expression that defines a VarProp can reference global symbols (e.g., Variables, \textsc{SaveValues}, system-maintained variables) in addition to other resource properties. The following example illustrates this:

\textit{VARPROP Loader TimeWaited SimTime-TimeIn};

VarProps can be defined at any point in a simulation model file; even after resources of the type have been created. Obviously, they cannot be used prior to their definition.
11.2 Creating Characterized Resources

The definition of characterized resource types and SubTypes does not create resources. Characterized types and SubTypes just specify the characteristics of resources based on them. Characterized resources are real, carry unique information with them, and must occupy a “physical location” in a model network. This location can be a Queue, an instance of an Activity (Combi, Normal, or Consolidator), or an auxiliary node (auxiliary nodes can hold a characterized resource only for an “instant”, during the termination of an Activity instance).

Characterized resources have a life that starts when they are created and ends when they are destroyed. Stroboscope creates characterized resources when initializing Queues during the execution of INIT statements (to be discussed below). Stroboscope can also create resources while the simulation is running by acting on the GENERATE action target (to be discussed in the next chapter) at the occurrence of the BEFOREDRAWS or BEFOREEND action events. Stroboscope destroys a resource when the Activity instance that holds it does not release it. Stroboscope will not destroy a characterized resource while the resource is in a Queue.

Structure of a resource

Characterized resources carry information with them as shown in Figure 21. A characterized resource does not carry the individual values of its “SubType Properties”. These values are stored in only one place, which is common for all resources of the given SubType. As a consequence, the number of “SubType Properties” has little impact in the space required to store a specific resource’s information.

VarProp formulas are not stored with the resource either. They are stored in a place that is common for all the resources of the given type (i.e., shared by all the SubTypes). VarProp values are never stored, they are calculated on demand by evaluating the formulas. As a consequence, resources can have a very large number of VarProps without affecting the storage requirements of individual resources.

The number of SaveProps, on the contrary, does impact the space required to store a specific resource’s information. This is because each individual resource has its
own value for each SaveProp. The space required to store a resource’s information is proportional to the number of SaveProps.

The storage issues mentioned above are only important for runtime memory considerations. For all practical purposes, and for the discussions that follow, it is assumed that “SubType Properties” and “VarProps” are carried by resources directly.

Creating Resources and Placing them in Queues before Simulating

Characterized resources are created and placed in Queues through the execution of INIT control statements:

Syntax: INIT QueueName nResources SubTypeName;
Example: INIT LoadersWait 3 S916;
Example: INIT LoadersWait nBigLoaders S926E;

*QueueName* is the name of the Queue where the resources will be placed. *nResources* is the number of resources to be created. If *nResources* is not an integer Stroboscope truncates it. *SubTypeName* is the SubType of the resources (note that the type is not used, it is inferred from the SubType).

The first example creates 3 loaders of SubType S916 and places them in *LoadersWait*. The second example creates *nBigLoaders loaders of SubType S926* and
places them in \textit{LoadersWait}. \textit{nBigLoaders} is assumed to be a variable or \textit{SaveValue} defined previously, and that indicates the number of “big” loaders. Assuming that the value of \textit{nBigLoaders} is 7, the two statements will create and place 3 \textit{S916} loaders \textbf{and} 7 \textit{S926E} loaders in \textit{LoadersWait}. As will be described later, the order occupied by these 10 loaders within \textit{LoadersWait} is determined by the Queue’s discipline.

INIT control statements are usually placed before any \textit{SIMULATE} or \textit{SIMULATEUNTIL} statements in order to make the resources available during simulation.

\section*{11.3 Accessing Resource Properties}

Characterized resource properties affect the behavior of simulations. Resource properties can be used to calculate probability distribution parameters. For example, the duration of a haul can be sensitive to the hauler’s nominal speed. Resource properties can also be used to select the most appropriate resources for a particular task. For example, bigger haulers may be preferred for longer hauls and smaller haulers for shorter hauls. In addition, properties are useful in matching different resources that must work together. For example, matching big loaders to big haulers and small loaders to small haulers.

Stroboscope exposes the properties of characterized resources through predefined system-maintained variables. These variables, combined with other facilities that will be described later, enable simulation models to use and recognize the properties of resources very effectively.

\subsection*{11.3.1 Accessing Property Values Collectively}

Characterized resources are often grouped into sets. For example, all the loaders currently in \textit{LoadersWait} form a set. All the steel shapes carried in instance number 28 of \textit{TransportSteelShapes} also form a set.

The properties of the resources in a set are accessed collectively through “aggregate” values. Aggregate values are such things as the sum, average, minimum, maximum, or standard deviation of a set of values.
Resources in Queues

The following global pre-defined variables access property values aggregated over the resources in a Characterized Queue (a Queue that holds characterized resources):

CharQueue.Property.AveVal
CharQueue.Property.MaxVal
CharQueue.Property.MinVal
CharQueue.Property.SDVal
CharQueue.Property.SumVal

In the variables above, CharQueue is the name of a Queue that holds characterized resources. Property is the name of a property belonging to the Queue’s resource type. The first three variables return the average, maximum, and minimum value of Property (respectively), considering all the resources in CharQueue. These variables cannot be used when CharQueue is empty because the average, maximum, and minimum are not defined in these cases. The SDVal variable returns the standard deviation of the values of Property, again considering all the resources in CharQueue. The SDVal variable requires that CharQueue contain at least two resources. Otherwise the standard deviation cannot be determined. The SumVal variable returns the sum of the values of Property, again considering all the resources in CharQueue. The SumVal variable can be used regardless of the current content of CharQueue.

The variable LoadersWait.BucketSize.MaxVal, for example, returns the bucket size of the largest loader currently in LoadersWait. The expression ‘SimTime-LoadersWait.TimeIn.MinVal’, returns the waiting time of the loader that has been longest in the Queue (and that is still in the Queue).

When a Characterized Queue contains exactly one resource, a property of the resource can be accessed through the following pre-defined system-maintained variable:

CharQueue.CharType.Property

In this case, CharType is redundant but necessary. It is redundant because the resource type of the Queue is known from the Queue definition. It is necessary so that it can be distinguished from the variable CharQueue.Property, which has a different meaning (to be discussed later). The variable LoadersWait.Loader.FuelLeft, for example, returns the amount of fuel in the tank of the loader held by LoadersWait. For this variable to be valid, LoadersWait must contain exactly one loader. Otherwise
Stroboscope will issue a runtime error. This is convenient because it warns against errors in logic.

If Property, in variables of the form CharQueue.CharType.Property, is a SaveProp, then the variable can be used as if it were a SaveValue (i.e., the value of Property can be updated). Thus, CharQueue.CharType.SaveProp can be used as an action target that accepts one action argument (an expression). The action changes the value of SaveProp for the resource in CharQueue. The following example illustrates this:

/ At this point LoadersWait must contain exactly one loader

In the above example, LoadersWait.Loader.FuelLeft is the action target and 'LoadersWait.Loader.TankCapacity' is the action argument. As soon as this statement is executed, the FuelLeft SaveProp of the only loader in LoadersWait will be assigned a value equal to its tank capacity.

The number of resources of a given SubType that exist in a Characterized Queue can be obtained through the following pre-defined system-maintained variable:

CharQueue.SubType.Count

Thus, the variable LoadersWait.S916.Count returns the number of S916 loaders currently in LoadersWait.

Resources in Activity Instances

Activity instances can hold any number of resources of different types and SubTypes. The following pre-defined instance variables access the properties of resources in Activity instances:

HeteroHolder.CharType.Property.AveVal
HeteroHolder.CharType.Property.MaxVal
HeteroHolder.CharType.Property.MinVal
HeteroHolder.CharType.Property.SDVal
HeteroHolder.CharType.Property.SumVal

These variables are similar to those used for Queues. What used to be the first part, CharQueue, has been replaced by HeteroHolder.CharType. HeteroHolder is the name of the Activity. HeteroHolder is used instead of Activity because these variables also apply to other nodes (to be discussed later). In general, a HeteroHolder is a node that can hold resources of several different resource types. Activities are the only
HeteroHolders discussed so far. CharType and Property are the characterized resource type and property of interest. The variables return the mean, maximum, minimum, standard deviation and sum of Property for all resources of type CharType held by the instance in context of HeteroHolder.

Note that these variables access Activity instance data. As such, they are only available when the Activity is in context (i.e., an instance of the Activity is starting or terminating). The instance data for Activity instances in the FEL are never accessible.

Assume, for example, that a certain Activity called TransportBeams represents the transportation of pre-fabricated beams on a flatbed. The beams and flatbed are characterized resources, and a single flatbed will usually carry more than one beam. The expected transport speed depends on the particular flatbed and on the load carried. Let us assume that flatbeds are represented by resource type FlatBed, and that their NominalSpeed property is a good indicator of speed. The expected speed is NominalSpeed corrected by a factor that depends on the weight of the load. The factor is given by the formula ‘(43000-LoadWeight)/30000’. Beams are represented by resource type Beam, and their Weight property indicates their weight. Some variable named CurDist indicates the current transport distance, which may be variable. Obviously, all these issues affect the duration of TransportBeams. Let us further assume that the actual transport time varies from the expected, with a 20% coefficient of variation. The duration for TransportBeams could then be specified with the following statements:

\[
\text{VARIABLE SpeedFactor} \ (43000-\text{TransportBeams.Beam.Weight.SumVal})/30000; \\
\text{VARIABLE ExpectedSpeed} \\
\quad \text{TransportBeams.FlatBed.NominalSpeed.SumVal*SpeedFactor;} \\
\text{DURATION TransportBeams CurDist/(ExptectedSpeed*Normal[1,0.2]);}
\]

SpeedFactor and ExpectedSpeed were defined to simplify the expression for the duration of TransportBeams. A single, long expression could have been used as well.

Notice that TransportBeams.FlatBed.NominalSpeed.SumVal was used to obtain the nominal speed of the flatbed. If the logic of the model ensures that TransportBeams holds exactly one flatbed, then “MinVal”, “MaxVal”, and “AveVal” could replace “SumVal”. The result would be the same. But also notice, that if there is an error in the logic, and 0 or more than 1 flatbeds are involved, each of the four possibilities will provide a different answer (all of them wrong). “SumVal” will always return a value,
even if the number of flatbeds is zero. “MinVal”, “MaxVal”, and “AveVal” will return a value as long as the number of flatbeds is one or more. If the number of flatbeds is zero, “MinVal”, “MaxVal”, and “AveVal” will produce a runtime error that points out the erroneous logic.

When the logic of model guarantees that exactly one resource of a given type is involved in an Activity instance, it is convenient to drop the aggregate selector (e.g., “MinVal”, “MaxVal”) and use the following variable:

\[ HeteroHolder.CharType.Property \]

This variable is convenient because it is shorter, (and) provides faster and more direct access to the information of interest, and issues runtime errors when the number of resources is not exactly one. Thus, it is better to use

\[ TransportBeams.FlatBed.NominalSpeed \]

in the definition of ExpectedSpeed.

As with Queues, if Property in HeteroHolder.CharType.Property is a SaveProp, the variable can be used as if it were a SaveValue. Thus, the value of Property can be updated as shown below:

\[ \begin{align*}
\text{NORMAL FillUpTank;} \\
\text{BEFOREEND FillUpTank ASSIGN} \\
\quad \text{FillUpTank.Loader.FuelLeft FillUpTank.Loader.TankCapacity;}
\end{align*} \]

It is also possible to obtain a count of the number of resources of a specific SubType held by an Activity instance through the following variable:

\[ HeteroHolder.SubType.Count \]

Thus, if one of the SubTypes of Beam is TBeam, the variable

\[ TransportBeams.TBeam.Count \]

returns the number of T-Beams in the instance in context of TransportBeams.
11.4 The Cursored Resource

Figure 22 shows how characterized resources are arranged in a Queue called LoadersWait at some point during simulation. LoadersWait currently holds 10 resources of type Loader (as defined in previous examples), 6 of them are S926Es and 4 are S916s. The current simulation time is 135, and many other loaders exist in the system (but only 10 are in LoadersWait).

Each tablet represents a loader. The slots in the loaders show the values of the different properties. The slot labeled “Pos In Queue” is the only one that does not represent a property. “Pos In Queue” indicates the position of the loader within LoadersWait. The loader in position 1 is the first one in line. The loader in position 10 is at the end of the line.
The loader in position 1 is an S916, as indicated by its *BucketSize*, *TankCapacity*, and *Power* “SubType Properties”. It was created at simulation time 22.7. Seventeen other loaders had been created earlier, hence this is loader number 18. The loader entered *LoadersWait* at simulation time 100.7 — it has been waiting for 135-100.7=34.3 time units. The loader is empty (*AmountLoaded* is 0) and its tank contains 17 liters of fuel.

All the information in the above paragraph is specific to the loader in position 1. It was possible to describe this information because the focus was on that specific loader. If the position were not specified, it would not have been possible to discuss these details.

Stroboscope modeling elements such as Queues set their focus on a specific resource through their “cursor”. The cursor is an indicator that can point to a specific resource. Sometimes the cursor is not pointing anywhere. When the cursor is pointing at a resource, the resource is said to be “cursored” by the modeling element, or the modeling element is said to “cursor” the resource. Each modeling element that “can cursor” has its own cursor. The *CursoringElement.HasCursor* pre-defined variable can be examined to determine if a cursoring modeling element is indeed cursoring. The variable returns TRUE when the element is cursoring and FALSE otherwise.

Modeling elements cursor resources to perform many tasks, such as to establish Queue disciplines and to filter resources that meet a given condition. The person writing a simulation model has no direct control over a modeling element’s cursor. The cursoring process, however, must be understood in order to work with characterized resources effectively.

The properties of a cursored resource are accessible explicitly via pre-defined variables of the form:

*ModelingElement.Property*

For example, the variable *LoadersWait.FuelLeft* returns the value of the *FuelLeft* property of the loader cursored by *LoadersWait*. This variable is only valid when *LoadersWait* is cursoring a loader. If the variable is used at a time at which *LoadersWait* is not cursoring, Stroboscope will issue a runtime error.

Figure 23 shows *LoadersWait* cursoring the resource at position 9.
While loader # 9 is cursored as shown in Figure 23, \( \text{LoadersWait} \text{.ResNum} \) returns 31, \( \text{LoadersWait} \text{.BirthTime} \) returns 26.2, \( \text{LoadersWait} \text{.TimeIn} \) returns 127.7, \( \text{LoadersWait} \text{.BucketSize} \) returns 1.5, \( \text{LoadersWait} \text{.TankCapacity} \) returns 126, \( \text{LoadersWait} \text{.Power} \) returns 73, \( \text{LoadersWait} \text{.FuelLeft} \) returns 60, and \( \text{LoadersWait} \text{.AmountLoaded} \) returns 0.

Most of the time, the property of a cursored resource is used in a context where Stroboscope expects a “cursored expression” (i.e., an expression that contains references to a property of the cursored resource). For example, the DrawDuration expression for a Characterized Draw Link may be expressed in terms of a property of the resource drawn. In order to make the properties of the drawn resource available, Stroboscope points the link’s cursor to the resource being drawn (while the DrawDuration expression is being evaluated). This makes it possible, for example, to specify the DrawDuration attribute for a loader Draw Link as follows:
$L1$'s DrawDuration can be set more conveniently as follows:

```
/ expected loader set up time is function of Bucket Size
DRAWDURATION L1 0.28*L1.BucketSize^2/9*Normal[1,0.20];
```

The difference between the two expressions is that $BucketSize$ is used instead of the longer form $L1.BucketSize$. Thus, $BucketSize$ becomes a variable in its own right when used directly in the expression for a “cursored attribute”.

The shorthand form of cursored property access is only valid in the statement that defines the cursored attribute. Hence the following statements are invalid and not equivalent to the two previous statements:

```
VARIABLE ExpectedLoaderDrawDuration 0.28*BucketSize^2/9; /Error !
DRAWDURATION L1 ExpectedLoaderDrawDuration*Normal[1,0.20];
```

$BucketSize$ as used in the definition of $ExpectedLoaderDrawDuration$ is not legal, even though it is used indirectly for the DrawDuration attribute. This is because $ExpectedLoaderDrawDuration$ is global in scope. When Stroboscope parses the expression, it does not know whether a loader will be cursored, or which element will do the cursoring. If the expression is used to change a cursored attribute directly, on the other hand, Stroboscope knows which element is cursoring and the resource type of the cursored resource. Thus, if the cursored resource is accessed indirectly, the long form of cursored access must be used:
/ expected loader set up time is function of Bucket Size
VARIABLE ExpectedLoaderDrawDuration 0.28*L1.BucketSize^2/9; /OK !
DRAWDURATION L1 ExpectedLoaderDrawDuration*Normal[1,0.20];

In this case, care must be taken not to use ExpectedLoaderDrawDuration in situations where link L1 is not cursoring.

The preferred method of simplifying cursored expressions, however, is through VarProps. The following example defines ExpectedLoaderDrawDuration as a VarProp:

VARPROP Loader ExpectedLoaderDrawDuration 0.28*BucketSize^2/9;
DRAWDURATION L1 ExpectedLoaderDrawDuration*Normal[1,0.20];

The example above has the added benefit that ExpectedLoaderDrawDuration can be used when a loader is cursored by any node (not just by L1).

In addition to the DrawAmount attribute of Characterized Draw Links, there are other cursored attributes and action events that are relevant given our current understanding of the Stroboscope simulation logic (i.e., given the issues discussed so far).

The Strength attribute of a characterized Dynafork branch (a link that leaves a Dynafork) is a cursored attribute. When the attribute is evaluated, the predecessor Dynafork is cursoring the characterized resource that will be routed. Thus, the properties of the resource that will be routed are available to the Strength of the branches that leave the Dynafork. Although the cursoring is done by the Dynafork, and Strength is an attribute of the Branch, the properties of the cursored resource can be accessed implicitly.

The following code illustrates cursored access in Dynafork branches:

```
ROUTE To WaitToFillUp if tank content is less than 20% of capacity
STRENGTH L3 FuelLeft/TankCapacity>=0.20;
STRENGTH L4 RouteToReFuel.FuelLeft/RouteToReFuel.TankCapacity<0.20;
```
The Strengths used for the above Dynafork branches depend on the amount of fuel left in the loader and on the loader’s fuel capacity. While the Strengths are evaluated, RouteToReFuel cursors the loader being routed. Although the Strengths are attributes of L3 and L4 (i.e., not attributes of RouteToReFuel), the short form of the cursored property access can be used. The short form is used in the Strength of L3, while the long form is used in the Strength of L4. Note that when the long form is used, the cursoring object is the Dynafork and not the Branch.

While actions are performed during the OnDraw, OnFlow, and OnRelease action events, characterized links cursor the resource that is flowing. The short form of cursored property access is available in expressions used as arguments for the action definition. The SaveProps of the cursored resource can additionally be used as action targets as if they were SaveValues. The following code uses both:

```plaintext
COLLECTOR FuelConsumed;

/keep track of fuel used between Fill-Ups
ONRELEASE L6 COLLECT FuelConsumed TankCapacity-FuelLeft;

/mark loader as filled-up
ONRELEASE L6 ASSIGN FuelLeft TankCapacity;
```

Similarly, while actions are performed during the OnEntry action event, a Characterized Queue cursors the resource that is entering. The short form of cursored property access and the use of SaveProps as SaveValues are available here as well.

### 11.5 A Simple Example - Scraper Operations

Stroboscope treats characterized resources and generic resources differently. The advantage of generic resources is that they can represent bulk quantities of any real amount. The advantage of characterized resources is that they carry information and that they are uniquely identifiable.

Stroboscope “simulates” the movement of generic resources using arithmetic. When a generic resource is drawn from a Queue to a Combi, Stroboscope subtracts from
the content of the Queue and adds to the content of the Combi instance. When a Generic Release Link releases resources it simply adds to the content of the successor without even subtracting from the predecessor.

While generic resources are indistinguishable and interchangeable, characterized resources are not. Characterized resources are real. They actually move from node to node carrying all the information attached to them.

The unique nature of characterized resources introduces many new capabilities that require somewhat different and additional concepts for their exploitation. These new capabilities and concepts are the subject of the next chapter.

In simple cases that do not involve complex resource drawing and releasing schemes, the use of characterized resources is similar to the use of generic resources. The advantage is that the properties of resources can be recognized, that the resources can carry information, and that substantially more detailed statistics are available.

The example that follows is simple in that it does not require complex resource drawing and releasing schemes. The example, however, illustrates how sophisticated construction engineering computations can be incorporated into simulations.

### 11.5.1 The Example

An earth-moving equipment fleet is composed of pusher tractors and two types of scrapers. These load and haul earth for the construction of a 4 km road segment. Haul distance is variable because as work progresses the scrapers need to dump the hauled earth further along the constructed segment.

For a particular scraper-pusher combination, the optimum loading time and corresponding optimum pay load depend on the current haul distance, on the scraper’s speed, and on the scraper’s load-growth curve (Nunnally 1977). In order to maximize production, the scraper operator attempts (but does not necessarily achieve) to load the optimum amount by attempting to control the loading time. Scraper speed depends on gross weight, which depends on load, and is thus also variable. In a highly non-linear situation such as this, simulation can be used to determine the optimal fleet composition.
Figure 24 shows the Stroboscope network for this example. The network shows that pushers push-load scrapers, backtrack, and then wait to push-load another scraper; scrapers are push-loaded by pushers, haul their load, dump it, return, and then wait to be push-loaded again; earth is created as result of the scraping action and carried implicitly by the scraper, it is hauled in the scraper, and explicitly dumped to become part of the road.

Figure 25 shows data regarding the earth-moving process. This data will be used to build the simulation source file for the example. The formulas shown for the optimum load time and optimum pay load were obtained empirically.

Decision variables and global problem parameters are defined towards the top of the source file:

```plaintext
/**Problem parameters and decision variables
VARIABLE NumPshrs 3; /number of pushers
VARIABLE Num651E 2; /number of big scrapers
VARIABLE Num621E 9; /number of small scrapers
VARIABLE PshrInitCost 900; /pusher initial cost ($)
VARIABLE PshrHrCost 55; /pusher hourly cost ($/hr)
VARIABLE OthrHrCost 200; /other hourly cost ($/hr)
VARIABLE EarthWgt 15.7; /weight of earth (kN/m3)
VARIABLE ShrkFct 0.95; /earth shrinkage factor
VARIABLE InitDst 1000; /initial haul distance (M)
VARIABLE FinDist 5000; /final haul distance (M)
VARIABLE RdCrsSct 12.5; /road cross sectional area (M2)
VARIABLE RollRst 0.03; /rolling resistance
VARIABLE Grade 0.02; /hauling slope (uphill)
```

Copyright © 1996 Julio C. Martínez
The next part of the source file is the definition of the resource types:

/**Resource Types

GENTYPE Earth; /ER
GENTYPE Pusher; /PS

CHARTYPE Scraper Wgt Pow Cap MaxV Trans Eff InitCost Hourly Cost
/ kN kW m3 km/hr $/hr $
SUBTYPE Scraper S621E 299 246 10.7 51 83% 750 32
SUBTYPE Scraper S651E 583 410 24.5 55 81% 1300 105
SAVEPROPS Scraper AmountLoaded;

Pushers are modeled as generic resources because they are all the same and their properties are already reflected in other data. Scraper’s AmountLoaded SaveProp will hold the amount of earth they carry.

The next part of the input file is the definition of the network itself. That part is not included here since it can be inferred from the network drawing. Note the

---

Figure 25 - Data for Earth-Moving Operation With Scrapers
superscripts in links \textit{ER1} and \textit{SC4}. They indicate that link \textit{ER1} should be defined before link \textit{SC4}.

Global variables and scraper VarProps to aid in the definition of element attributes are defined after the definition of the network:

/**Auxiliary global Variables and scraper VarProps

VARIABLE DstPerBV 'ShrkFct/RdCrsSct'; / (m/m²)
VARIABLE Distance 'InitDst+DstPerBV*MvdEarth.CurCount'; / (m)
VARPROP Scraper OptLdTm '125*(1+0.48*Ln[0.08*Distance/Pow])';
VARPROP Scraper OptPayLd 'Cap*(1+Cap/60*Ln[Distance/5000])';

\textit{DstPerBV} is the rate of road advance per bank m³ of earth placed. It depends on the shrinkage factor of the earth and on the road’s cross sectional area. \textit{Distance} is the current travel distance. As \textit{MvdEarth} receives earth, travel distance increases.

\textit{OptLdTm} and \textit{OptPayLd} are scraper VarProps that represent the optimum loading time and pay load for a scraper. They are defined in terms of other scraper properties and on the current haul distance.

The next part of the input file defines modeling element attributes, along with any necessary Variables, SaveValues, or VarProps that aid in the definition of the attributes:

/**Attributes of Load and related links

VARPROP Scraper ActLdTm 'OptLdTm*Pertpg[0.95,1,1.1]';
VARIABLE SpotTime 'Pertpg[24,36,95]';
DURATION Load 'SpotTime+Load.Scraper.ActLdTm';
ONRELEASE SC2 ASSIGN AmountLoaded 'OptPayLd*Max[Normal[1,0.3],0.25]';

The duration of \textit{Load} is composed of two parts, the time to spot (\textit{SpotTime}) and the actual loading time (\textit{ActLdTm}). \textit{ActLdTm} is a scraper VarProp because it applies exclusively to a scraper and depends on another scraper property (\textit{OptLdTm}). When a scraper is released through \textit{SC2}, the amount of earth loaded is sampled and stored in the scraper’s \textit{AmountLoaded} SaveProp.

/**Attributes of BkTrack and related links

DURATION BkTrack '15+0.40*Haul.Scraper.OptLdTm';

The duration of \textit{BkTrack} includes the time during which pushers transfer (15 seconds) and then backtrack (40% of the optimum loading time). Note that the optimum
loading time is accessed through the scraper, which is already in the starting instance of
Haul. This is because the duration attribute of a starting Normal is not evaluated until all
the resources in the terminating predecessor instance have been released.

/**Attributes of Haul and related links

VARPROP Scraper GrossWgt 'Wgt+AmountLoaded*EarthWgt';
VARPROP Scraper HaulFrc 'GrossWgt*(Grade+RollRst)';
VARPROP Scraper ThHaulV 'Min[Pow*Eff/HaulFrc,MaxV/3.6]';
VARPROP Scraper ThHaulTm 'Distance/ThHaulV';
DURATION Haul 'Haul.Scraper.ThHaulTm*Normal[1,0.25]';

Since haul time depends mostly on scraper properties and distance, the
computations are simplified with scraper VarProps. GrossWgt is the weight of the scraper
and the earth it contains. HaulFrc is the force required to move the scraper. ThHaulV is
the theoretical hauling speed; it depends on the scraper’s power, transmission efficiency,
required hauling force, and maximum speed. ThHaulTm is the theoretical hauling time; it
depends on travel distance and speed. Finally, the actual haul time varies from the
theoretical with a 25% coefficient of variation.

/**Attributes of Dump and related links

DURATION Dump 'Pertpg[24,36,78]';
RELEASEAMT ER1 'Dump.Scraper.AmountLoaded';

The earth was implicitly carried by the scraper. Since no earth actually enters
Dump, the amount of earth released must be specified explicitly. The amount is
determined from the AmountLoaded SaveProp of the scraper. Note that when the earth is
released the scraper is still in the terminating instance of Dump. This is because link ER1
is defined before link SC4. Had the links been defined in the reverse order, the scraper
would have been in the starting instance of Return and not in the terminating instance of
Dump. When instances terminate, characterized resources do not behave like generic
resources. Characterized resources actually move from one node to the other, whereas
generic resources are created on release (leaving the amount of generic resource in the
terminating instance intact).

/**Attributes of Return and related links

VARPROP Scraper RetFrc Wgt*(RollRst-Grade);
VARPROP Scraper ThRetV 'Min[Pow*Eff/RetFrc,MaxV/3.6]';
VARPROP Scraper ThRetTm 'Distance/ThRetV';
DURATION Return 'Return.Scraper.ThRetTm*Normal[1,0.15]';

The computations that determine the return time of a scraper are very similar to the computations of the haul time. In this case the scraper is empty and grade is favorable. RetFrc is the force required to move the scraper as it returns. ThRetV is the theoretical return speed. ThRetTm is the theoretical return time. The actual duration varies from the theoretical with a 15% coefficient of variation.

At this point, the model is completely defined. The Queues can be initialized and the simulation run:

/**Initialize Queues
INIT Pshrs NumPshrs;
INIT Scrprs Num651E S651E;
INIT Scrprs Num621E S621E;
/**Simulate until entire road segment is built
SIMULATEUNTIL 'Distance>=FinDist';

The simulation will stop when the current travel distance is at least as large as the final distance (the road segment has been built).

After the simulation is done, the data of interest needs to be extracted from the simulation results:

VARIABLE TotSetupCst 'NumPshrs*PshrInitCost+Num621E*S621E.InitCost+
                      Num651E*S651E.InitCost';
VARIABLE TotHrs     'SimTime/3600';
VARIABLE TotHrCst   'NumPshrs*PshrHrCost+Num621E*S621E.HrCost+
                      Num651E*S651E.HrCost+OthrHrCost';
VARIABLE TotalCst   'TotSetupCst+TotHrs*TotHrCst';
VARIABLE Product    'MvdEarth.CurCount/TotHrs';
VARIABLE UnitCost   'TotalCst/MvdEarth.CurCount';

TotSetupCst is the total setup cost. TotHrs converts the elapsed simulation time from seconds to hours. TotHrCst is the total hourly cost. TotalCst is the total cost incurred. Product is the productivity of the process in bm$^3$/hr. UnitCost is the unit cost in $/bm^3$.

Finally, the results of interest are presented:

PRINT StdOutput "Number of pushers : %7.0f\n" NumPshrs;
When the above source input file is run, Stroboscope produces an output similar to the following:

Stroboscope Model SCRAPERS.STR (888494496)

Number of pushers : 3
Number of 651E scrapers : 2
Number of 621E scrapers : 9
Total earth moved : 52642.09 bm3
Total time required : 103.88 hours
Total cost of road segment : 116242.39 $
Production rate : 506.76 bm3/hr
Unit cost : 2.21 $/bm3
Av. pusher wait : 12.41 secs
Av. scraper wait : 117.34 secs

Execution Time = 6.078 seconds
Processor Time = 5.95313 seconds

A quick look at the results indicate that the fleet is not well balanced. Scrapers wait on average 2 minutes, while pushers wait an average of barely 12 seconds. Other fleet configurations should be tried, perhaps with more pushers or fewer scrapers.
11.6 Characterized Resource Statistics

Characterized resources are discrete entities. Each resource has a life that spans from the time it is created to the time it is destroyed. Characterized resources are created when Queues are initialized. They are also created by acting on the GENERATE pre-defined action target (to be discussed in the next chapter) during the BEFOREDRAWS or BEFOREEND action events. Characterized resources are destroyed when unreleased by terminating Activity instances. Throughout its lifetime, a characterized resource visits the nodes on a network any number of times.

Stroboscope keeps various statistics about the lifetime and visiting experiences of characterized resources on a Type and SubType basis. These statistics are available through pre-defined system-maintained variables. They are also displayed in the standard report (produced by the REPORT statement).

Visit statistics

Characterized resources are analogous to customers, and network nodes to hotels. A customer visits a hotel, spends some time in it, and leaves towards another hotel. The customer may later come back to a previously visited hotel, and spend some more time there (although not necessarily the same time as before).

Stroboscope keeps discrete statistics on resource visit times. Every node in a Stroboscope network contains one discrete statistics collector for each SubType that can visit the node. Scprvs, for example, has one collector that keeps track of the visit times of S621E scrapers and another collector that keeps track of the visit times of S651E scrapers. An Activity has one collector for each of the SubTypes of each characterized type that can enter the Activity. The visit collectors in a node sample visit times when resources leave the node. As a consequence, they do not consider the time spent by resources in their current nodes (their visit time is not known until they actually leave); they only consider complete visits. The visit statistics for a run of the scraper earth-moving example of the previous section, as shown on the standard report, appears below:
<table>
<thead>
<tr>
<th>Customer</th>
<th>Hotel</th>
<th>AveEnts</th>
<th>AvTotTm</th>
<th>AvVstTm</th>
<th>SDAvVst</th>
<th>MnVstTm</th>
<th>MxVstTm</th>
</tr>
</thead>
<tbody>
<tr>
<td>S621E</td>
<td>Dump</td>
<td>442.00</td>
<td>20293.12</td>
<td>45.91</td>
<td>16.80</td>
<td>17.05</td>
<td>106.24</td>
</tr>
<tr>
<td>S621E</td>
<td>Haul</td>
<td>442.00</td>
<td>139445.63</td>
<td>315.49</td>
<td>164.03</td>
<td>74.24</td>
<td>287.62</td>
</tr>
<tr>
<td>S621E</td>
<td>Load</td>
<td>442.33</td>
<td>73404.64</td>
<td>165.95</td>
<td>35.51</td>
<td>74.24</td>
<td>287.62</td>
</tr>
<tr>
<td>S621E</td>
<td>Return</td>
<td>441.67</td>
<td>89175.41</td>
<td>201.91</td>
<td>88.44</td>
<td>38.11</td>
<td>495.73</td>
</tr>
<tr>
<td>S621E</td>
<td>Scrprs</td>
<td>442.56</td>
<td>51540.79</td>
<td>116.46</td>
<td>82.07</td>
<td>0.00</td>
<td>437.45</td>
</tr>
<tr>
<td>S651E</td>
<td>Dump</td>
<td>438.00</td>
<td>20344.87</td>
<td>46.45</td>
<td>17.33</td>
<td>17.19</td>
<td>102.20</td>
</tr>
<tr>
<td>S651E</td>
<td>Haul</td>
<td>438.00</td>
<td>158991.87</td>
<td>363.00</td>
<td>203.79</td>
<td>48.55</td>
<td>1043.76</td>
</tr>
<tr>
<td>S651E</td>
<td>Load</td>
<td>439.00</td>
<td>59667.05</td>
<td>135.92</td>
<td>36.19</td>
<td>51.69</td>
<td>237.41</td>
</tr>
<tr>
<td>S651E</td>
<td>Return</td>
<td>438.00</td>
<td>81713.72</td>
<td>185.33</td>
<td>79.75</td>
<td>51.21</td>
<td>450.04</td>
</tr>
<tr>
<td>S651E</td>
<td>Scrprs</td>
<td>439.00</td>
<td>53302.21</td>
<td>121.42</td>
<td>82.56</td>
<td>0.00</td>
<td>319.69</td>
</tr>
</tbody>
</table>

Each row corresponds to the statistics kept by each visit collector. The “Customer” column contains the SubType. The “Hotel” column contains the node. The first row, for example, contains statistics on the visit of S621E scrapers to Dump.

The number of samples collected by each collector is not shown on the table. Instead, the table shows the values listed under the “AveEnts” column. The value under “AveEnts” is the number of samples normalized through division by the total population of resources of the SubType. The total population is the total number of resources of the SubType that were created during simulation, including those that have already been destroyed. The 442.00 in the first row, for example, is the number of samples collected by Dump’s S621E visit collector (3978) divided by the total population of S621E scrapers in this run (9). This value can be interpreted as the average number of dumps performed by an S621E scraper.

The “AvTotTm” column contains the sum of the time of all the visits normalized by the total population. The value should be interpreted as the total time spent inside the node by the average resource of the SubType. The 20,293.12 in the first column, for example, is the sum of the values collected by Dump’s S621E visit collector (182638.08) divided by the total population of S621E scrapers (9). This means that, on average, an S621E scraper spent 20,293.12 seconds dumping.

The values in the remaining columns, “AvVstTm”, “SDAvVst”, “MnVstTm”, and “MxVstTm”, are not normalized. These values are simply the average, standard deviation, minimum, and maximum of the values collected by the corresponding collector. The values on row one indicate that the average dumping time for S621E scrapers was 45.91 seconds with a standard deviation of 16.80 seconds; that the fastest
dump by any 621E scraper took 17.05 seconds; and that the slowest one took 106.24 seconds.

Note that the above statistics are collective over all the resources of a SubType. Hence it is unlikely, although possible, for the same resource to experience the fast dumping time of 17.05 seconds and the slow dumping time of 106.24 seconds. For the same reason, the statistics in the first row do not provide the average visit time and standard deviation of visit time for a specific 621E scraper. The statistics apply to 621E scrapers collectively.

Visit statistics are accessible through pre-defined variables of the following form:

Hotel.Customer.{AveEnts|AvTotTm|AvVstTm|SDAvVst|MnVstTm|MxVstTm}

Hotel is the name of the node and Customer is the name of the SubType. The last part must be one of AveEnts, AvTotTm, AvVstTm, SDAvVst, MnVstTm, or MxVstTm. The pre-defined variables return values that correspond to the columns in the report. As with all discrete statistics, access to the average, minimum, or maximum visit time requires that at least one visit be completed. Access to the standard deviation requires the completion of at least two visits.

**Life-span Statistics**

Stroboscope creates characterized resources when it initializes Queues. Stroboscope can also create characterized resources during the BEFOREDRAWS or BEFOREEND action events (this will be discussed in the next chapter). When Activity instances terminate, they destroy any unreleased characterized resources.

The simulation time that elapses between the creation (BirthTime) and destruction of a resource is the lifetime of the resource. At any point during the processing of a simulation model, the population of characterized resources in the system consists of those resources that are “alive”. Stroboscope keeps statistics on the lifetime and population of characterized resources on a Type and SubType basis.

These statistics are particularly interesting for resources that enter the system (they are born) and then leave (die). The life-span of these resources differs from the duration of the simulation. Lifetime and population statistics provide little information for resources that live throughout the entire simulation.
Population statistics are continuous and weighted by time. Stroboscope uses one time-weighted collector for each SubType and for each characterized resource type in the system. The current value of each of the time-weighted collectors is given by the corresponding current population. These statistics are always up to date — their total weight equals SimTime all the time. The following pre-defined variables provide access to population statistics:

\[
\text{CharType} \cdot \{\text{AvePp} | \text{CurPp} | \text{MaxPp} | \text{MinPp} | \text{SDPp} | \text{TotPp}\} \\
\text{SubType} \cdot \{\text{AvePp} | \text{CurPp} | \text{MaxPp} | \text{MinPp} | \text{SDPp} | \text{TotPp}\}
\]

The first form provides statistics on the population of resources of type \text{CharType}. The second form provides statistics on the population of resources of \text{SubType} \text{SubType}. In both forms, the second part must be one of \text{AvePp}, \text{CurPp}, \text{MaxPp}, \text{MinPp}, \text{SDPp} or \text{TotPp}. \text{AvePp} returns the average population. \text{CurPp} returns the current population. \text{MaxPp}, \text{MinPp} and \text{SDPp} return the maximum, minimum, and standard deviation of the population. \text{TotPp} returns the total population (i.e., the total number of resources that have been created, including those that have already been destroyed).

The example of the previous section used two \text{S621E} and nine \text{S651E} scrapers. These scrapers lived throughout the entire simulation. Hence their population was fixed and constant (this case falls into the uninteresting category). The population statistics pre-defined variables had the following values during the entire run:

\[
\begin{align*}
\text{Scraper.AvePp} &= 11 & \text{S621E.AvePp} &= 9 & \text{S651E.AvePp} &= 2 \\
\text{Scraper.CurPp} &= 11 & \text{S621E.CurPp} &= 9 & \text{S651E.CurPp} &= 2 \\
\text{Scraper.MaxPp} &= 11 & \text{S621E.MaxPp} &= 9 & \text{S651E.MaxPp} &= 2 \\
\text{Scraper.MinPp} &= 11 & \text{S621E.MinPp} &= 9 & \text{S651E.MinPp} &= 2 \\
\text{Scraper.SDPp} &= 0 & \text{S621E.SDPp} &= 0 & \text{S651E.SDPp} &= 0 \\
\text{Scraper.TotPp} &= 11 & \text{S621E.TotPp} &= 9 & \text{S651E.TotPp} &= 2
\end{align*}
\]

Stroboscope starts tracking population statistics just before it executes the first \text{SIMULATE} or \text{SIMULATEUNTIL} statement (i.e., after the \text{INIT} statements have created resources). Otherwise the minimum population would always be zero. The standard deviation of the population in all the cases above was zero — there was no variability in the scraper population. Notice that the statistics for the characterized type (e.g., \text{Scraper}), encompass the statistics for its SubTypes (e.g., \text{S621E} and \text{S651E}).

Life-span statistics are discrete. Stroboscope considers in its statistics only those resources that have been destroyed — the life of resources that are still in the system.
does not count. Life-span statistics are available through pre-defined variables of the following form:

CharType.{AveLife|MaxLife|MinLife|SDLife}
SubType.{AveLife|MaxLife|MinLife|SDLife}

The first form provides statistics on the life-span of resources of type CharType. The second form provides statistics on the life-span of resources of SubType SubType. In both forms, the second part must be one of AveLife, MaxLife, MinLife, or SDLife. The variables return the average, maximum, minimum, and standard deviation of the life-span. Since these statistics are discrete, at least one resource must have been destroyed before AveLife, MaxLife, or MinLife are used; at least two must have been destroyed before SDLife is used. No scrapers are destroyed in the example of the previous section, and thus none of these variables are available for that run.

11.7 Recap

- Characterized resources of the same type share common traits.
- Resources are characterized by their properties.
- The type of a resource determines which properties are borne by the resource, but not the specific values of the properties.
- Resources of the same characterized type have the same properties, but the values assigned to these properties may vary among the individual resources of the type.
- Characterized resources must belong to a SubType of their resource type.
- SubType Properties are fixed and have common values for all resources of the same SubType.
- SaveProps are properties whose value can be changed through action events. Every resource has its own copy of a SaveProp.
- Stroboscope pre-defines and maintains the BirthTime, ResNum, and TimeIn property for all resources. BirthTime is the time at which the resource was created.
ResNum is the individual ID of the resource within its type. TimeIn is the time at which the resource entered the last node it visited.

- VarProps are properties defined in terms of other properties and global symbols. The expression used to define a VarProp is recalculated every time the property is accessed.

- Resources are created and placed in Queues by the execution of INIT control statements.

- Stroboscope destroys a resource when the Activity instance that holds it terminates and does not release it.

- A property can be aggregated over all the resources in a Queue or Activity instance. Statistics on the aggregated properties, such as count, average, standard deviation, maximum, or minimum, can then be accessed through pre-defined system-maintained variables.

- A property does not need to be aggregated when exactly one resource is in a Queue or Activity instance. In these cases, if the property is a SaveProp, it can be the target of an ASSIGN action.

- Modeling elements set their focus on a particular resource through their cursor. The resource on which the modeling element is focused is the cursored resource.

- Modeling elements cursor resources for many purposes. When a resource is cursored, a property of the cursored resource can be accessed explicitly with variables of the form `CursoringElement.Property`. In expressions where Stroboscope expects access to a cursored resource, a property of the resource can be accessed without preceding it with the modeling element’s name.

- Stroboscope “simulates” the movement of generic resources using arithmetic. Characterized resources, on the other hand, are real. They actually move from node to node carrying all the information attached to them.
• Stroboscope collects statistics on the number of times resources enter nodes, and on the time they spend in those nodes. This is done on a SubType basis. These statistics are discrete and consider only complete visits.

• Stroboscope maintains continuous statistics, weighted by simulation time, on the population of resources. These statistics are kept on a Type and SubType basis.

• Stroboscope maintains statistics on the life-span of resources. These are discrete statistics, kept on a Type and SubType basis, that consider only those resources that have already been destroyed.
Chapter 12

Drawing, Releasing, Generating, and Filtering
Characterized Resources

This chapter introduces the tools necessary for detailed control of characterized resources. The first part of this chapter is about selective drawing, releasing, and generation of characterized resources. The second part is about Filters, which enable access to resource properties aggregated over very precisely defined sets.

12.1 Controlling Characterized Resource Flow

Stroboscope “simulates” the movement of generic resources using arithmetic. When a generic resource is drawn from a Queue to a Combi, Stroboscope subtracts from the content of the Queue and adds to the content of the Activity instance. When a Generic Release Link releases resources it simply adds to the content of the successor without even subtracting from the predecessor.

While generic resources are indistinguishable and interchangeable, characterized resources are not. Characterized resources are real. They actually move from node to node carrying all the information attached to them.

The unique nature of characterized resources introduces many new capabilities that require somewhat different and additional concepts for their exploitation. The first of the following subsections deals with the ordering of resources within Queues. The second subsection deals with the acquisition of characterized resources by Combis. The third
subsection is about the release of characterized resources during the termination of Activity instances. The fourth subsection is about the generation of characterized resources while the simulation is running.

12.1.1 Queue Disciplines

By default, when characterized resources enter a Queue they are placed at the back of the Queue (i.e., they go to end of the line). Also by default, the next resource removed from a Queue is the one at the front (i.e., the first one in line). Thus, the default discipline for Queues is FIFO (i.e., First In First Out).

Often, the discipline for a Queue is not FIFO. It is possible to specify a different discipline with the DISCIPLINE statement:

Syntax: DISCIPLINE CharQueue CursoredExpression;

Example: DISCIPLINE LoadersWait ‘SimTime<160 ? TimeIn : FuelLeft-Power’;

The discipline expression determines the position of an entering resource. The positions of the resources already in the Queue do not change, except that some resources may move back by one position in order to make room for the entering resource.

When a characterized resource enters a Queue, Stroboscope sets the resource’s TimeIn to SimTime. As a next step, the Queue cursors the entering resource and evaluates the Discipline expression. The entering resource is “tagged” with the result. The Queue then cursors the resource at the front of the Queue and evaluates the Discipline expression again. Because a different resource is cursored, the result can be different (unless the Discipline expression is not dependent on resource properties). If the result is strictly greater than the tag of the entering resource, the entering resource goes to the front of the Queue. Otherwise, the Queue cursors the resource in position 2 and reevaluates the Discipline expression. If the result is strictly greater than the tag of the entering resource, the entering resource is inserted between the resources in positions 1 and 2 (resources that previously occupied position 2 and greater now occupy positions 3 and greater). Otherwise, the process continues until the result of the Discipline for the cursored resource exceeds the entering resource’s tag. If this never happens, the entering resource goes to the end of the Queue.
The example sets `LoadersWait`’s discipline to ‘SimTime<160 ? TimeIn : FuelLeft-Power’. This discipline is somewhat arbitrary but it is a good example because it is dynamic and not simply the value of a property.

Let us assume that the current simulation time is 150 and that the resources in `LoadersWait` occupy positions as shown in Figure 22 on page 177. The loader in Figure 26 enters `LoadersWait` and is about to be positioned:

<table>
<thead>
<tr>
<th>Pos In Queue</th>
<th>ResNum</th>
<th>Birth Time</th>
<th>TimeIn</th>
<th>Bucket Size</th>
<th>Tank Capacity</th>
<th>Power</th>
<th>Fuel Left</th>
<th>Amount Loaded</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>72</td>
<td>80.3</td>
<td>150</td>
<td>1.5</td>
<td>126</td>
<td>73</td>
<td>94</td>
<td>0</td>
</tr>
</tbody>
</table>

*Figure 26 - First Loader to Enter LoadersWait*

`LoadersWait` cursors the resource and evaluates the discipline, which returns 150 because SimTime is less than 160. Thus the tag of the entering resource is 150.

`LoadersWait` cursors every loader it contains, one by one starting at the front, and reevaluates the discipline every time. It goes through all the resources in search of the first one that will yield a discipline greater than 150. It finds none, and thus places the entering loader at the end of the Queue in position 11. Figure 27 shows the content of `LoadersWait` at this point.

Let us assume that without an advance in simulation time, the loader in Figure 28 arrives. The procedure to determine its position is the same as in the previous case. This time, however, the entering loader’s tag matches the result of the discipline when the loader at position 11 is cursored. Since the tag of the entering loader must be strictly smaller (not smaller or equal), the entering loader is not inserted in position 11. Thus, the entering loader goes to the end of the line. Figure 29 shows the content of `LoadersWait` after the second loader arrives.

An inspection of the loader ordering shows that it is consistent with the discipline, which up to now has been essentially ‘TimeIn’ (since SimTime has never been greater than 160). Notice also, that the Queue discipline has so far been strictly FIFO. Every new loader that arrives goes to the end of the Queue.
Using ‘TimeIn’ or any constant expression (i.e., an expression that returns the same number regardless of which resource is cursored) as discipline results in FIFO, which is the default. In practice, it is not wise to explicitly specify a Discipline to achieve FIFO. This is because when the default Discipline has not been changed through a DISCIPLINE statement, Stroboscope places new resources at the end of Queues without evaluating any expressions or performing comparisons.

Let us continue with the example and assume that the simulation clock advances

![Figure 27 - LoadersWait After the Entry of the First Loader](image)

Figure 27 - LoadersWait After the Entry of the First Loader

Using ‘TimeIn’ or any constant expression (i.e., an expression that returns the same number regardless of which resource is cursored) as discipline results in FIFO, which is the default. In practice, it is not wise to explicitly specify a Discipline to achieve FIFO. This is because when the default Discipline has not been changed through a DISCIPLINE statement, Stroboscope places new resources at the end of Queues without evaluating any expressions or performing comparisons.

Let us continue with the example and assume that the simulation clock advances

![Figure 28 - Second Loader to Enter LoadersWait](image)

Figure 28 - Second Loader to Enter LoadersWait
to time 165.4. At this time another loader, shown in Figure 30, arrives.

LoadersWait cursors the entering loader and evaluates the Discipline to determine the loader’s tag. The result is 138-82=56 because SimTime is greater than 160. When LoadersWait cursors the loader in position 1, the result is 17-73 = -56, which is smaller than 56. LoadersWait then cursors the loader in position 2 with a result of 137-82 = 55,
which is also smaller than 56. *LoadersWait* continues cursoring and evaluating its Discipline. When it gets to position 3, the result is 144-82 = 62, which is greater than 56. At this point, *LoadersWait* inserts the entering loader in position 3, moving the loaders that occupied positions 3 to 12, to positions 4 to 13. *LoadersWait* now looks as shown in Figure 31.

![Figure 31 - LoadersWait After the Entry of the Third Loader](image)

An inspection of the loaders in *LoadersWait* indicates that the Queue is not sorted according to the current discipline. Only the first 4 loaders follow the discipline expression. Evaluating the discipline when the loader at position 5 is cursored returns 95-73 = 22, which is smaller than the result when the loaders at positions 2,3, or 4 are cursored. The discipline expression used to illustrate this is not realistic; it was
specifically chosen to point out Queue Discipline limitations. Most real queuing disciplines do not exhibit this behavior. For these cases, the use of Disciplines to maintain ordering is adequate.

Stroboscope does not maintain the resources in Queues continuously sorted according to the discipline expression for several reasons:

- A continuous re-sort is very expensive in terms of speed.
- Most of the time the “sort on insertion” method described above works well.
- More than one Combi can require the resources in the Queue; and the order in which they demand these resources can be different. Thus, one of the Combis could take advantage of the Queue’s discipline but the other must re-sort the resources in the Queue anyway.
- Discipline expressions can be very dynamic. As a result, the time at which a Queue becomes “unsorted” may not match the occurrence of a discrete event.
- As will be seen later, Draw Link attributes allow for enormous flexibility in selecting resources from Queues. Those attributes can be used in cases where the Queue’s Discipline does not suffice.

If a Queue’s Discipline is ‘-TimeIn’, the result is LIFO (i.e., Last In, First Out). This is because the TimeIn property of the entering resource is guaranteed to be the largest (although not strictly largest) among the resources in the Queue (i.e., resources in the Queue could not have entered the Queue in the future). Thus, a ‘-TimeIn’ discipline guarantees that the tag of the entering resource is the smallest (large, but negative). As a consequence, the entering resource goes to the front of the Queue.

The above paragraph’s discussion of “-TimeIn” as a Discipline is not absolutely correct. If several resources enter the Queue at exactly the same simulation time, they will be arranged in the order in which they arrived (within themselves). This is because these resources tie in the evaluation of ‘-TimeIn’, and the Queue discipline logic gives preference to resources already in the Queue.

An “absolute LIFO” Queue discipline requires a little extra work. The following code illustrates this:
SAVEPROPS Loader EntryNumber; /additional SaveProp for Loaders

ONRELEASE L2 ASSIGN EntryNumber LoadersWait.TotCount;
DISCIPLINE LoadersWait -EntryNumber;

The above example works well because LoadersWait.TotCount, whose value updates the EntryNumber SaveProp, increases by one every time a loader enters LoadersWait. Thus, no two resources will tie in their value of EntryNumber, and the entering resource is guaranteed to have the largest one (when negated becomes the smallest one). The EntryNumber property is updated when loaders are released through L2. If other links enter LoadersWait, then EntryNumber must be updated when loaders flow through those links as well.

At first glance, it may appear that updating EntryNumber in the “OnEntry LoadersWait” action event would be wiser. This is not the case, because the OnEntry actions for a Characterized Queue are performed after the entering resource has been positioned.

It is not necessary to resort to work-arounds in order to achieve an “absolute LIFO” Queue discipline (or any other discipline where ties have an undesirable effect). It is more straight forward to keep the Queue as FIFO (the default), and to remove resources from the “back” of the Queue. This is done by setting the “RevOrder” flag of the Draw Link that removes resources from the Queue. The REVORDER statement turns on the RevOrder flag:

Syntax:  REVORDER DrawOrReleaseLink [LogicalExp];
Example:  REVORDER L1;

LogicalExp is an optional argument. If used, it determines whether to turn the RevOrder flag on or off. When LogicalExp returns TRUE, the flag is turned on. When it returns FALSE, the flag is turned off. When LogicalExp is omitted, as in the example, the RevOrder flag is turned on. The RevOrder flag has the additional benefit that it is an attribute of the link rather than of the Queue. Thus, not all Draw Links need to get resources from the “back of the Queue”.

Copyright © 1996 Julio C. Martínez
12.1.2 Drawing Characterized Resources

A Characterized Draw Link with all its default attributes behaves pretty much like a Generic Draw Link.

- The predecessor Queue is assumed to support the successor Combi if the Queue is not empty (i.e., the link’s Enough is !Queue.CurCount).

- If the Combi starts, it will remove one resource from the Queue — the one at the front of the Queue (i.e., the link’s DrawUntil is Link.nDraws).

Thus, as far as Combi instantiations are concerned, generic resources used in discrete amounts can be substituted by characterized resources easily. The required modifications to a model are minor.

Characterized resources can carry very rich information. This enables precise control over characterized resource acquisition. This control can be achieved through Draw Link attributes that:

- Determine if the Queue can support the Combi by detailed examination of the Queue’s content. This is done by specifying an Enough attribute that can not only count how many resources reside in a Queue, but can also access aggregate statistics about their properties. With the use of “Filters”, to be described later, property statistics can be aggregated over a carefully selected subset of the Queue’s content.

- Draw resources from the “back” of a Queue through the RevOrder attribute (a flag described in the previous subsection).

- Draw resources from Queues in an order given by a “sort key” that can be as complex and dynamic as needed, thus bypassing the Queue’s Discipline and overcoming discipline limitations. This is done through the DrawOrder attribute (to be discussed below).

- Draw only the resources that meet a condition. The condition can be as dynamic and complex as required. This is done through the DrawWhere attribute (to be discussed below).
• Draw resources until the set of resources drawn meet any complex condition (or until no more resources are available for drawing). This is accomplished through the DrawUntil attribute, which can reference the content of both the Queue and the starting Activity instance at a very detailed level (using aggregate or filtered aggregate statistics on the properties of the resources involved).

Characterized Draw Links include ordering attributes (e.g., RevOrder and DrawOrder) that are unavailable to Generic Draw Links. Characterized Draw Links have a DrawWhere attribute instead of the DrawAmount attribute of Generic Draw Links. This is because the concept of “amount” is not applicable to a specific characterized resource, and because the total number of resources drawn can be controlled through the DrawUntil attribute. What matters instead, is whether a particular characterized resource is suitable for drawing.

The aspects of controlling characterized resource acquisition are discussed below in the order in which they affect the drawing process.

**Sufficiency (when does a Characterized Queue contain enough ?)**

As with Generic Draw Links, a Characterized Draw Link’s Enough attribute determines whether the Queue can support the Combi. The pre-defined variables that access aggregate properties of the resources in Queues are particularly useful in the Enough expression.

The following example illustrates the use of such an Enough:

```plaintext
CHARTYPE SteelShape Weight;
/
  (tons)
/
Several shapes are transported at the same time.
/ It is not cost effective to transport shapes if their cumulative
/ weight is less than 5 tons.
ENOUGH SS1 ShapesToTransport.Weight.SumVal>=5;
```

The code above will only allow *TransportShapes* to start if the shapes to move weigh at least 5 tons.
Ordering (the order in which resources are removed)

By default, resources in a Queue are considered for drawing in the order in which they appear in the Queue. The resource at the front of the Queue is considered first. The resource at the back of the Queue is considered last. The Queue’s Discipline attribute allows us to control, to a certain degree, the order in which resources are placed within the Queue. Thus, in most cases the default order of consideration is adequate.

A Queue’s Discipline alone cannot model a situation in which more than one Combi draws resources from the same Characterized Queue, if the drawing order for each Combi is different. Queue Discipline is also inadequate in cases where the required order of consideration is highly dynamic.

It is possible to change the order of resource consideration in two ways. The first alternative is to turn on the link’s RevOrder flag through the REVORDER statement. This only works if the desired order is the reverse of the Queue’s discipline. A typical use of RevOrder is to model LIFO (i.e., Last In, First Out) because a Queue’s default discipline is the reverse (FIFO, or First In, First Out).

The following statement enhances the preceding example so that steel shapes are removed in the reverse order in which they entered ShapesToTransport:

REVORDER SS1; /get shapes in reverse order of entry

The second alternative in changing the order of resource consideration for drawing is through the DrawOrder cursored attribute. Although DrawOrder is an attribute of the link, the link cursors the resources that reside in the Queue. The DrawOrder expression is used as a key to sort the resources within the Queue in ascending order. The sorting does not alter the order of the resources within the Queue, but rather the order in which the resources are considered for drawing through the link. The DrawOrder attribute is changed with the DRAWORDER statement:

Syntax:    DRAWORDER DrawLink CursoredExpression;
Example:   DRAWORDER L1 FuelLeft;

Because the sole purpose of DrawOrder is to sort resources within the predecessor Queue, the shorthand form of cursored property access is available. The example above
uses $FuelLeft$ as a shorthand for $L1.FuelLeft$. In any case, the cursored resource is not held by $L1$, but by the Queue ($LoadersWait$).

The sort is performed by using $CursoredExpression$ as the sort key (ascending). The link cursors every resource in the predecessor Queue one by one. While each resource is cursored, the link evaluates $CursoredExpression$, and temporarily attaches the result to the resource. The resources are then sorted in ascending order according to the temporary values attached to them.

The following example illustrates the use of the DrawOrder attribute.

```
/L2 draws loaders with emptier tanks first
DRAWORDER L2 FuelLeft-TankCapacity;
```

The DrawOrder in the above example is necessary because both $LoadHaulers$ and $FillUpLoader$ draw loaders from $LoadersWait$, yet the appropriate order of consideration for drawing them are completely different.

The sorting done by the DrawOrder attribute is expensive in terms of simulation speed. Therefore, DrawOrders should only be used when the Queue’s discipline or the link’s RevOrder flag are not capable of modeling the drawing process appropriately.

**Discriminating** (which resources can be removed)

In some situations, not all the resources in a Queue are suitable for the task represented by a succeeding Combi. In these cases, it is necessary to discriminate which resources are allowed to be drawn. For example, a Combi can represent the transportation of steel shapes in a flatbed. The shapes have a certain length and cannot be carried in flatbeds that are too short to accommodate them. Thus, depending on the length of the flatbed, a steel shape may or may not be carried.

A Characterized Draw Link’s DrawWhere cursored attribute can be used to determine which resources can be drawn through the link. DrawWhere is similar to
DrawOrder in that the resource cursored by the link is inside the Queue. Resources are eligible for drawing if, when cursored, the DrawWhere returns TRUE. The default DrawWhere always returns TRUE and thus, by default, all the resources in a Queue are eligible for drawing. The DrawWhere attribute can be changed with the DRAWWHERE statement:

Syntax: DRAWWHERE DrawLink CursoredExpression;
Example: DRAWWHERE L2 ‘FuelLeft/TankCapacity < 0.20’;

The example above indicates that only loaders whose tank contain less than 20% of the capacity should be drawn through link L2.

The DrawWhere is evaluated to determine if a resource should be drawn, just before the potential draw. Thus, at the time of evaluation, other resources may have already been drawn through the link. The example for the “stopping” subsection below shows the DrawWhere attribute in use.

**Stopping (when and how to stop drawing)**

Draw Links stop drawing when their DrawUntil returns TRUE regardless of whether they are Characterized or Generic. The DrawUntil is not a cursored attribute because at the time of its evaluation there is no logical choice of resource to cursor. The ability to aggregate the properties of resources already drawn by the Combi, or of the resources remaining in the Queue, or of the resources in any other node, provide a plethora of possibilities for DrawUntil expressions. An example is illustrated below:

```
CHARTYPE FlatBed MaxPayLoad Length; / FB
CHARTYPE SteelShape Weight Length; / SS

/Get only shapes that fit in the flatbed AND that will not make the
/weight carried by the flatbed exceed the flatbed’s maximum pay load.
DRAWWHERE SS1 'Length<LoadShapes.FlatBed.Length & Weight<
   LoadShapes.FlatBed.MaxPayLoad=LoadShapes.SteelShape.Weight.SumVal';

/The above assumes that one flatbed and an undetermined number of steel
/shapes are already in the LoadShapes instance.

/Load the flatbed until its capacity is reached (this may not happen)
DRAWWUNTIL SS1
   'LoadShapes.FlatBed.MaxPayLoad==LoadShapes.SteelShape.Weight.SumVal';
```
In the above example, two conditions are required for a steel shape to be drawn: 1) the steel shape must fit in the flatbed, and 2) the weight of the steel shape must not make the load carried by the flatbed exceed its maximum payload. Note that the DrawWhere assumes that a flatbed is already in LoadShapes, and that it is sensitive to the weight of the steel shapes that have already been drawn. In order for the flatbed to be drawn before the steel shapes, link FB1 must be defined before link SS1.

The DrawUntil in the above example stops the drawing through link SS1 when the load carried by the flatbed equals the flatbed’s capacity. This will only happen when the weight of the steel shapes can add up exactly to the flatbed’s payload. Note that the DrawWhere ensures that the load carried will never exceed the payload.

The Enough, RevOrder, DrawOrder, DrawWhere, and DrawUntil attributes of a characterized link can be combined to model very complex resource acquisitions. The modeling power of these attributes can be enhanced through the use of “Filters”, which will be discussed in section 12.2.

12.1.3 Releasing Characterized Resources

Mass conservation exists in both Generic and Characterized resources during Combi startup. Resources, whether generic or characterized, move from Queues to Combis. The total amount of resources in a system, before and after a Combi starts, is the same.

When Activity instances terminate, however, Stroboscope does not manage characterized and generic resources in the same manner. Generic resources never leave terminating Activity instances. What actually happens, as described in detail in earlier chapters, is that Release Links “create” a certain amount of the generic resource and then transfer it to the successor. When the instance finishes terminating, the generic resources held by it are destroyed.

Characterized resources are different. Mass conservation exists both at Combi startup and at Activity instance termination. When a terminating Activity instance releases a characterized resource, the resource actually moves from the predecessor to the successor. After the release, the amount of resource in the terminating instance decreases.
If a characterized resource is not released, then it is destroyed with the instance (thus reducing the population of resources of that type).

The release process for characterized links is very much like the drawing process. The attributes have names that suggest the same behavior: RevOrder, ReleaseOrder, ReleaseWhere, and ReleaseUntil. The default for these attributes are also the same except for that of ReleaseUntil.

**RevOrder**

When an Activity instance is starting, it receives characterized resources in the order in which the resources are drawn. The resources stay in this order within the instance. When the instance terminates, the resources are considered for release in the order in which they were drawn. The order of consideration can be reversed by setting the Release Link’s RevOrder attribute. This is done with the REVORDER statement:

Syntax:  \texttt{REVORDER DrawOrReleaseLink \{LogicalExp\};}
Example:  \texttt{REVORDER L2;}

This is the same syntax and usage discussed earlier for Characterized Draw Links. The \textit{LogicalExp} is optional. If omitted or used and TRUE, the RevOrder flag is turned on. If used and FALSE, the RevOrder flag is cleared. Since Activities do not have the equivalent of a Discipline, RevOrder is rarely used in Characterized Release Links. When release order is important, it is more common to use the ReleaseOrder attribute discussed below.

**ReleaseOrder**

A Characterized Release Link’s ReleaseOrder attribute is analogous to a Characterized Draw Link’s DrawOrder. Although ReleaseOrder is an attribute of the link, the link cursors the resource in the terminating Activity instance. The ReleaseOrder expression is used as a key to sort the resources (of the type released by the link) within the terminating Activity instance in ascending order. The order does not actually affect the resources in the Activity instance, but rather the order in which they are considered for release through the link. The ReleaseOrder attribute is changed with the RELEASEORDER statement:
Syntax:  RELEASEORDER CharReleaseLink CursoredExpression;
Example:  RELEASEORDER L2 FuelLeft;

CursoredExpression can access the properties of the cursored resource implicitly as shown in the example.

ReleaseWhere

A Characterized Release Link’s ReleaseWhere attribute is analogous to a Characterized Draw Link’s DrawWhere. The ReleaseWhere is used as a filter that determines which resources are suitable for release through the link. The link cursors the resource considered for release before evaluating the ReleaseWhere expression. If the result is TRUE, the link moves the resource to the successor. Otherwise, the link considers the next resource. The default ReleaseWhere always returns TRUE, indicating that all resources of the type that flow through the link are acceptable for release. The ReleaseWhere attribute is changed with the RELEASEWHERE statement:

Syntax:  RELEASEWHERE CharReleaseLink CursoredExpression;
Example:  RELEASEWHERE L2 FuelLeft/TankCapacity>0.20;

The example above indicates that only loaders whose tank contains more than 20% of the capacity should be released through link L2.

Like Draw Links, Characterized Release Links evaluate their ReleaseWhere attribute just before resources are considered for release. At that time, other resources may have already been released through the link. Thus, ReleaseWhere expressions can reference properties of the resources that remain in the predecessor, or that have already reached the successors.

ReleaseUntil

A Characterized Release Link’s ReleaseUntil is analogous to a Characterized Draw Link’s DrawUntil. When the ReleaseUntil expression returns TRUE, the release process through the link stops.

Contrary to the default DrawUntil, the default ReleaseUntil always returns FALSE. Thus, by default, all the resources that can flow through the link (i.e., that pass the ReleaseWhere) will be released. Due to “mass conservation” of characterized resources during Activity instance termination, the default behavior has implications when more than one link of the same characterized resource type leaves an Activity. All
the resources will flow through the link that is defined first (while processing the model input file), leaving no resources in the terminating instance. Thus, the second link of the given resource type will find no resources to release. This is contrary to the release of generic resources, where the Generic Release Links always create (as opposed to move) the amounts released, and leave the generic resource in the terminating instance intact.

The following network and code fragment illustrate the releasing process:

```
RELEASEORDER SH2 Weight;
RELEASEWHERE SH2 ‘Length<=5 & Weight+ShapesArrived.Weight.SumVal<=20’;
RELEASEUNTIL SH2 ShapesArrived.CurCount>=10;
```

The ReleaseOrder ensures that lighter shapes are unloaded first. The ReleaseWhere does not allow shapes longer than 5 meters to be unloaded. In addition, the ReleaseWhere only allows shapes whose weight does not make the total weight in ShapesArrived exceed 20 tons. The ReleaseUntil stops the releasing process when 10 or more shapes are already in ShapesArrived. Note that it is not necessary to consider the weight in the ReleaseUntil expression because the ReleaseWhere takes care of not allowing the weight in ShapesArrived to exceed the maximum allowed.

### 12.1.4 Generating Resources while Simulating

An Activity instance can only release the characterized resources it acquires. If a situation requires the release of characterized resources not held by an instance, then
these resources must be created explicitly at the appropriate moment so that they are available for release. Characterized resources are generated while the simulation is running (i.e., during the execution of a SIMULATE or SIMULATEUNTIL statement) through actions defined with the following action definition statements:

Syntax: BEFOREDRAWS Combi GENERATE [PRECOND LogExp] number SubType;
Syntax: BEFOREEND Activity GENERATE [PRECOND LogExp] number SubType;
Example: BEFOREEND BigLoadersArrive GENERATE 1 S926E;

These are action definition statements restricted to the BEFOREDRAWS and BEFOREEND action events. GENERATE is a special pre-defined action target that creates characterized resources; it must be typed exactly as shown. The first action argument (number) indicates the number of resources to create and the second action argument (SubType) indicates the SubType of the characterized resources to create. The specified number of resources of the corresponding SubType are created every time the action event occurs (unless PRECOND is used and LogExp is FALSE). The created resources join the instance in question. When the action is performed at the BEFOREDRAWS action event, the resources become part of the starting instance before any other resources are drawn from the Queues that precede the Combi. When the action is performed at the BEFOREEND action event, the resources become part of the terminating instance before any resources are released. The example instructs Stroboscope to create one S926E loader just before resources are released in terminating instances of BigLoadersArrive.

Action definition statements where GENERATE is the action target are useful for the modeling of arrivals. Assume, for example, that workers of different trades arrive at a tool crib to request tools. The “world” outside the tool crib is not modeled explicitly, hence the arrival of workers is modeled as a Poisson process. If workers need to be characterized resources, their arrival can be modeled with the following code:

```
Workers Wait W! Steel Workers Arrive

Assume that SteelWorker is a SubType of Worker and that
their Interarrival rate is 12 minutes (5 per hour)

SEMAPHORE SteelWorkersArrive !SteelWorkersArrive.CurInst; one at a time
```
Due to mass conservation of characterized resources, it would be cumbersome to model the arrival of the workers without acting on the GENERATE action target. Essentially, SteelWorkersArrive would need to remove workers from a “World” Queue. The Queue must be initialized with a very large number of workers so that it can always provide one to SteelWorkersArrive. When the workers leave, they should be returned to the “World” Queue for reuse. This arrangement works, but the population and lifetime statistics automatically collected by the system are not as good.

### 12.2 Filters

The role of a Draw Link’s Enough attribute is to determine if the predecessor Queue can support the task represented by the successor Combi. If all the predecessor Queues can support the Combi (and the Combi’s Semaphore is TRUE), an instance of the Combi starts and draws resources through its incoming links.

A Draw Link’s Enough attribute should predict if a drawing will be satisfactory. If the Enough attributes of all the links that enter a Combi predict that the drawing will be successful (i.e., they return TRUE), the Combi is committed to start. If the predictions were accurate, the Combi will be able to draw all the resources it needs from the Queues that precede it. As in real life, predictions are not always correct and a Combi may not be able to draw all the resources it requires. The quality of a prediction depends on how much information is available (to make the prediction).

So far, the information accessible to Enough expressions is limited to a count of the resources in Queues (e.g., Queue.CurCount), or to a property (in the aggregate) of the resources in Queues (e.g., Queue.Property.SumVal). Given the selectivity with which a Combi can draw, this information is not sufficient to predict the success of a drawing.

Filters are cursoring objects that restrict the set of resources used for a resource count or for aggregating property values. Filters are necessary, among other things, because they provide information that allow Enoughs to be very good predictors.

Filters are defined with the FILTER statement:
Syntax: FILTER FilterName CharType CursoredFilterExpression;
Example: FILTER EmptyTankLoaders Loaders FuelLeft/TankCapacity<=0.20;

*FilterName* is the name of the Filter. *CharType* is the characterized resource type of the resources to which the Filter can be applied. *CursoredFilterExpression* is the expression that restricts the domain of the filter. Only those resources that, when cursored by the Filter, make *CursoredFilterExpression* return TRUE, pass the filter. The example defines *EmptyTankLoaders* as a Filter. It applies to loaders, and restricts them to those whose ratio of fuel left to tank capacity is less than 20%.

The FILTEREXP statement can be used to change a Filter’s filtering expression after the Filter has been defined:

Syntax: FILTEREXP FilterName CursoredFilterExpression;
Example: FILTEREXP EmptyTankLoaders TankCapacity-FuelLeft>=70;

The example replaces *EmptyTankLoaders’s* filter expression with ‘*TankCapacity-FuelLeft*>=70’. The FILTEREXP statement allows “forward definition” of Filters, which is useful in situations that will be discussed later.

Filters do not hold resources; they work on the resources held by Queues or Activity instances. Filters are used in the following pre-defined variables:

\[Node.FilterName.Property.\{AveVal|MaxVal|MinVal|SDVal|SumVal\}\]
\[Node.FilterName.Count\]

*Node* is the Queue or HeteroHolder (instance) that holds the resources to be filtered. *FilterName* is the name of the Filter. *Property* is the property of the resource that will be aggregated. The last part of the first variable shown can be omitted if the number of resources that pass the filter is exactly one. Otherwise, one of AveVal, MaxVal, MinVal, SDVal, or SumVal must be used. AveVal, MaxVal, and MinVal require that at least one resource pass the filter. SDVal requires that at least two resources pass the filter.

The variable *LoadersWait.EmptyTankLoaders.Count*, for example, returns the number of loaders in *LoadersWait*, whose ratio of fuel left to tank capacity is less than 20%. The variable *LoadersWait.EmptyTankLoaders.TankCapacity.SumVal* returns the sum of the tank capacity of all the loaders in *LoadersWait* that have less than 20% of fuel left.
If it is necessary, for example, to determine the total fuel needed by all the empty loaders in \textit{LoadersWait}, the following expression could be used:

\texttt{‘LoadersWait.EmptyTankLoaders.TankCapacity.SumVal - LoadersWait.EmptyTankLoaders.FuelLeft.SumVal’}. A simpler but equivalent expression can be used with the aid of a \texttt{Loader VarProp}:

\texttt{VARPROP Loader FuelNeeded TankCapacity-FuelLeft;}

The expression can now be simplified to

\texttt{‘LoadersWait.EmptyTankLoaders.FuelNeeded.SumVal’}.

The resources in Activity instances can also be filtered. In these cases, the pre-defined variables that use the Filter are instance variables. Thus, an instance of the Activity in question must be in context when the variable is evaluated. It is rarely necessary to apply Filters to Activity instances because the resources in an Activity instance have already been filtered by the DrawWhere attributes of the incoming links.

\section*{12.2.1 How Filters Work}

Filters are cursoring objects that operate in a very simple manner. They cursor each and every resource held by the node to which the Filter is applied. While each resource is cursored, the Filter evaluates its filtering expression. Those resources that, when cursored, make the filtering expression TRUE, form a filtered subset. The Filter pre-defined variables allow us to count the number of resources in the subset, or to aggregate a property of the resources across the subset.

A Filter’s filtering expression has implicit access to the properties of the resource cursored by the Filter. The ‘\texttt{FuelLeft}’ and ‘\texttt{TankCapacity}’ used in the filtering expression ‘\texttt{FuelLeft/TankCapacity<=0.20}’, refer to the corresponding properties of the resource cursored by \texttt{EmptyTankLoaders} (a Filter). The same expression, using explicit access, can be rewritten as

\texttt{‘EmptyTankLoaders.FuelLeft / EmptyTankLoaders.TankCapacity<=0.20’}.

The Filter’s cursor is independent of the filtered node’s cursor. Thus, even if \texttt{EmptyTankLoaders} is applied to \texttt{LoadersWait} (as it is in the above long expression), one
must use \( \text{EmptyTankLoaders}.\text{FuelLeft} \) and not \( \text{LoadersWait}.\text{FuelLeft} \). This is because the resource in question is cursored by \( \text{EmptyTankLoaders} \) and not by \( \text{LoadersWait} \).

When Stroboscope evaluates the expression

\[ \text{LoadersWait}.\text{EmptyTankLoaders}.\text{FuelNeeded}.\text{SumVal} \]

it goes through several steps:

- \( \text{EmptyTankLoaders} \) focuses on \( \text{LoadersWait} \).

- For each loader in \( \text{LoadersWait} \):
  
  - \( \text{EmptyTankLoaders} \) cursors the loader.

  - \( \text{EmptyTankLoaders} \) evaluates its filtering expression
    \[ \text{FuelLeft}/\text{TankCapacity} \leq 0.20 \].

  - If the result is TRUE, \( \text{EmptyTankLoaders} \) tags the loader.

- \( \text{EmptyTankLoaders} \) forms a subset of loaders that consists of all the loaders in \( \text{LoadersWait} \) that have been tagged.

- \( \text{EmptyTankLoaders} \) calculates the \( \text{FuelNeeded} \) VarProp for each of the loaders in the subset. Each value is placed in a “statistical calculator”.

- All the values in the “statistical calculator” are added, and the result is returned.

It is important to observe that each part (parts are separated by periods) of

\[ \text{LoadersWait}.\text{EmptyTankLoaders}.\text{FuelNeeded}.\text{SumVal} \]

is completely carried out before going to the next. Stroboscope does not look at \( \text{FuelNeeded} \) until the entire subset of loaders in \( \text{LoadersWait} \) is determined. Similarly, Stroboscope sums up the values of \( \text{FuelNeeded} \) only after the value of \( \text{FuelNeeded} \) for all resources has been computed.

The purpose of Filters is to aggregate a property over the resources that pass the Filter, or to count these resources. Thus, the order of the resources in the set that results from the application of a Filter to a node is irrelevant.
12.2.2 Common Filter Applications

Filters are very powerful modeling objects. They can be designed to provide a wide range of useful information about the resources in Queues and Activities. The power of Filters is limited only by the imagination of the modeler. The next few subsections show how Filters can be applied to several common situations.

The next characterized resource to be drawn

The resources drawn from one Queue often depend on the resources drawn from another Queue. For example, the amount of dirt to load on a hauler usually depends on the size of the hauler. Once a Combi is committed to start, it is very easy to draw the correct amount of dirt. It is sufficient to draw the hauler first, and then use its size as the DrawAmount attribute for the dirt link.

Suppose that haulers are only loaded when there is enough dirt to fill them up. In this case it is necessary to specify the Enough attribute of the link that brings in dirt in terms of the size of the hauler. Since the hauler that will be drawn (if there is enough soil to fill it up) is still in the Queue, it is necessary to find out the size of the hauler at the front of the Queue before making the commitment to actually draw it. To do this, a Filter must be applied to the Queue so that only the hauler at the front of the Queue passes the Filter. The following code fragment illustrates this for a hauler Queue with default discipline (FIFO):

```
FILTER HaulerAtFront Hauler TimeIn==HaulersWait.TimeIn.MinVal;
ENOUGH SL1 SoilToMove.CurCount>=HaulersWait.HaulerAtFront.Size;
```

/Hauler is characterized and Soil is generic

/The hauler at the front is the one that entered the Queue earliest

Copyright © 1996 Julio C. Martínez
Recall that TimeIn is a system-maintained property that indicates the simulation time at which the resource entered the last node (in this case HaulersWait). Thus, the lower the value of TimeIn, the longer the hauler has been in HaulersWait. The hauler with the smallest TimeIn is the hauler that has been waiting the longest. The variable HaulersWait.TimeIn.MinVal returns the minimum value of TimeIn aggregated over all the resources in HaulersWait. HaulerAtFront allows only those haulers whose TimeIn matches HaulersWait.TimeIn.MinVal to pass. When HaulerAtFront is applied to HaulersWait, only the hauler with the minimum TimeIn passes. Thus, the variable HaulersWait.HaulerAtFront.Size returns the size of the hauler that has been waiting the longest (and will be loaded first in a FIFO discipline).

The Filter and code above work well if haulers never enter HaulersWait at the same time. This will rarely happen after the simulation starts, but often happens at the beginning of the simulation when Queues are initialized (all such resources enter the Queue at the same time). If more than one hauler enters HaulersWait at the same simulation time, their TimeIn will be the same. When these haulers get to the front of the Queue, they will both pass HaulerAtFront. HaulersWait.HaulerAtFront.Size will not be legal because the number of haulers in HaulersWait that pass HaulerAtFront is not exactly one.

In order to overcome the problem of ties, it is convenient to set the Queue’s Discipline or the Draw Link’s DrawOrder with a VarProp that is guaranteed to be unique. The following code is a revision of the one above:

/mimic ResNum as a 2nd sort key
VARPROP Hauler DiscipProp TimeIn+ResNum*10^-5;

/for all practical purposes this discipline is FIFO
DISCIPLINE HaulersWait DiscipProp;

/The hauler at the front is the one with smallest DiscipProp
The code defines the `DiscipProp` VarProp so that no two resources tie in their values of it. The haulers are essentially sorted by TimeIn. Since ResNum is unique within all resources of the same type, ResNum is used as a tie breaker or second sort key. The value of $10^{-5}$ used to multiply ResNum is chosen arbitrarily so that the result is insignificant in comparison to TimeIn. The code works with any queuing discipline by simply redefining the `DiscipProp` VarProp with the appropriate (unique) sorting key.

Note that in the specific case of strict FIFO, it is more convenient to assign a constantly increasing value (e.g., `HaulersWait.TotCount`) to a SaveProp when each hauler enters `HaulersWait`. This property could then be used to identify the hauler at the front of the Queue. In this case there is no need to explicitly set a discipline for `HaulersWait` (and thus benefit from the speed of the default FIFO discipline). This method is actually used in the example below.

**Determining if sufficient characterized resources of one type are available to match the first resource of another type to be drawn**

In the case described above, the amount of generic resource (soil) necessary depends on a property (size) of a characterized resource (hauler) that will also be drawn. Sometimes the series of characterized resources (e.g., steel shapes) necessary for an operation depend on a property of another characterized resource (e.g., flatbed). The following code illustrates how **two Filters** can be used to model such situations:

```
CHARTYPE FlatBed PayLoad Length;
/                (tons)   (m)
/ ... definition of FlatBed SubTypes omitted
```
SAVEPROP FlatBed EntryNum; /unique, strict FIFO, sort key

CHARTYPE SteelShape Weight Length;
   (tons) (m)
/ ... definition of SteelShape SubTypes omitted

/Strategy:
/ SteelShapes must fit in FlatBed
/ FlatBed not loaded with less than 75% of PayLoad
/ FlatBeds line up and must be served in strict FIFO order

/Assign value to EntryNum SaveProp when FlatBeds enter Queue
ONENTRY FlatBedsWait ASSIGN EntryNum FlatBedsWait.TotCount;

/FILTER & variables to determine length and pay load of next Flatbed
FILTER NextFlatBed FlatBed EntryNum==FlatBedsWait.EntryNum.MinVal;
VARIABLE NextFlatBedLength FlatBedsWait.NextFlatBed.Length;
VARIABLE NextFlatBedPayLoad FlatBedsWait.NextFlatBed.PayLoad;

/FILTER to determine subset of SteelShapes that Fit in next Flatbed
FILTER FitInNextFlatBed SteelShape Length<=NextFlatBedLength;

VARIABLE TotalWgtOfShapesThatFit
   SteelShapesWait.FitInNextFlatBed.Weight.SumVal;

/we have enough Steel Shapes if the weight of those that fit is at least
/75% of the pay load of the next Flatbed
ENOUGH SS1 TotalWgtOfShapesThatFit>=0.75*NextFlatBedPayLoad;

/if LoadShapes is indeed able to start, the attributes of SS1
/must now assume that the FlatBed is in the LoadShapes instance

/Get only shapes that fit in the flatbed AND that will not make the
/weight carried by the flatbed exceed the flatbed's pay load.
DRAWWHERE SS1 'Length<=LoadShapes.FlatBed.Length & Weight <=
   LoadShapes.FlatBed.PayLoad-LoadShapes.SteelShape.Weight.SumVal';

/Load the flatbed until it is fully loaded. This may not happen, but
/all shapes will be tried before giving up
DRAWUNTIL SS1
   'LoadShapes.SteelShape.Weight.SumVal==LoadShapes.FlatBed.PayLoad';

In the code above, FitInNextFlatBed's filter expression depends (indirectly) on
another Filter, NextFlatBed. The following pseudo-code illustrates the process that
Stroboscope goes through when it evaluates TotalWgtOfShapesThatFit:

nSS=SteelShapesWait.CurCount
nFB=FlatBedsWait.CurCount
TotalWgtOfShapesThatFit=0
For I=1 to nSS
   NextFlatBedLength=0;
   For J=1 to nFB
      if FlatBed(J).EntryNum=FlatBedsWait.EntryNum.MinVal then
         'in this case the above will be TRUE only for J=1
         NextFlatBedLength=NextFlatBedLength+ FlatBed(J).Length
      End if
   Next J
if SteelShape(I).Length<=NextFlatBedLength Then
    TotalWgtOfShapesThatFit=TotalWgtOfShapesThatFit+SteelShape(I).Weight
End if
Next I

Essentially, a Filter that uses another Filter is analogous to a nested loop in a conventional programming language. Notice that the entire inner loop (J) is repeated for every iteration of the outer loop (I). Thus, the value of NextFlatBedLength can be “made” to be different for each of the steel shapes. Even though it makes no sense to do so here, this observation illustrates that the use of two filters can provide a very powerful selection mechanism.

**Determining if sufficient characterized resources of one type are available to match any of the resources of another type**

The example above works when FlatBedsWait’s discipline is strictly FIFO and the flatbed at the front of the Queue must be loaded before any other flatbed is loaded. If there are not enough steel shapes to load the flatbed at the front of the Queue, the other flatbeds have to wait.

When the FIFO discipline is not strictly enforced, it may be possible to load a flatbed that is not at the front of the Queue even though the one at the front cannot be loaded. The following code illustrates one way to model this:

```
CHARTYPE FlatBed PayLoad Length;
/ (tons) (m)
/ ... definition of FlatBed SubTypes omitted

CHARTYPE SteelShape Weight Length;
/ (tons) (m)
/ ... definition of SteelShape SubTypes omitted
```

/Strategy:

Copyright © 1996 Julio C. Martínez
SteelShapes must fit in FlatBed.
FlatBed not loaded with less than 75% of PayLoad.
FlatBeds served in FIFO order if possible. Otherwise serve the first
FlatBed that can be loaded with at least 75% of PayLoad.

"Forward definition" of Filter. Later, the filter expression will be
changed so that only the steel shapes that fit the loader cursored by
Filter FBCanBeLoaded may pass (the filter expression cannot be set
now because FBCanBeLoaded is not defined yet)
FILTER SSThatFit SteelShape 1;

VarProp that will be TRUE for FlatBeds that can be loaded to at least
75% of their capacity (with the shapes in SteelShapesWait)
VARPROP FlatBed PlentyShapesAvailable
   SteelShapesWait.SSThatFit.Weight.SumVal>=0.75*PayLoad;

Filter to single out those flatbeds that have plenty of shapes
available to load them
FILTER FBCanBeLoaded FlatBed PlentyShapesAvailable;

A flatbed can be cursored by either the FBCanBeLoaded Filter or the
FB1 link. This variable provides common access to the length of the
cursored flatbed
VARIABLE CurrentlyCursoredFlatBedLength

Set filter expression that determines the subset of SteelShapes (in the
SteelShapes Queue) that fit in the Flatbed currently cursored
FILTEREXP SSThatFit Length<=CurrentlyCursoredFlatBedLength;

we have enough flatbeds if at least one of them can be loaded
(steel shapes are taken care of automatically)
ENOUGH FB1 FlatBedsWait.FBCanBeLoaded.Count;

if LoadShapes is indeed able to start, certain link attributes must
ensure that the appropriate resources are drawn

the flatbed must be such that plenty shapes fit in it
DRAWWHERE FB1 PlentyShapesAvailable;

the attributes of SS1 must assume that the FlatBed is in the
LoadShapes instance

Get only shapes that fit in the flatbed AND that will not make the
weight carried by the flatbed exceed the flatbed's pay load.
DRAWWHERE SS1 'Length<=LoadShapes.FlatBed.Length & Weight <=
   LoadShapes.FlatBed.PayLoad-LoadShapes.SteelShape.Weight.SumVal';

Load the flatbed until it is fully loaded. This may not happen, but
all shapes will be tried before giving up
DRAWUNTIL SS1
   'LoadShapes.SteelShape.Weight.SumVal==LoadShapes.FlatBed.PayLoad';

In the code above, two mutually dependent Filters are used. This requires that one
Filter, SSThatFit, be defined initially with a dummy filter expression. Once SSThatFit is
defined, it can be used (indirectly) in the definition of the second Filter,
FBCanBeLoaded. The filter expression for SSThatFit, which makes indirect reference to the flatbed cursored by FBCanBeLoaded, can then be set.

SSThatFit’s role is very simple, it filters out those steel shapes that fit in the flatbed currently being examined by either FBCanBeLoaded or by FB1’s DrawWhere.

FBCanBeLoaded’s role is to filter out those flatbeds that can be sufficiently loaded (75% of pay load) with the steel shapes available in SteelShapesWait.

The code is very straight forward. After SSThatFit is defined, it is assumed to be capable of performing its duty (even though it is defined with a dummy filter expression). Under that assumption, it is possible to define a flatbed VarProp (PlentyShapesAvailable) that compares the total weight of those steel shapes in SteelShapesWait that fit in the flatbed, with the flatbed’s pay load. If the weight of these shapes add up to at least 75% of the flatbed’s pay load, PlentyShapesAvailable is TRUE, otherwise it is FALSE. FBCanBeLoaded is then used to filter out those flatbeds whose PlentyShapesAvailable VarProp is TRUE.

Flatbeds are examined under two situations. The first is through FBCanBeLoaded, during the evaluation of FB1’s Enough. The second is when FB1 is considering the flatbed for drawing (while evaluating FB1’s DrawWhere).

CurrentlyCursoredFlatBedLength is defined to provide access to the length of the flatbed, whether it is cursored by FBCanBeLoaded, or by FB1. In its definition, FBCanBeLoaded.HasCursor is used to determine if FBCanBeLoaded is cursoring (if not, then FB1 is cursoring). Obviously, this variable could have used ‘FB1.HasCursor?’ with the TRUE and FALSE parts of the conditional reversed.

The filtering expression for SSThatFit, which up to now was assumed to be correct, can now be set. It simply compares the length of the steel shape to the length of the currently cursored flatbed. Only those shapes that fit in the cursored flatbed pass.

LoadShapes can start whenever flatbeds that can be loaded exist in FlatBedsWait. The existence of such a flatbed guarantees that enough steel shapes are available in SteelShapesWait. Hence, resource sufficiency can be determined entirely by FB1’s Enough, which simply counts how many flatbeds can be loaded (any non-zero number is TRUE and will do).
Once _LoadShapes_ is committed to start, it must draw a flatbed that can be loaded (_FB1_’s _Enough_ indicates if at least one flatbed exists, but it does not guarantee that the _next_ flatbed considered for drawing can indeed be loaded). The correct drawing is done by setting _FB1_’s _DrawWhere_ to the potential flatbed’s _PlentyShapesAvailable_ VarProp.

_SS1_’s _DrawWhere_ and _DrawUntil_ attributes are similar to those used in previous examples. They ensure that the appropriate steel shapes are loaded.

The following pseudo-code illustrates the process that Stroboscope goes through when it evaluates _FB1_’s _Enough_:

```plaintext
nSS=SteelShapesWait.CurCount
nFB=FlatBedsWait.CurCount
NumOfFlatBedsThatCanBeLoaded=0
For I=1 to nFB
    WeightOfShapesThatFit=0;
    For J=1 to nSS
        if SteelShape(J).Length<=FlatBed(I).Length then
            WeightOfShapesThatFit = WeightOfShapesThatFit + SteelShape(J).Weight
        End if
    Next J
    if WeightOfShapesThatFit>=0.75*FlatBed(I).PayLoad Then
        NumOfFlatBedsThatCanBeLoaded=NumOfFlatBedsThatCanBeLoaded+1
    End if
Next I
```

**Aggregating a property over the M<sup>th</sup> to N<sup>th</sup> resources, ranked according to a given key, that meet a certain condition**

This example is the most generic form of access to a ranked set of resources that meet a specific condition. The example is first presented in a highly parameterized form that can be used as a recipe. A subsequent example modifies the recipe for a particular situation.

//Assumptions:
//   ResType:
//       Defined as a characterized resource type that will be ranked.
//   ConditionVP:
//       Defined as a VarProp of ResType. It is TRUE for resources that meet the condition of interest and FALSE for resources that do not.
//   RankingVP:
//       Defined as a VarProp of ResType. It has been defined so that it is unique within all resources being ranked. RankingVP provides the sorting key for the ranking.
// AggProp:
// Defined as a property of ResType. This is the property to be aggregated.

// AggPropSelector:
// This is one of AveVal, MaxVal, MinVal, SDVal, or SumVal

// Container:
// Defined as Characterized Queue that holds resources of type ResType. This Queue contains the resources being ranked.

// M and N:
// These are numbers that indicate the rankings of interest.
// Resources ranked M through N, inclusive, will be aggregated.
// The ranking is only within those resources that meet the condition (ConditionVP).

FILTER ResRankedLower ResType 1; /forward declaration
FILTER ResRankedMtoN ResType 'ConditionVP &
  Container.ResRankedLower.Count>=M-1 &
  Container.ResRankedLower.Count<N';
FILTEREXP ResRankedLower 'ConditionVP &
  RankingVP<ResRankedMtoN.RankingVP';
DISPLAY Container.ResRankedMtoN.AggrProp.AggrPropSelector;

The following example uses the above recipe to determine the total amount of fuel needed to fill up the 4th to 7th emptiest loaders (ranked by percentage of fuel left), but considering only those loaders that contain less than 25% of fuel in their tanks:

/ Loader is ResType
CHARTYPE Loader TankCapacity;
SAVEPROPS Loader FuelLeft;

/ FuelLeftPct is RankingVP
VARPROP Loader FuelLeftPct FuelLeft/TankCapacity;

/ NeedsFuel is ConditionVP
VARPROP Loader NeedsFuel FuelLeftPct<=0.25;

/ FuelNeededAmt is AggProp
VARPROP Loader FuelNeededAmt TankCapacity-FuelLeft;

/ LoadersWait is Container
QUEUE LoadersWait Loader;

/ Substitute appropriately in the recipe
FILTER LoadersEmptier Loader 1; /forward declaration
FILTER FourthToSeventhEmptiest Loader 'NeedsFuel &
  LoadersWait.LoadersEmptier.Count>=3 &
LoadersWait.LoadersEmptier.Count<7';
FILTEREXP LoadersEmptier
 'NeedsFuel &
 FuelLeftPct<FourthToSeventhEmptiest.FuelLeftPct';
/ SumVal is AggrtSelector

When applied to a specific situation, the recipe is straightforward. It can be explained as follows:

- Define LoadersEmptier and assume that it is capable of selecting those loaders that meet the following conditions:
  - they need fuel
  - they have less fuel left (percentage wise) than the loader cursored by FourthToSeventhEmptiest.

- Define FourthToSeventhEmptiest so that only those loaders that meet the following conditions pass:
  - they need fuel
  - the count of loaders in LoadersWait that are emptier is >= to 3 and <7

- Set LoadersEmptier’s filter expression (now that FourthToSeventhEmptiest has been defined). Note that the filter expression references the FuelLeftPct VarProp of the loader cursored by FourthToSeventhEmptiest.

It is important to observe that in this example, both FourthToSeventhEmptiest and LoadersEmptier cursor loaders in LoadersWait. This may be confusing if one does not realize that each Filter has its own independent cursor.

The following pseudo-code illustrates the procedure by which Stroboscope determines the value of ‘LoadersWait.FourthToSeventhEmptiest.FuelNeededAmt.SumVal’.

```
nLD = LoadersWait.CurCount
TotalFuelNeeded = 0
For I = 1 to nLD
  If NeedsFuel(I) then
    LoadersEmptierCount = 0
    For J = 1 to nLD
```

Copyright © 1996 Julio C. Martinez
If FuelLeftPct(J)<FuelLeftPct(I) then
    LoadersEmptierCount=LoadersEmptierCount+1
End if
Next J
If LoadersEmptierCount>=3 AND LoadersEmptierCount<7 then
    TotalFuelNeeded=TotalFuelNeeded+FuelNeededAmt(I)
End if
End if
Next I

Notice in the above pseudo-code that both $I$ and $J$ are used as indexes into properties of the loaders in $LoadersWait$. $I$ is analogous to $FourthToSeventhEmptiest$’s cursor. $J$ is analogous to $LoadersEmptier$’s cursor.

### 12.3 Recap

- Stroboscope “simulates” the movement of generic resources using arithmetic.

- Characterized resources are real; they actually move from node to node carrying all the information attached to them.

- A Queue’s Discipline determines the position at which entering resources are placed in the Queue.

- The default Queue Discipline results in FIFO behavior.

- The ordering of resources maintained by a Queue’s Discipline is adequate for most cases.

- By default, a Combi draws resources from the front of a Queue.

- Combis can draw from the back of Queues if the RevOrder flag of the link connecting the Queue to the Combi is set.

- The DrawOrder attribute of Characterized Draw Links allow Combis to draw resources from Queues in any dynamic order.

- The DrawWhere attribute of Characterized Draw Links allow Combis to determine which resources are eligible for drawing.
• Characterized Draw Links do not have a DrawAmount attribute. The number of characterized resources drawn is controlled through the DrawUntil attribute.

• Characterized Release Links can control the order in which resources are released through their ReleaseOrder attribute.

• Characterized Release Links can control which resources are released through their ReleaseWhere attribute.

• Characterized Release Links can determine when to stop releasing through their ReleaseUntil attribute.

• An Activity instance can only release the resources it holds. In order to release more resources than those originally held, Activity instances must “generate” resources before they draw or before they end.

• The GENERATE pre-defined action target can create characterized resources while the simulation is running.

• Filters are cursoring objects that restrict the set of resources used for a resource count or for aggregate property access.

• Two or more Filters can be used together to provide very powerful set selection capabilities.
Chapter 13

Compound Resources

Resources often need to be combined and act as a group. For example, a truck and the earth loaded onto the truck travel through a system as one entity, “a loaded truck”.

Stroboscope allows us to combine resources into a compound resource; to add new resources to a previously existing compound resource; to remove some of the components from a compound resource; or to break up a compound resource into its constituents.

Compound resources are a kind of characterized resources. (The other kind, seen in the previous chapters, are simple characterized resources.) As a result, compound resources can be characterized (i.e., their properties can be defined and given appropriate values) according to their components, or to the state of the simulation at any significant point in their lifetime.

For example, a flatbed (a resource of type FlatBed) and steel shapes (resources of type SteelShape) can be grouped into a “loaded flatbed” (a compound resource of type LoadedFlatBed). The loaded flatbed can then be treated as a single resource. The loaded flatbed can be characterized by defining GrossWeight as a LoadedFlatBed SaveProp (assuming that the gross weight of the loaded flatbed is of interest). When the loaded flatbed is created, its GrossWeight SaveProp can be assigned an appropriate value. As time passes, some of the steel shapes can be removed and the value of GrossWeight updated accordingly. Similarly, more steel shapes or other resources such as rebar can be
added, and the value of *GrossWeight* updated again. Eventually, the loaded flatbed may be completely broken up into a plain flatbed and whatever steel shapes or rebar it carries.

In Stroboscope, resources can form a “containment” hierarchy. Compound resources and other non compound resources can be combined to form a single compound resource. This compounding can continue as needed. Eventually, a high level compound resource can be broken up. If any of the constituents are themselves compound resources, then these can also be broken up and so on.

It may be helpful to think of compound resources as grouped graphic images in a typical modern drawing program. Several images can be grouped into one. The resulting group, as well as other images or groups, can be grouped to form yet another group. The group at the top of the hierarchy can later be broken up into its original constituents; and those constituents that are themselves groups can also be broken up.

This chapter introduces compound resources and the modeling elements with which they can be manipulated. The first section describes compound resources and resource types as well as issues regarding statistics. The second section describes the Assembler and Disassembler auxiliary nodes and how they are used to assemble and disassemble compound resources. The third section presents a very detailed description of the sequence of events that occur during the termination of Activity instances; the section summarizes issues related to Assemblers and Disassemblers, as well as issues related to Forks, Disassemblers, Queues, Consolidators, and Normals. The fourth section contains several examples that show compound resources in use. The last section is a recap.

### 13.1 Compound Types, Resources and Statistics

In view of the existence of compound resources, the characterized resources mentioned in the preceding two chapters are “simple characterized resources”. The kind of resources discussed in this chapter are “compound characterized resources”. **The term “characterized resources” refers to both the simple and compound variety.**

Compound characterized resource types are defined as follows:
Syntax: \texttt{COMPTYPE \texttt{CompCharType};}
Example: \texttt{COMPTYPE \texttt{LoadedFlatBed};}

The example defines \textit{LoadedFlatBed} as a compound characterized resource type. Notice that, in contrast to the \texttt{CHARTYPE} statement, 	exttt{COMPTYPE} does not accept optional arguments for the \texttt{SubType} properties. This is because the concept of \texttt{SubType} is not useful for compound characterized resources and has been omitted. As a result, simple characterized resources must belong to a \texttt{SubType}, while compound characterized resources cannot. It thus follows that compound resource types cannot have \texttt{SubType} properties.

13.1.1 Compound vs. Simple Characterized Resources

Simple and compound characterized resources are very similar (they are both characterized resources). They both have the \texttt{TimeIn}, \texttt{ResNum}, and \texttt{BirthTime} predefined properties; and can have \texttt{VarProps} and \texttt{SaveProps} (these are defined and used in the same way for both simple and compound characterized resources).

An interesting question is when to use simple and when to use compound resources. The answer is that in many cases it makes little difference and is a matter of convenience. When compared to simple characterized resources, compound resources give up \texttt{SubTypes} in exchange for the ability to contain other resources within them.

Simple characterized resources are convenient when resources have fixed properties that can be represented in tabular format. The typical example is for resources that represent construction equipment. In these cases, the \texttt{SubTypes} correspond to equipment models; the \texttt{SubType} properties are obtained directly from equipment handbook tables.

In cases where \texttt{SubTypes} are not needed, it may be convenient to use compound instead of simple characterized resources (even if the resources will not be used as containers for other resources). The advantage is that the definition of the type is sufficient, whereas with simple characterized resources both the type and a \texttt{SubType} need to be defined.
The SubType is a required argument for the initialization of Queues that hold simple characterized resources. Since SubTypes do not exist for compound Characterized Queues, the syntax for initializing compound Characterized Queues is different:

Syntax: INIT CompCharQueue PositiveIntExpression;
Example: INIT LoadedFlatBedsWait 7;

Notice that the SubType is not used. The example creates 7 “loaded flatbeds” and places them in LoadedFlatBedsWait. Obviously, “loaded flatbeds” created in this manner are empty (i.e., they are not composed of other resources). They contain no flatbeds, or steel shapes, or anything else.

When generating simple characterized resources, the SubType is also required (as an argument in actions applied to the GENERATE action target). In the case of compound resources, the compound type is used instead of the SubType:

Syntax: BEFOREEND Activity GENERATE [PRECOND LogExp] number CompType;
Example: BEFOREEND CreateLoadedFlatBed GENERATE 1 LoadedFlatBed;

The example generates one “loaded flatbed” every time instances of CreateLoadedFlatBed terminate (before releasing resources). As in the case of Queue initialization, the loaded flatbed created in this manner is “empty”.

13.1.2 Compound Resource Statistics

In terms of statistics and pre-defined variables, compound and simple characterized resource types are the same except for visit statistics. Recall that visit statistics were prototyped as follows:

Hotel.Customer.{AveEnts|AvTotTm|AvVstTm|SDAvVst|MnVstTm|MxVstTm}

The use of Hotel instead of Node, and of Customer instead of SubType is not coincidental. This is because a compound characterized resource type can be both. A compound type acts as a Hotel when a compound resource contains other resources. It
acts as a *Customer* when it visits a node or when it is a component of another compound resource.

For example, if the *TransportShapes* Activity uses compound resources of type *LoadedFlatBed*, *LoadedFlatBed* is a *Customer* and *TransportShapes* is a *Hotel*. The variable “*TransportShapes.LoadedFlatBed.AvVstTm*” returns the average of the times it took a loaded flatbed to transport shapes.

Let us assume that the components of a resource of type *LoadedFlatBed* are simple characterized resources of type *FlatBed* and *SteelShape*. In this case, *LoadedFlatBed* is a *Hotel* and the various SubTypes of *FlatBed* and *SteelShape* are *Customers*. For example, if *W127X14* is a SubType of *SteelShape*, “*LoadedFlatBed.W127X14.AvVstTm*” returns the average of the times that a *W127X14* steel shape spent inside a loaded flatbed.

### 13.2 Compound Resource Assembly and Disassembly

There are three ways to create compound resources. The first is by using the INIT statement. The second is by performing an action on the GENERATE action target. These two methods do not “assemble” various resources into a compound resource. The third method is through resource assembly, which is the process of putting together several resources into a single compound resource. Resource assembly also refers to the case where resources are added to a compound resource that already exists. In any case, the idea of assembly implies that a compound resource “contains” or “has as parts” other resources.

Resource disassembly is the process of breaking up a compound resource into its components. Resource disassembly also refers to removing some of the components of a compound resource, without destroying the compound resource and the remainder of its content.

Stroboscope defines two kinds of auxiliary nodes to assemble and disassemble compound resources. (A generic definition of auxiliary nodes and their characteristics appears in sections 9 and 9.3.1.) These new nodes are called Assembler and Disassembler.
Assemblers and Disassemblers are HeteroHolders. The reason should be obvious. Assemblers can receive and temporarily hold resources of various types before creating a compound resource. Disassemblers can extract resources of various types from a compound resource and temporarily hold them until they are released.

Since Assemblers and Disassemblers are HeteroHolders, the properties of the resources they hold are accessible as if the resources were held by an Activity instance. The following pre-defined global variable can be used to determine if an Assembler or Disassembler is “in context” (i.e., it is assembling or disassembling).

\[ \text{HeteroHolder.InContext} \]

If an Assembler or Disassembler is indeed in context, then the following “instance” variables can access the resources it holds:

\[ \text{HeteroHolder.Filter.Property.}[\text{AveVal}|\text{MaxVal}|\text{MinVal}|\text{SDVal}|\text{SumVal}] \]
\[ \text{HeteroHolder.Filter.Count} \]
\[ \text{HeteroHolder.CharType.Property.}[\text{AveVal}|\text{MaxVal}|\text{MinVal}|\text{SDVal}|\text{SumVal}] \]
\[ \text{HeteroHolder.ResourceType.Count} \]
\[ \text{HeteroHolder.SubType.Count} \]

Notice that these are the exact same prototypes used to access Activity instance variables (those related to resources). \textit{HeteroHolder} in the variables above is the name of the Assembler or Disassembler in question.

Assemblers and Disassemblers are cursoring nodes. When an Assembler is assembling, it cursors the compound resource being assembled. When a Disassembler is disassembling it cursors the compound resource being disassembled. The following predefined global variable can be used to determine if an Assembler or Disassembler is “cursoring” a resource:

\[ \text{Assembler.HasCursor} \]
\[ \text{Disassembler.HasCursor} \]

When an Assembler or Disassembler is “cursoring”, the properties of the compound resource being assembled or disassembled are accessible using cursored resource syntax as follows:

\[ \text{[Assembler.]}\text{Property} \]
\[ \text{[Disassembler.]}\text{Property} \]
The Assembler and Disassembler names enclosed in brackets in the above variables need not be specified explicitly (thus they are optional) in situations where Stroboscope expects access to the cursored resource. (These situations will be discussed later as appropriate.) The square brackets themselves are not part of the variables. (See section 11.4 for a detailed explanation of explicit versus implicit cursored property access.)

The resources that are contained inside a compound resource are accessible only in the following situations:

- When the compound resource is in the Assembler that just created it.
- When the compound resource is in an Assembler that is adding new components to the compound resource.
- When the compound resource is in a Disassembler that removes some components from it.
- When the compound resource is in a Disassembler that will completely break it up into its components.

It is at these times that a compound resource’s SaveProps should be updated (those that reflect what is inside the compound resource). SaveProps whose values do not depend on the quantity or properties of its components can be updated whenever the compound resource is cursored, or when it is the only resource of the type held by a node.

The following two subsections discuss the particulars of Assemblers and Disassemblers.

### 13.2.1 The Assembler

Assemblers are shown in a network drawing with a small circle that encloses the letter A. The name of the Assembler is placed on top. Assemblers are defined in a simulation model source file with the ASSEMBLER statement:

Syntax: `ASSEMBLER AssemblerName CompCharType;`

Example: `ASSEMBLER BuildLoadedFlatBed LoadedFlatBed;`
AssemblerName is the name of the Assembler. CompCharType is the type of the compound resources that the Assembler creates. The example defines BuildLoadedFlatBed as an Assembler. BuildLoadedFlatBed assembles resources into compound resources of type LoadedFlatBed.

Assemblers can receive resources of any generic, simple or compound characterized type. Thus, the links that enter an Assembler can be of any type. Only the resources created by the Assembler can leave it. Thus, the link that leaves an Assembler must be of the same compound type as the Assembler (Assemblers must have exactly one outgoing link).

Assemblers can either create a new compound resource, or add more components to one that already exists. Which one of these two happens depends on whether the Assembler receives a compound resource that serves as an “Assembly Base”.

An “Assembly Base” is simply a compound resource, of the type assembled by the Assembler, that enters the Assembler through a special kind of link called “Assembly Base Link”. Assembly Base Links are distinguished from other links in a network drawing because they have double arrowheads. They are defined as follows:

Syntax: ASMBASELINK LinkName Predecessor Assembler;
Example: ASMBASELINK LF2 PickUpShapes AttachToFlatBed;

Note that the definition of an Assembly Base Link does not require a resource type. The link determines the type from the Assembler. The example defines LF2 as an Assembly Base Link. LF2 goes from the PickUpShapes Activity to the AttachToFlatBed Assembler.

In all other respects, Assembly Base Links are like ordinary links and retain their attributes and function as implied by their predecessor node. For example, if the predecessor to an Assembly Base Link is a Fork or Dynafork, the Assembly Base Link has a Strength attribute. If the predecessor to an Assembly Base Link is an Activity instance, then it has ReleaseUntil, ReleaseWhere, RevOrder, and ReleaseOrder attributes.

Assemblers work in a very straightforward manner:

- When a predecessor Activity instance terminates, but before it releases its resources, the Assembler is put in context and gets ready to receive resources.
• As the Assembler receives resources through regular links (i.e., not through an Assembly Base Link), it does not pass them on to the successor. The Assembler simply holds them. The count or properties of these resources are available as if they were in an Activity instance.

• If a compound resource is received through an Assembly Base Link:
  • Stroboscope verifies that no other resource has been received through an Assembly Base Link. If an assembly base had already been received, Stroboscope issues a runtime error.
  • The Assembler cursors the compound resource it received and makes it the assembly base.
  • The Assembler extracts all the components from the assembly base. Then it places these resources in front of any other resources (that the Assembler had received through regular links).
  • The properties of the assembly base are accessible through cursored resource access syntax (e.g., Assembler.Property or Property). The assembly base itself is not included in the count or properties accessed via HeteroHolder syntax (e.g., Assembler.CharType.Property.SumVal or Assembler.Type.Count). The components of the assembly base, however, are included in such variables (together with any previously received resources).

• When the terminating Activity instance finishes releasing, the Assembler checks to see if it had received an assembly base. If not, the Assembler creates a new compound resource and cursors it. The created resource is now the assembly base.

• The Assembler performs its OnAssembly cursored actions (to be described below).

• The Assembler packs all the resources it holds and “puts them inside” the assembly base.
• The Assembler uncursors the assembly base.

• The resource leaves through the outgoing link towards the successor node.

Note that an Assembler creates a compound resource only if it does not receive an assembly base through an Assembly Base Link. Also note that if more than one assembly base enters the Assembler, Stroboscope stops the simulation and issues a runtime error. There is no relationship between these facts and the number of Assembly Base Links that go into the Assembler. An Assembler can have any number of Assembly Base Links coming in. This is fine as long as none of them bring in a resource (in this case the Assembler creates the resource), or only one of them brings one in.

Before an Assembler packs resources into the compound resource it has either created or received, the Assembler performs its OnAssembly actions. The main purpose of the OnAssembly actions is to assign values to the SaveProps of the assembled resource (which is cursored at this point). For this reason, expressions or action targets used in OnAssembly actions have implicit access to the properties of the assembled resource. OnAssembly actions are defined with the ONASSEMBLY statement:

Syntax: \[
\text{ONASSEMBLY Assembler [ActionVerb] ActionTarget [PRECOND LogicalExp] [TargetArg1 TargetArg2 ...];}
\]

Example: \[
\text{ONASSEMBLY BuildLoadedFlatBed ASSIGN GrossWeight \ 'BuildLoadedFlatBed.FlatBed.Weight+ BuildLoadedFlatBed.SteelShape.Weight.SumVal';}
\]

The example assumes that \text{GrossWeight} is a SaveProp of \text{LoadedFlatBed}. It defines an action that sets the value of the \text{LoadedFlatBed’s GrossWeight} SaveProp. The value is the weight of the flatbed and the steel shapes being assembled. Notice that, during the OnAssembly action, the components being assembled are accessed as if the Assembler were an Activity instance in context. The compound resource that results from the assembly is accessed as a cursored resource.
13.2.2 The Disassembler

Disassemblers complement Assemblers. They are shown in a network drawing with a small circle that encloses the letter D. The name of the Disassembler is placed on top. Disassemblers are defined in a simulation model source file with the DISASSEMBLER statement:

Syntax:  \text{DISASSEMBLER} \text{DisassemblerName} \text{CompCharType};
Example: \text{DISASSEMBLER} \text{BreakUpLoadedFlatBed} \text{LoadedFlatBed};

\text{DisassemblerName} is the name of the Disassembler. \text{CompCharType} is the type of the compound resources that the Disassembler breaks up or extracts from. The example defines \text{BreakUpLoadedFlatBed} as a Disassembler. \text{BreakUpLoadedFlatBed} disassembles \text{LoadedFlatBed}s into their components (e.g., \text{FlatBed}s and \text{SteelShape}s).

Disassemblers can only receive resources whose type matches the compound type of the Disassembler. Although several links can enter a Disassembler, they must all carry resources of the matching type. Disassemblers release resources of any type. Thus, the links that leave a Disassembler can be of any generic, simple or characterized type.

Disassemblers can completely break up and then destroy a compound resource. They can also extract some of the components without destroying the original resource and any un-extracted resources. Whether a Disassembler destroys the original compound resource or not, depends on the existence of a “Disassembly Base Link”.

A “Disassembly Base Link” is a special link that leaves a Disassembler. When this link exists, a Disassembler connected to the tail does not destroy the compound resource it disassembles. Instead, the resource retains any unreleased components and leaves through the Disassembly Base Link. Disassembly Base Links are shown on a network drawing with a filled circle in the tail. Disassembly Base Links are defined as follows:

Syntax:  \text{DISASMBASELINK} \text{LinkName} \text{Disassembler} \text{Successor};
Example: \text{DISASMBASELINK} \text{LF3} \text{DetachSomeShapes} \text{LoadedFBContinues};

Note that the definition of a Disassembly Base Link does not require a resource type. The link determines its type from the Disassembler. The example defines \text{LF3} as a
Disassembly Base Link. *LF3* goes from the *DetachSomeShapes* Disassembler to the *LoadedFBContinues* Activity.

Regardless of the successor connected by the Disassembly Base Link, Disassembly Base Links have no attributes except for OnFlow actions.

The order in which a Disassembly Base Link is defined is irrelevant. Stroboscope will always consider a Disassembly Base Link as if it were defined after any of the other links that leave the Disassembler (even if it appears before the others in the input file).

When a Disassembler receives a resource, it breaks up the resource into its components. It then releases these components in a process that is very much like the release process of terminating Activity instances. Not surprisingly then, the links that leave Disassemblers are Release Links (see section 12.1.3 for a detailed explanation of Release Links and their attributes). Note that Disassembly Base Links are an exception, they are not considered Release Links.

Recall that characterized resources flow from node to node one at a time. Thus, Disassemblers receive compound resources one at a time. This is true even in the case where a terminating Activity instance releases several compound resources to a Disassembler. As the Disassembler receives each resource, it breaks up the resource and releases the resulting components as follows:

- The Disassembler puts itself in context and cursors the compound characterized resource it receives. The resource is the disassembly base.

- The properties of the disassembly base are accessible through cursored resource access syntax.

- The Disassembler detaches the disassembly base from its components. The count and property of the components are now accessible as if they were the resources held by an Activity instance.

- The Disassembler releases all the components through its outgoing Release Links in exactly the same manner that Activity instances release their resources. Note that the Disassembly Base Link, if any, is not a Release Link.
• The Disassembler performs its OnDisassembly cursored actions (to be described below).

• The Disassembler uncursors the disassembly base and puts itself out of context.

• The Disassembler re-attaches any unreleased components to the assembly base. Note that any released generic resources were actually created by the Release Link. Thus, the original generic resource components remain in the Disassembler and are included in those re-attached.

• If a Disassembly Base Link exists, the disassembly base (with all the resources that were re-attached) is sent to the successor connected by the Disassembly Base Link.

• If a Disassembly Base Link does not exist, the Disassembler destroys the disassembly base and everything it contains.

Before a Disassembler destroys or releases its disassembly base, it performs its OnDisassembly actions. The main purpose of the OnDisassembly actions is to assign values to the SaveProps of the resource being disassembled (which is cursored at this point). For this reason, expressions or action targets used in OnDisassembly actions have implicit access to the properties of the disassembly base. OnDisassembly actions are defined with the ONDISASSEMBLY statement.

Syntax:  
ONDISASSEMBLY Disassembler [ActionVerb] ActionTarget  
[PRECOND LogicalExp]  
[TargetArg1 TargetArg2 ...];

Example:  
ONDISASSEMBLY DetachSomeShapes ASSIGN GrossWeight  
'DetachSomeShapes.FlatBed.Weight+  
DetachSomeShapes.SteelShape.Weight.SumVal';

The example assumes that GrossWeight is a SaveProp of LoadedFlatBed, and DetachSomeShapes is a Disassembler of type LoadedFlatBed. The statement defines an action that updates the value of the LoadedFlatBed’s GrossWeight SaveProp. The value is the weight of the flatbed and the steel shapes that remain after the Disassembler releases other shapes (assuming it does not release the flatbed). Notice that, during the
OnDisassembly action, the components being disassembled are accessed as if the Disassembler were an Activity instance in context. The disassembly base is accessed as a cursored resource.

OnDisassembly actions are rarely used when a Disassembler does not have a Disassembly Base Link. This is because it doesn’t make sense to update the SaveProp of a resource that Stroboscope will destroy right away. OnDisassembly actions that collect statistics, print, or assign values to SaveValues or Arrays may make sense regardless of whether a Disassembly Base Links exists.

### 13.2.3 Dual Base Links

A link that enters an Assembler is made an Assembly Base Link by defining it with the ASMBASELINK statement. A link that leaves a Disassembler is made a Disassembly Base Link by defining it with the DISASMBASELINK statement. Thus, using these statements it is impossible to define a link that is simultaneously a Disassembly Base Link and an Assembly Base Link. This requires a new statement, the DUALBASELINK:

**Syntax:**

```
DUALBASELINK LinkName Disassembler Assembler;
```

**Example:**

```
DUALBASELINK LF5 DetachSomeShapes AttachOtherShapes;
```

A Dual Base Link connects a Disassembler to an Assembler. It is a Disassembly Base Link and an Assembly Base Link at the same time. In Dual Base Links, some components are removed in the Disassembler, and then some new components are attached in the Assembler. The compound resource type disassembled by the Disassembler must match the type assembled by the Assembler. The example defines LF5 as `DetachSomeShapes`’s Disassembly Base Link, and as `AttachOtherShapes`’s Assembly Base Link.

### 13.3 Sequence of Actions During Activity Terminations

During the termination of an Activity instance, resources flow through the links and auxiliary nodes in the terminating Activity’s release graph. The resources eventually
reach the terminal nodes of the graph (Queues, Normals or Consolidators). Also during the termination of an Activity instance, the Assemblers that form part of the release graph assemble and the Normals start.

In certain situations it may be necessary to have an in-depth understanding of the sequence in which resources flow, Assemblers assemble, and Activity instances start. Most of the time the precise sequence does not matter. When it does matter, it is usually obvious and intuitive.

The sequence of events may matter in situations where the expressions used to define attributes such as Duration or Strength, or action events such as OnStart or OnAssembly, depend on the location of certain resources. Figure 32 shows an arbitrarily complex release graph. All the nodes and links shown are part of TerminatingActivity’s release graph. If the expression for the duration of Term4 depends on the properties of the resources released through L7, how are these properties accessed? Are the resources in Assembler Aux5, Assembler Aux8, or Normal Term13?

In order to determine the sequence of events in the rare cases in which it does matter, it is necessary to examine the auxiliary nodes in the graph and the order in which the links in the graph were defined. It is also necessary to understand very precisely the series of messages that terminating Activity instances broadcast, and how the different nodes in a release graph react to them.

This discussion will refer to the release graph for TerminatingActivity in Figure 32. The release graph shown is extremely complex. Since the assemblies and disassemblies implied by Assemblers and Disassemblers usually take up some time, such a release graph will rarely, if ever, be used to model a real system. Models for complex cases are likely to contain several Normals, each with its own duration and simple release graph. In order to eliminate unnecessary complexity, the release graph of Figure 32 does not include Forks. (See section 9.3.1 for release graphs involving Forks.) Note that whenever a node has more than one outgoing link, the relative order of definition of the links is shown with superscripts.

The very first step performed by a terminating Activity instance is to put itself in context, thus allowing access to the resources it holds (through instance variables). Next, the Activity executes its BeforeEnd actions. After performing the BeforeEnd actions, the
Activity decrements its current instance count. At this point the Activity starts to broadcast messages as described below. Messages are broadcast to successors in an order that corresponds to the order in which the links that connect to the successors are defined. Thus, TerminatingActivity sends messages to Term1, Aux2, Aux5, Aux6 (twice), Aux11, Aux10 and Term15 (twice). The messages are sent in the order listed.

13.3.1 The “Get In Context” Message

The first message that an Activity broadcasts is the “Get In Context” message. Each node reacts to the “Get In Context” message differently. Most nodes simply get ready to receive resources and, as implied by the message, get in context. Additionally, most auxiliary nodes broadcast the message to their successors. Thus, for example, when
Aux2 receives the message, it in turn sends a “Get In Context” message to Aux3 and Aux5.

The only auxiliary nodes that do not broadcast the “Get In Context” message to all their successors are the Forks. A Fork relays the message to only one of its successors. A Fork ignores its other successors until the next time that a predecessor Activity terminates. Section 9.3 covers this procedure in detail.

The “Get In Context” message, and well as all the other messages discussed here, is not complete until the node that receives the message reacts to it. For example, TerminatingActivity waits until Aux2 processes the “Get In Context” message before it sends the “Get In Context” message to Aux6.

If a node has more than one predecessor, it can receive the “Get In Context” message more than once. Aux8, for Example, will receive the “Get In Context” message from Aux5 and Aux7. Nodes only react to the first “Get In Context” message. They ignore subsequent messages received during the same predecessor Activity instance termination.

As a result of the above rules, nodes receive the “Get In Context” message in the order in which the nodes are encountered in a depth-first traversal of the release graph. The following outline shows the order in which the various nodes process the “Get In Context” message:

- Term1
- Aux2
  - Aux3
    - Term4
    - Term13
  - Aux5
  - Aux8
    - Term13 (ignored)
- Aux5 (ignored)
• Aux6
  • Aux7
  • Aux8 (ignored)
  • Term9
  • Term12
• Aux6 (ignored)
• Aux11
  • Term12 (ignored)
• Aux10
  • Aux11 (ignored)
  • Term13 (ignored)
  • Term14
• Term15
• Term15 (ignored)

13.3.2 The “Receive” Message

Resources flow from node to node as part of “Receive” messages. Each “Receive” message carries with it a certain amount of generic resource or one specific characterized resource. A terminating Activity instance sends “Receive” messages to its successors as determined by the Activity’s release process (see sections 6.3 and 12.1.3). Thus, in the case of Characterized Release Links, the number of “Receive” messages sent to the successor varies (could be zero, one, or several). The number of “Receive” messages sent through Generic Release Links is always one (even if the corresponding amount is zero). In all cases, a terminating Activity finishes releasing through one link completely before starting to release to another.
Each node reacts differently to the “Receive” message:

- Forks relay the message to the successor chosen during the broadcast of the “Get In Context” message.

- Dynaforks choose a successor for every “Receive” message, and relay the message to the chosen successor.

- Disassemblers break up the resource received with the message, and dispatch the components in a process similar to the release process of Activities (see section 13.2.2). If they have an outgoing Disassembly Base Link, the last thing they do is to send the compound resource they received to the successor connected by the Disassembly Base Link.

- Assemblers hold the resource and do not relay the message at this point.

The links in Figure 32 are numbered in an order that resembles the order in which resources flow through them (and thus the order in which successor nodes receive “Receive” messages). When Characterized Release Links dispatch more than one resource, sequences of links are traversed several times. If link $L_2$ releases 2 resources, for example, one resource (and its disassembly components) flows through $L_2, L_3, L_4, L_5$, and $L_6$. The other resource repeats in the same manner. Thus, the second resource flows through $L_2$ after some resources have flowed through $L_4$.

When TerminatingActivity has finished broadcasting the “Receive” message (i.e., it has finished releasing), resources have flowed through links $L_1$ to $L_{16}$. Notice that at this point resources have not flowed through links $L_{17}$ to $L_{23}$. Resources are now held by terminal nodes Term1, Term4, Term13, Term14, and Term15; or by Assemblers Aux3, Aux5, Aux6, and Aux11. Aux8 has not received resources because Aux5 and Aux6 have not yet assembled the resources it may receive. Term9 has not received resources because Aux6 has not yet assembled the resource it may receive. Term12 has not received resources because Aux6 and Aux11 have not yet assembled the resources it may receive. Although Term13 holds resources, it still has not received the resource that will eventually flow through $L_{20}$. 
When a terminating Activity finishes broadcasting “Receive” messages, the successor (direct or indirect) Normals have not determined their duration or performed their OnStart actions; and the Assemblers that are part of the release graph have not assembled. Consolidators may or may not have consolidated any number of times as they received resources (see section 10.2).

13.3.3 The “Stop Receiving” Message

Once the terminating Activity instance performs its release process (by broadcasting “Receive” messages), it starts to broadcast “Stop Receiving” messages. In contrast to the “Get In Context” message, nodes only react to the last “Stop Receiving” message they will get. (Nodes keep a count of the number of “Get In Context” messages they have received and use the count to determine which is the last “Stop Receiving” message.)

Only Assemblers and Normals react to the “Stop Receiving” message. Assemblers react by assembling all the resources held, attaching them to the assembly base (which is created if not received through an Assembly Base Link), performing the OnAssembly actions, and then sending the assembled resource to the successor (via a “Receive” message). The Assembler concludes by sending a “Stop Receiving” message to the successor. Normals react by creating an instance of themselves. This involves determining the instance duration, incrementing the current and total instance count, inserting the instance in the FEL (Future Events List), and performing the OnStart actions.

When a node receives the last “Stop Receiving”, the node is “complete”. The nodes in Figure 32 are numbered in the order in which they “complete”. Thus, for example, *Aux10* is complete before *Term13*. Since “complete” translates to “assembles” for Assemblers and to “create instance” for Normals, *Aux10* assembles before the instance of *Term13* is created, but after the instance of *Term4* is created.

The following outline illustrates the broadcast of the “Stop Receiving” message:

- *Term1* receives “Stop Receiving” (ignored)
- *Aux2* receives “Stop Receiving”
• *Aux3* receives “Stop Receiving”
  • *Term4* receives “Stop Receiving” and creates instance
  • *Term13* receives “Stop Receiving” (ignored)
• *Aux5* receives “Stop Receiving” (ignored)
• *Aux5* receives “Stop Receiving” and assembles
  • *Aux8* receives the resource assembled by *Aux5*
  • *Aux8* receives “Stop Receiving” (ignored)
• *Aux6* receives “Stop Receiving” (ignored)
• *Aux6* receives “Stop Receiving” and assembles
  • *Aux7* receives the resource assembled by *Aux6*
  • *Aux7* chooses whether to send the resource to *Aux8, Term9, or Term12*
    • The chosen successor receives the resource assembled by *Aux6*
  • *Aux7* receives “Stop Receiving”
  • *Aux8* receives “Stop Receiving” and assembles
    • *Term13* receives the resource assembled by *Aux8*
    • *Term13* receives “Stop Receiving” (ignored)
  • *Term9* receives “Stop Receiving” (ignored)
  • *Term12* receives “Stop Receiving” (ignored)
• *Aux11* receives “Stop Receiving” (ignored)
• *Aux10* receives “Stop Receiving”
  • *Aux11* receives “Stop Receiving” and assembles
    • *Term12* receives the resource assembled by *Aux11*
• *Term12* receives “Stop Receiving” and creates instance

• *Term13* receives “Stop Receiving” and creates instance

• *Term14* receives “Stop Receiving” and creates instance

• *Term15* receives “Stop Receiving” (ignored)

• *Term15* receives “Stop Receiving” and creates instance

At this point all the Assemblers in the release graph have assembled and all the Normals have instantiated.

### 13.3.4 The “Get Out of Context” Message

As a final step, the terminating Activity instance performs the OnEnd actions for the Activity. After the OnEnd actions are performed, the terminating Activity broadcasts the “Get Out of Context” message.

Nodes only react to the first “Get Out of Context” message they receive. As the name implies, auxiliary nodes get out of context, destroy any resources they still have (if any), and then broadcast the message to the successors. Normals simply get out of context. Queues and Consolidators ignore the message.

### 13.4 Examples

Compound resources enhance Stroboscope’s modeling power significantly. The several subsections that follow present various model fragments that illustrate different aspects of compound resources, Assemblers, Disassemblers, Assembly Base Links, Disassembly Base Links, and Dual Base Links.

The most natural and straightforward compound resources are actual containers such as boxes, pallets, packets, and shipping containers. The purpose of the following examples is to illustrate the resource assembly and disassembly mechanisms in Stroboscope. For this reason, the examples use several fictitious resources that allow us
to package items, collect several packages in boxes, and several boxes into shipping containers.

Since the release process of a terminating Activity instance takes zero time, the number of auxiliary nodes that follow an Activity (the Activity’s “release graph”) is usually small. Complex resource release cases are usually split into several Activities (and their corresponding release graphs) so that the appropriate delay is modeled at each transformation. For example, it is not reasonable to assume that it takes zero time to put tiles into a tile box, then put the tile box inside a bigger box, and finally put the bigger box in a shipping container. Such a process should probably be modeled with a separate Activity and Assembler for each assembly step.

The release graphs in some of the examples presented in the following subsections can be considered unrealistically complex. Their purpose is to illustrate the assembly and disassembly mechanisms available in Stroboscope.

In spite of this “warning”, Stroboscope is very well suited for situations that do require complex release graphs. There is no limit to the number of auxiliary nodes. The only requirement is that the release graph for an Activity be acyclic.

The following resource type definitions are common for all the example fragments that follow:

/laborer - LB
GENTYPE Laborer;

/bulk cement - CM
GENTYPE Cement; /in kg
VARIABLE CementDensity 2400; /in kg/m3

/cement in bags - CB
COMPTYPE CementBag;
SAVEPROPS CementBag Weight; /in kg
VARPROP CementBag Volume Weight/CementDensity; /in m3

/tile - TL
CHARTYPE Tile RefNum Width Length Thickness Density;
/ unique cm cm cm kg/m3
SUBTYPE Tile CeramicTile 10021 4 4 0.65 875;
SUBTYPE Tile GraniteMosaic 10027 6 6 1.25 1125;
VARPROP Tile Weight Width*Length*Thickness*Density/100^3;/in kg

/box - BX
COMPTYPE Box;
/ m m m kg;
SAVEPROPS Box Width Length Height Weight;
13.4.1 Simple Assembly of All Released Resources

Assumptions: The terminating instance of Activity PackTile contains 1 resource of type Box and several resources of type Tile. By design, all of the Tile resources fit inside the Box resource. CreateTileBox is an Assembler of type TileBox (can be determined by looking at TB1).

Links BX1, TL1, and TB1 retain their default attributes. Hence, all the resources of type Tile will be released through TL1 and the resource of type Box will be released through BX1. The tiles and box are held in CreateTileBox until PackTile has released all its resources. At that point, a compound resource of type TileBox is created. This resource contains all the tiles and the box held by CreateTileBox.

Note that the TileBox is a new resource that “contains”, but is separate from, the Box and Tiles. The TileBox is a logical construct that represents the combination of the Box and Tiles. Thus, in this case, the Box and Tiles contained in the TileBox are “siblings”.

Once CreateTileBox releases the TileBox, the count and properties of the resources that compose it are inaccessible. The TileBox needs to be characterized while held by CreateTileBox so that any relevant information related to its components is visible. The OnAssembly actions defined below the network drawing assign appropriate
values to the SaveProps of the TileBox. Notice that the action targets are the SaveProps of the resulting TileBox and they use the implicit short form (i.e., Count instead of CreateTileBox.Count although both would be the same in this case).

After the OnAssembly actions are performed, the TileBox is sent though TB1 to DelayTileBox. DelayTileBox then starts.

### 13.4.2 Simple Assembly to Base

Assumptions: The terminating instance of PackTile contains 1 resource of type Box and several resources of type Tile. All of the Tile resources fit inside the Box resource. AttachTilesToBox is an Assembler of type Box (can be determined by looking at BX2).

```
/Update Characteristics of the Box after tiles are attached
ONASSEMBLY AttachTilesToBox ASSIGN Weight
  AttachTilesToBox.Tile.Weight.SumVal;
```

In this example, a resource of type TileBox is not created. Instead, the tiles are put inside the resource of type Box that comes in from the terminating PackTile instance.

Link BX1 is an Assembly Base Link. The box that comes through BX1 should already have appropriate values assigned to its properties (i.e., they are not all zero). The box possibly contains other tiles in addition to those that flow through TL1; those tiles that were contained in the box, if any, join those that enter through TL1.

When PackTile has finished releasing, AttachTilesToBox performs its OnAssembly actions. In this case, there is only one action; the action updates the box’s Weight SaveProp. Note that the variable used as an action argument, ‘AttachTilesToBox.Tile.Weight.SumVal’, represents the sum of the weights of all the tiles. Those that were already contained in the box and those released through TL1.

Two variations of this example are interesting.

First, if the terminating instance of PackTile holds two resources of type Box, then two boxes reach AttachTilesToBox. When the second box reaches
AttachTilesToBox, Stroboscope issues a runtime error indicating that AttachTilesToBox received more than one assembly base.

Second, if the terminating instance of PackTile holds no resources of type Box, then no boxes reach AttachTilesToBox. When PackTile finishes releasing its resources, AttachTilesToBox creates a new resource of type Box. The OnAssembly actions assign a value to the box’s Weight SaveProp. The other SaveProps of the box, however, are not initialized with appropriate values; they are all zero.

**13.4.3 Simple Assembly of Some of the Released Resources**

Assumptions: The terminating instance of PackTile contains 1 resource of type Box, several resources of type Tile, and one resource of type Laborer. All of the Tile resources fit inside the Box resource. CreateTileBox is an Assembler of type TileBox (can be determined by looking at TB1).

```
/Characterize the TileBox after assembled
ONASSEMBLY CreateTileBox ASSIGN Count CreateTileBox.Tile.Count;
ONASSEMBLY CreateTileBox ASSIGN Width CreateTileBox.Box.Width;
ONASSEMBLY CreateTileBox ASSIGN Height CreateTileBox.Box.Height;
ONASSEMBLY CreateTileBox ASSIGN Length CreateTileBox.Box.Length;
ONASSEMBLY CreateTileBox ASSIGN Weight CreateTileBox.Tile.Weight.SumVal;
```

This example is almost exactly the same as the example of section 13.4.1. The difference is that now PackTile holds a laborer resource that it releases to DelayTileBox. This resource is transferred to DelayTileBox without being assembled into a TileBox. In this case, DelayTileBox starts with two resources, a TileBox and a Laborer.

Note that CreateTileBox holds the box and tiles until PackTile has finished releasing all the resources it holds. By the time CreateTileBox assembles, the laborer has already been released and is in DelayTileBox.
13.4.4 Parallel Assembly

Assumptions: The terminating instance of PackTileAndCement contains 1 resource of type Box, several resources of type Tile, one resource of type CementBag, and a certain amount of resource of type Cement. The tiles fit inside the box. CreateTileBox is an Assembler of type TileBox. The cement fits inside the cement bag. PutCementInBag is an Assembler of type CementBag, and CM1 is an Assembly Base Link.

```
/Characterize the TileBox after assembled
ONASSEMBLY CreateTileBox ASSIGN Count CreateTileBox.Tile.Count;
ONASSEMBLY CreateTileBox ASSIGN Width CreateTileBox.Box.Width;
ONASSEMBLY CreateTileBox ASSIGN Height CreateTileBox.Box.Height;
ONASSEMBLY CreateTileBox ASSIGN Length CreateTileBox.Box.Length;
ONASSEMBLY CreateTileBox ASSIGN Weight CreateTileBox.Tile.Weight.SumVal;

/Update Characteristics of the Cement Bag
ONASSEMBLY PutCementInBag ASSIGN Weight PutCementInBag.Cement.Count;
```

This example is a superset of the example in section 13.4.1. The terminating Activity instance additionally releases cement and a cement bag. The PutInCementBag Assembler puts the cement it receives into the cement bag. The cement bag may contain cement before it reaches the Assembler. The variable PutCementInBag.Cement.Count includes any cement previously inside the cement bag as well as the cement that comes through link CM1.

In this example, Delay starts with a resource of type TileBox and a resource of type CementBag.

13.4.5 Assembly on Top of Parallel Assembly

Assumptions: The terminating instance of PackTileAndCement contains one resource of type Box, several resources of type Tile, one resource of type CementBag, a certain amount of resource of type Cement, and one resource of type Container. The tiles
fit inside the box. *CreateTileBox* is an Assembler of type *TileBox*. The cement fits inside the cement bag. *PutCementInBag* is an Assembler of type *CementBag*, and *CB1* is an Assembly Base Link. *PutBagAndBoxInContainer* is an Assembler of type *Container*. The cement bag and the tile box fit inside the container. *CT1* is an Assembly Base Link and the container that comes through it may already have cement bags or tile boxes in it.

```
/Characterize the TileBox after assembled
ONASSEMBLY CreateTileBox ASSIGN Count CreateTileBox.Tile.Count;
ONASSEMBLY CreateTileBox ASSIGN Width CreateTileBox.Box.Width;
ONASSEMBLY CreateTileBox ASSIGN Height CreateTileBox.Box.Height;
ONASSEMBLY CreateTileBox ASSIGN Length CreateTileBox.Box.Length;
ONASSEMBLY CreateTileBox ASSIGN Weight CreateTileBox.Tile.Weight.SumVal;

/Update Characteristics of the Cement Bag
ONASSEMBLY PutCementInBag ASSIGN Weight PutCementInBag.Cement.Count;

/Update Characteristics of the Container
ONASSEMBLY PutBagAndBoxInContainer ASSIGN CurWeight
              PutBagAndBoxInContainer.CementBag.Weight.SumVal+
              PutBagAndBoxInContainer.TileBox.Weight.SumVal;
```

This example is an extension of the example in the previous subsection. Instead of *Delay* starting with a *TileBox* and a *CementBag*, it starts with a *Container*. The container holds the cement bag and the tile box. It also holds any other cement bags and tile boxes that were already in the container.

### 13.4.6 Assembly with Components of the Same Compound Type

Assumptions: The terminating instance of *PackTileAndCement* contains two resources of type *Box*, several resources of type *Tile*, one resource of type *CementBag*, and a certain amount of resource of type *Cement*. One of the boxes is small and is used to package the loose tiles; the loose tiles do fit inside it. *PutTilesInBox* is an Assembler of type *Box*, and *BX1* is an Assembly Base Link. The box that comes through *BX1* may already contain other tiles. The cement fits inside the cement bag. *PutCementInBag* is an
Assembler of type CementBag, and CB1 is an Assembly Base Link. 
PutBagAndBoxInBigBox is an Assembler of type Box. The cement bag and the box with tiles fit inside the big box. BX3 is an Assembly Base Link and the box that comes through it may already have cement bags or other boxes in it.

This example is a variation of the example in the previous subsection. Instead of putting a tile box and a cement bag into a container, a small box and a cement bag are put into a big box. The big box holds the cement bag and the small box containing tiles. It also holds any other cement bags or boxes that were already in the big box.

BX3’s attributes make sure that it releases only the big box. The ReleaseWhere makes sure that the first box to be released is the big one. The ReleaseUntil makes sure that only one box is released. The ReleaseUntil checks whether PutBagAndBoxInBigBox is already cursoring its assembly base in order to determine if a box has already been released to it by PackTileAndCement. BX1’s attributes do not need to be modified. BX1 will simply release any boxes still held by PackTileAndCement (after BX3 has released).
Notice *PutBagAndBoxInBigBox*’s OnAssembly actions. The action target is the *Weight* SaveProp of the box that entered through *BX3*. The action argument is an expression with two terms. The first term is simply the weight of the cement bags. The second term is the total weight of the boxes that will be put inside the big box. The weight of the big box (the assembly base), even though a resource of type *Box*, is not included in the variable *PutBagAndBoxInBigBox.Box.Weight.SumVal*.

### 13.4.7 Simple Disassembly

Assumptions: The terminating instance of *DelayTileBox* contains one resource of type *TileBox*. This is the resource it acquired in the example of subsection 13.4.1. *BreakUpTileBox* is a Disassembler of type *TileBox* (can be determined from *TB2*).

When *BreakUpTileBox* receives a *TileBox*, it extracts the box and tiles. The box is released through *BX2* and the tiles are released through *TL2*. Since there is no Disassembly Base Link, the compound resource of type *TileBox* is destroyed. The box and tiles released are the exact same resources that where assembled into the *TileBox*.

If the terminating instance of *DelayTileBox* holds several resources of type *TileBox*, they are released one at a time through *TB2*. *BreakUpTileBox* extracts and releases the box and tiles from each of the tile boxes it receives (before any other resources are released to *BreakUpTileBox* through *TB2*).

Notice that no code accompanies the network. This is because the default attributes of the links are sufficient to model the behavior described.

### 13.4.8 Simple Disassembly With Release Control

Assumptions: The terminating instance of *DelayTileBox* contains one resource of type *TileBox*. The tile box contains one box and several tiles. Some tiles are of SubType
CeramicTile and others are of SubType GraniteMosaic. This is similar to the example of the previous subsection, except that the different kinds of tiles follow different routes. BreakUpTileBox is a Disassembler of type TileBox. It first sends the CeramicTile tiles to CeramicTiles and then the GraniteMosaic tiles to GraniteTiles.

When BreakUpTileBox receives a “tile box”, it extracts the box and tiles. The box is released through BX2. TL2’s ReleaseWhere ensures that only the ceramic tiles flow towards CeramicTiles. There are no non-default attributes for TL3, so all the remaining tiles flow through it (these must be granite mosaics). Since there is no Disassembly Base Link, the “tile box” is destroyed.

If the terminating instance of DelayTileBox holds several resources of type TileBox, they are released one at a time through TB2. BreakUpTileBox extracts the box and tiles from each of the tile boxes it receives.

13.4.9 Simple Disassembly From Base

Assumptions: The terminating instance of DelayTileBox contains one resource of type TileBox. The tile box contains one box and several tiles. Some tiles are of SubType CeramicTile and others are of SubType GraniteMosaic. This is similar to the example of the previous subsection, except that only the ceramic tiles are extracted. The original tile box and the granite tiles it contains are passed on to the starting instance of BoxWithGraniteDelay. ExtractCeramicTiles is a Disassembler of type TileBox. Link TB3 is a Disassembly Base Link. ExtractCeramicTiles extracts the CeramicTile tiles from the tile box and sends them to CeramicTiles. The tile box and any GraniteMosaic tiles it had are sent to BoxWithGraniteDelay.
/Route the Ceramic Tiles
RELEASEWHERE TL2 RefNum==CeramicTile.RefNum;

/Update characteristics of TileBox after ceramic tiles are removed
ONDISASSEMBLY ExtractCeramicTiles ASSIGN Weight
    ExtractCeramicTiles.Tiles.Weight.SumVal;

When ExtractCeramicTiles receives a “tile box”, it extracts the box and tiles. TL2’s ReleaseWhere ensures that only the ceramic tiles flow towards CeramicTiles. Since TB3 is a Disassembly Base Link, the unreleased tiles and box are reattached to the “tile box”. Before releasing the tile box, its Weight SaveProp is updated. The tile box and its contents are then released through TB3. An instance of BoxWithGraniteDelay then starts with a “tile box”.

If the terminating instance of DelayTileBox holds several resources of type TileBox, they are released one at a time through TB2. ExtractCeramicTiles extracts the ceramic tiles from each of the tile boxes and sends them to CeramicTiles. The original tile boxes and their content are sent to BoxWithGraniteDelay. The instance of BoxWithGraniteDelay that starts, will do so holding all the tile boxes released by DelayTileBox.

### 13.4.10 Generic Resource Disassembly From Base

Assumptions: The terminating instance of DelayCementBag contains one resource of type CementBag. The cement bag contains one bag and a certain amount of cement. Half of the cement in the bag is sent to CementPowder. UnloadSomeCement is a Disassembler of type CementBag and CB2 is a Disassembly Base Link. The bag continues with all the cement it contained to Delay. (Even though it appears to have unloaded half of the cement to CementPowder.)
RELEASEAMT CM1 UnloadSomeCement.Cement.Count/2;

The ReleaseAmount for CM1 tells the link to create an amount of cement equal to one half of the cement held by UnloadSomeCement, and to send the cement to CementPowder. The amount of cement held by UnloadSomeCement is not altered with the release. (This is consistent with the way Generic Release Links out of Activities release resources.) For this reason, UnloadSomeCement re-attaches the same amount of Cement originally held by the CementBag (to the CementBag).

The decision to design Stroboscope so that Generic Release Links create resources instead of moving (transfering) them, implies that the amount of generic resource held by a compound resource can never decrease. This is true regardless of whether the compound resource enters a Disassembler or not. This limitation can be overcome easily by using a SaveProp to keep track of the amount of generic resource carried by the compound resource.

### 13.4.11 Chained Disassembly

Assumptions: The terminating instance of DelayContainer contains one resource of type Container. This is the resource assembled in the example of subsection 13.4.5. BreakUpContainer is a Disassembler of type Container and BreakUpTileBox is a Disassembler of type TileBox.
When BreakUpContainer receives a container, it extracts the tile boxes and the cement bags from the container. The cement bags are released one by one to CementBags (because CB3 is defined before TB2). After all the cement bags have been released, the tile boxes are released one by one to BreakUpTileBox. As BreakUpTileBox receives each “tile box”, it extracts the box and tiles. The tiles are sent to Tiles, and then the box is sent to Boxes.

It is not necessary to set attributes for any of the links. The behavior described above happens by default.

If DelayContainer contained several containers, these would be released one by one to BreakUpContainer. BreakUpContainer will not receive a second container until it has released all the components of the first. If the first container holds several “tile boxes”, each one of the tile boxes will be released one by one to BreakUpTileBox and so on. The following pseudo-code further illustrates the release process of a terminating instance of DelayContainer.

```
For I=1 to Number Of Containers In DelayContainer
    Send Container(I) to BreakUpContainer
    For J=1 to Number Of Cement Bags In Container(I)
        Send CementBag(J) in Container(I) to CementBags
    Next J
    For J=1 to Number of Tile Boxes in Container(I)
        Send TileBox(J) of Container(I) to BreakUpTileBox
        For K=1 to Number of Tiles in TileBox(J) of Container(I)
            Send Tile(K) in TileBox(J) of Container(I) to Tiles
        Next K
        Send Box in TileBox(J) of Container(I) to Boxes
        Destroy TileBox(J)
    Next J
    Destroy Container(I)
Next I
```

Thus, for example, all the Tiles within all the TileBoxes in the first container reach the Tiles Queue before BreakUpContainer receives a second container.

**13.4.12 Disassembly With Components of the Same Type as the Container**

Assumptions: The terminating instance of DelayBigBox contains one resource of type Box. This is the same resource finally assembled in the example of section 13.4.6.
The *Box* contains one or more small boxes (these are also of type *Box*) and one or more Cement bags.

When *BreakUpBigBox* receives the big *Box*, it extracts the smaller boxes and cement bags from it. The small boxes that were contained in the big box go to *SmallBoxes*. The cement bags go to *CementBags*. The big *Box*, which now contains no cement bags nor smaller boxes inside, goes to *BigBox*. All this is achieved without any code because it is the default behavior. The code shown simply marks the new weight of the big box as being zero and is not necessary to route the resources.

Notice that *BX5* is a Release Link, whereas *BX6* is a Disassembly Base Link. The small boxes that were contained in the big box can only flow through *BX5*. The only resource that can and will flow through *BX6* is the big box released through *BX4*. *BreakUpBigBox* cannot release small boxes through *BX6* nor the big box through *BX5*.

### 13.4.13 Assembling Disassembly Components

Assumptions: The terminating instance of *DelayBoxesAndTileBoxes* contains one resource of type *TileBox* and several resources of type *Box*. The *TileBox* contains a resource of type *Box* and possibly some resources of type *Tile*. The resources of type *Box* may contain resources of type *Tile* and a certain amount of *Cement*. *RemoveTilesFromPlainBoxes* is a Disassembler of type *Box* and *AttachTiles* is an Assembler of type *TileBox*. Link *TB1* is an Assembly Base Link. Link *BX2* is a Disassembly Base Link.
When `RemoveTilesFromPlainBoxes` receives a box, it extracts the tiles and sends them one at a time to `AttachTiles`. The `Weight` SaveProp of the box is then updated, and the box is sent to `Delay`. `AttachTiles` breaks up and holds the `TileBox` it receives through `TB1` into a box and tiles. It also holds the tiles it receives through `TL1`. When `DelayBoxesAndTileBoxes` finishes releasing through all its outgoing links, `AttachTiles` updates the `Weight` SaveProp of the “tile box” and re-attaches all the tiles and the box (to the “tile box”). The tile box is then sent to `Delay`, which starts with the tile box and plain boxes released by `DelayBoxesAndTileBoxes`. Notice that the effect of the entire release process is to move all the resources from `DelayBoxesAndTileBoxes` to `Delay`, and to transfer all the tiles that used to be in plain boxes to the “tile box”.

### 13.4.14 Parallel Dual Base Links

Assumptions: The terminating instance of `DelayTwoTileBoxes` contains two resources of type `TileBox`. Each one of the two tile boxes contains a mixture of ceramic tiles and granite mosaics.

The purpose of the auxiliary node graph rooted at `DelayTwoTileBoxes` is to rearrange the content of the tile boxes so that the ceramic tiles are in one box and the granite mosaics are in the other.
/Release one box through TB1 and the other through TB4
/Assume link TB1 is defined before link TB4
RELEASEUNTIL TB1 DetachCeramicTiles.HasCursor;

/Release only CeramicTiles through TL1
RELEASEWHERE TL1 RefNum==CeramicTile.RefNum;

/Release only GraniteMosaics through TL2
RELEASEWHERE TL2 RefNum==GraniteMosaic.RefNum;

/Update characteristics of ceramic tile box
ONASSEMBLY AttachCeramicTiles ASSIGN Weight
AttachCeramicTiles.Tiles.Weight.SumVal;

/Update characteristics of granite mosaic box
ONASSEMBLY AttachGraniteMosaics ASSIGN Weight
AttachGraniteMosaics.Tiles.Weight.SumVal;

One “tile box” is released through TB1. The ceramic tiles within it are extracted in DetachCeramicTiles and sent to AttachCeramicTiles. The box and the un-extracted granite mosaics are re-attached to the “tile box” and sent to AttachGraniteMosaics (thus the tile box released through TB1 becomes the assembly base of AttachGraniteMosaics).

The other tile box is released through TB4. The granite mosaics within it are extracted in DetachGraniteMosaics and sent to AttachGraniteMosaics. The box and the un-extracted ceramic tiles are re-attached to the tile box and sent to AttachCeramicTiles (thus the tile box released through TB4 becomes the assembly base of AttachCeramicTiles).

The granite mosaics that reached AttachGraniteMosaics through TL2 are put together with those already in the tile box and leave through TB3. Thus the tile box that reaches Delay through TB3 contains the granite mosaics it originally contained, plus the granite mosaics that were originally contained in the other tile box (which was released through TB4 and will enter Delay through TB6).

The ceramic tiles that reached AttachCeramicTiles through TL1 are put together with those already within the tile box and leave through TB6. Thus the tile box that
reaches Delay through TB6 contains the ceramic tiles it originally contained, plus the ceramic tiles that were originally contained in the other tile box (which was released through TB1 and will enter Delay through TB3).

Delay then starts with the two tile boxes. One containing granite mosaics and the other ceramic tiles.

### 13.4.15 A Practical Example

The following example fragment is typical of how compound resources, Assemblers, and Disassemblers, can be used in practice. Figure 33 is a network fragment for a bus stop at a commuter parking lot in a large civil construction site. The resources involved here are of compound type Bus and Worker. Workers are modeled as compound resources for convenience, not because they will serve as “containers”.

Buses circulate through several commuter parking lots and work areas (these are bus stops). Bus stops are identified with unique numbers. The bus stop that corresponds to this fragment is represented by variable TheBusStop.

![Figure 33 - Network Fragment for Bus Stop](image)

VARIABLE TheBusStop 4; /assume this is bus stop # 4
COMPTYPE Bus;
SAVEPROPS Bus Capacity Content NumToUnloadNow;
VARPROP Bus IsFull Content==Capacity;

COMPTYPE Worker;
SAVEPROPS Worker Source Destination;

/*Assume that the bus released by BusTravelsFromPrevStop has updated
values in the Capacity and Content SavePros, but that the
NumToUnloadNow SaveProp needs to be updated.

The purpose of the LookInsideBox Disassembler is to update the value
of NumToUnloadNow. There is actually no Disassembly going on.
We need the assistance of a Filter for this.
FILTER WorkersToUnloadNow Worker Destination==TheBusStop;
ONDISASSEMBLY LookInsideBus ASSIGN NumToUnloadNow
    LookInsideBus.WorkersToUnloadNow.Count;

/CheckIfAnyToUnload is a Dynafork and not a Fork because it needs to
/choose a branch based on the properties of the resource that flows
/(and thus cannot choose before BusTravelsFromStop releases, which a
/Fork would do)

/Note that ReadyToUnload and UnloadWorker cannot be substituted for
/a Normal because then the Normal would always start, even if the
/bus is not routed towards it.

/Route Bus to UnloadWorker if any worker needs to get off
STRENGTH B3 NumToUnloadNow;
STRENGTH B7 !NumToUnloadNow;

/Unloading a worker takes time
DURATION UnloadWorker Normal[0.5,0.1];

/Detach only one worker (of those that need to get off at this station)
RELEASEWHERE P1 Destination==TheBusStop;
RELEASEUNTIL P1 WorkerLeaves.Worker.Count;

/Update SavePros of Bus after Worker is detached
ONFLOW B6 ASSIGN Content  Content-1;
ONFLOW B6 ASSIGN NumToUnloadNow  NumToUnloadNow-1;

/Give LoadWorker a high priority and allow it to load as long as it can:
/There are people to load and space on the Bus. Otherwise
/BusTravelsToNextStop starts and removes the Bus from BusWaits.
/A Filter is required as an aid
PRIORITY LoadWorker 10; /*any positive number will do
FILTER BusesWithSpace Bus !IsFull;
ENOUGH B8 BusWaits.BusesWithSpace.Count;
DRAWHERE B8 !IsFull;

/*Assume that somehow workers enter WorkersWait and their
/SavePros have appropriate values (i.e., Source==TheBusStop).
/*Note that one worker is loaded at a time, and thus any workers
/*that enter WorkersWait after the bus has started loading still have
/*a chance to get in.

/Loading a worker takes time
DURATION LoadWorker Normal[0.7,0.15];
13.5 Recap

- Compound resources are characterized resources that can “contain” any number of other resources. The “contained” resources can be dissimilar and of any type: generic, simple characterized, or compound.

- A compound resource can contain resources of its own type.

- Compound resources can form a “containment” hierarchy that is a deep as necessary.

- Characterized resources that are not compound are called simple characterized resources.

- While simple characterized resources must belong to a SubType, compound characterized resources cannot.

- In cases where subtyping is not useful, it may be more convenient to use compound resources instead of simple characterized resources.

- Components can be added to, or removed from, a compound resource after the resource is created.

- To add or remove resources from a compound resource, the compound resource must not be contained in another compound resource.

- Compound resources can be characterized by their SaveProps.

- The SaveProps of a compound resource can be updated to reflect the number or properties of the resources it contains (whenever resources are added or removed from the resource).
• In terms of visit statistics, compound resource types can be both “Hotels” and “Customers”.

• Assemblers and Disassemblers are HeteroHolders. They are also cursoring nodes.

• Assemblers are auxiliary nodes that create a compound resource from other resources, or that attach resources to a previously existing assembly base.

• When an Assembler receives a resource through an Assembly Base Link, the Assembler does not create a new compound resource. Instead, the Assembler attaches the resources that enter through the other links to the assembly base.

• An Assembler’s OnAssembly actions are appropriate for updating the SaveProps of the compound resource being assembled.

• Disassemblers are auxiliary nodes that extract the resources contained in a compound resource.

• When a Disassembler has an outgoing Disassembly Base Link, the Disassembler does not destroy the compound resource it disassembles. Instead, the Disassembler releases the resource through the Disassembly Base Link.

• The links that leave a Disassembler, with the exception of the Disassembly Base Link (if any), are Release Links. They release resources as if the Disassembler were an Activity instance.

• The compound resource that the Disassembler receives and disassembles (the disassembly base) cannot be released through a Release Link even if a Release Link of the appropriate type exists.

• The Disassembly Base Link will only release the disassembly base. Even if some of the components of the disassembled resource match the type of the Disassembly Base Link, these components cannot flow through the Disassembly Base Link.

• The Disassembly Base Link, if present, is always considered last, after the other outgoing links of a Disassembler. This is true regardless of the order in which the Disassembly Base Link was defined.
• When a Disassembler has an outgoing Disassembly Base Link, the Disassembler’s OnDisassembly actions are appropriate for updating the SaveProps of the compound resource being disassembled.

• By studying the release graph of the terminating Activity instance, it is possible to determine the precise sequence in which resources flow, Assemblers assemble, and Normals instantiate.
Chapter 14

Flow Control Statements

The tools discussed so far provide no control over the order in which Stroboscope executes statements. In the models presented, Stroboscope simply executes the statements in the order in which they appear in the input source file. This chapter introduces several new statements that control the order in which Stroboscope executes the statements in a source file.

Sections 5.1.1 (Element Definition Statements), 5.1.2 (Attribute Statements), and 5.1.3 (Control Statements), discuss the different kinds of statements used in previous chapters. The statements introduced in this chapter belong to a different category. They are “Flow Control Statements”.

As their name indicates, flow control statements control the order in which Stroboscope executes the statements that appear in an input source file. Flow control statements can be used in two different kinds of flow structures: While-Wend blocks, and IF blocks; or, in the case of the statement that ends the model, anywhere.

14.1 Ending Model Processing

The ENDMODEL flow control statement can be used anywhere, but only makes sense within a While-Wend or IF block:

Syntax:    ENDMODEL;
Example:   ENDMODEL;
When Stroboscope executes the ENDMODEL statement, it simply stops processing the source file. The ENDMODEL statement is used in the example that illustrates the IF block.

### 14.2 IF Blocks

IF blocks are used for the conditional execution of a series of statements. An IF block begins with the IF statement:

**Syntax:**
```
IF LogicalIfExpression;
```

**Example:**
```
IF ServOrder==LIFO;
```

and ends with the ENDIF statement:

**Syntax:**
```
ENDIF;
```

**Example:**
```
ENDIF;
```

The statements in an IF block can be grouped into clauses by using the ELSEIF and ELSE statements:

**Syntax:**
```
ELSEIF LogicalElseIfExpression;
```

**Example:**
```
ELSEIF ServOrder==BIGFIRST;
```

**Syntax:**
```
ELSE;
```

**Example:**
```
ELSE;
```

Any number of other statements can appear in each of the clauses. The general layout of an IF block is as follows:

```
IF LogicalIfExpression;
    /Statements in the IF clause
        .........
ELSEIF LogicalElseIfExpression1; /this statement is optional
    /Statements in the first ELSEIF clause
        .......
ELSEIF LogicalElseIfExpression2; /this statement is optional
    /Statements in the second ELSEIF clause
        .......
ELSEIF LogicalElseIfExpressionN; /this statement is optional
    /Statements in the Nth ELSEIF clause
        .......
ELSE; /this statement is optional
    /Statements in the ELSE clause
        .......
ENDIF;
```
An IF block can contain any number of optional ELSEIF clauses. These must appear after the IF clause (obviously), but before the optional ELSE clause.

Stroboscope will execute the statements in, at most, one of the clauses within the IF block. Which one of the clauses is executed depends on the results of evaluating LogicalIfExpression or any of the LogicalElseIfExpressions.

When Stroboscope executes an IF statement it evaluates LogicalIfExpression. If the result is TRUE, Stroboscope executes the statements in the IF clause and then continues with the statements that appear after the ENDIF statement. If the result of LogicalIfExpression is FALSE, Stroboscope continues with the first ELSEIF, ELSE, or ENDIF it finds.

The execution of an ELSEIF statement is similar to the execution of an IF statement. Stroboscope evaluates LogicalElseIfExpression. If the result is TRUE, Stroboscope executes the statements in the ELSEIF clause and then continues with the statements that appear after the ENDIF statement. If the result of LogicalElseIfExpression is FALSE, Stroboscope continues with the next ELSEIF, ELSE, or ENDIF it finds.

When Stroboscope executes an ELSE statement, it simply executes the statements in the ELSE clause. Note that Stroboscope only executes an ELSE statement if none of the preceding clauses were executed.

Stroboscope takes no action when it executes an ENDIF statement. The only purpose of the ENDIF statement is to mark the end of the IF block.

The following example is an extension to the example of section 11.5.1. It illustrates how an IF block can be used to model different configurations based on the value of a variable:

/ these statements should be added to the beginning of the file
VARIABLE BigScrapersFirst 1;
VARIABLE SmallScrapersFirst 2;
VARIABLE LIFO 3;
VARIABLE FIFO 4;

/ Define ServOrder as equal to one of BigScrapersFirst, SmallScrapersFirst, LIFO, or FIFO
VARIABLE ServOrder LIFO;
IF ServOrder==BigScrapersFirst;
    DISPLAY "Big scrapers are served first";
    DISCIPLINE Scrprs -Cap;
ELSEIF ServOrder==SmallScrapersFirst;
    DISPLAY "Small scrapers are served first";
    DISCIPLINE Scrprs Cap;
ELSEIF ServOrder==LIFO;
    DISPLAY "Scrapers are served in reverse order of arrival";
    REVORDER SC1;
ELSEIF ServOrder==FIFO;
    DISPLAY "Scrapers are served in order of arrival";
ELSE;
    DISPLAY "Error in the definition of ServOrder, you should define";
    DISPLAY "ServOrder to equal one of the following values: ";
    DISPLAY "BigScrapersFirst, SmallScrapersFirst, LIFO, or FIFO";
    DISPLAY "Simulation processing will stop now";
    ENDMODEL;
ENDIF;

Notice that if ServOrder has not been defined with an acceptable value, the statements in the ELSE clause display some error information and terminate model execution.

Since in this example ServOrder is defined as LIFO, Stroboscope executes the "ELSEIF ServOrder==LIFO" clause. Stroboscope does not even look at the statements in the other clauses.

14.3 While-Wend Blocks

While-Wend blocks are used to “loop” over the series of statements enclosed within the block. A While-Wend block begins with the WHILE statement:

Syntax: WHILE LogicalWhileExpression;
Example: WHILE CurItem<nItems;

and ends with the WEND statement:
Syntax:    WEND;
Example:   WEND;

Any number of other statements can appear between the While and the Wend. These statements form the “body” of the block.

When Stroboscope executes a WHILE statement, it first evaluates $LogicalWhileExpression$. If the result is FALSE, Stroboscope skips the remainder of the statements in the block and continues processing by executing the statement that follows the Wend statement.

If the result is TRUE, Stroboscope executes the statements in the body of the block in the order in which it encounters them. After Stroboscope executes the statements in the body of the block, it goes back to the While statement and executes it again. Thus Stroboscope re-evaluates $LogicalWhileExpression$ and either re-executes the statements in the body of the block, or skips them and executes the statements that appear after the block.

The effect is that Stroboscope executes the statements in the body of the While-Wend block until $LogicalWhileExpression$ returns FALSE (when evaluated during the execution of the While statement).

The following example prints the numbers from 1 to 10 and their corresponding square roots:

```
PRINT StdOutput ”Num   Sqrt
”;
PRINT StdOutput ”===========
”;
SAVEVALUE CurNum 1;
WHILE CurNum<=10;

PRINT StdOutput ”%3.0f %7.4f
”
    CurNum
    Sqrt[CurNum];

ASSIGN CurNum CurNum+1;

WEND
```

In the example, the logical expression ‘$CurNum<=10$’ is used to control the loop. Stroboscope will continuously execute the body of the block until $CurNum$ is larger than 10. Note that value of $CurNum$ is incremented at the end of every loop. Otherwise,
Stroboscope will enter into an endless loop, executing the body of the block over and over.

While the example does not run a simulation, it constitutes a perfectly valid and complete input file. If submitted to Stroboscope for processing, Stroboscope would print a table that looks as follows:

<table>
<thead>
<tr>
<th>Num</th>
<th>Sqrt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0000</td>
</tr>
<tr>
<td>2</td>
<td>1.4142</td>
</tr>
<tr>
<td>3</td>
<td>1.7321</td>
</tr>
<tr>
<td>4</td>
<td>2.0000</td>
</tr>
<tr>
<td>5</td>
<td>2.2361</td>
</tr>
<tr>
<td>6</td>
<td>2.4495</td>
</tr>
<tr>
<td>7</td>
<td>2.6458</td>
</tr>
<tr>
<td>8</td>
<td>2.8284</td>
</tr>
<tr>
<td>9</td>
<td>3.0000</td>
</tr>
<tr>
<td>10</td>
<td>3.1623</td>
</tr>
</tbody>
</table>

Obviously, Stroboscope was not designed to substitute conventional programming languages. The real utility of flow control statements is to generate simulation code automatically and to perform multiple replications. These uses are the subject of subsequent chapters.

Two other flow control statements are available for use in While-Wend blocks; these are the CONTINUE and BREAK statements:

Syntax: CONTINUE;
Example: CONTINUE;

Syntax: BREAK;
Example: BREAK;

These statements can only be used in the body of a While-Wend block.

When Stroboscope executes the CONTINUE statement, it does not execute the remainder of the body. Instead, Stroboscope goes back to the beginning of the block and executes the corresponding WHILE statement again.

When Stroboscope executes the BREAK statement, it exits the While-Wend block. Processing continues with the first statement that appears after the end of the block (i.e., the statement immediately after the corresponding WEND statement).
The following example uses the CONTINUE and BREAK statements in a variation of the previous example.

```
PRINT StdOutput "Num    Sqrt\n";
PRINT StdOutput "==========\n";
SAVEVALUE CurNum 0;
WHILE 1;/always TRUE, loop must exit with a BREAK
  ASSIGN CurNum CurNum+1;
  IF CurNum>10;
    BREAK; /exit loop
  ENDIF;
  /do not show trivial numbers in table
  IF Sqrt[CurNum]==Int[Sqrt[CurNum]];
    CONTINUE; /skip rest of block and go to While again
  ENDIF;
  PRINT StdOutput "%3.0f %7.4f\n" CurNum  Sqrt[CurNum];
WEND;
```

This example is different in several ways. The While condition will always return TRUE. Therefore, the body of the block will be executed over and over unless the execution of a BREAK forces an exit of the loop. The value of CurNum is incremented at the beginning of the body so that it is not bypassed with the execution of a CONTINUE. When CurNum is larger than 10, the BREAK statement is executed and Stroboscope continues processing with the statement after the WEND (there is none in this example). Trivial cases, where the square root of a number is an integer, are not printed in the table. In these cases Stroboscope executes the CONTINUE and skips the remainder of the body (but goes back to the WHILE).

Note that the IF blocks are embedded in the While-Wend block. In general, IF and While-Wend blocks can be nested as necessary.

If submitted to Stroboscope for processing, Stroboscope would print a table that looks as follows:

<table>
<thead>
<tr>
<th>Num</th>
<th>Sqrt</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.4142</td>
</tr>
<tr>
<td>3</td>
<td>1.7321</td>
</tr>
<tr>
<td>5</td>
<td>2.2361</td>
</tr>
<tr>
<td>6</td>
<td>2.4495</td>
</tr>
<tr>
<td>7</td>
<td>2.6458</td>
</tr>
<tr>
<td>8</td>
<td>2.8284</td>
</tr>
<tr>
<td>10</td>
<td>3.1623</td>
</tr>
</tbody>
</table>
Chapter 15
Statement Preprocessing and Automatic Code Generation

This chapter introduces the Stroboscope statement preprocessor and describes how it can be used to generate code automatically. The first section introduces the preprocessor operator and the second section uses the operator in a complete example.

15.1 The Preprocessor Operator $<\text{Arg}>$

Before Stroboscope executes a statement, it searches the statement for a dollar sign “$” immediately followed by a less than sign “<”. If it finds the sequence “$<”, it treats what follows as a preprocessor replacement expression. The preprocessor replacement expression spans to, but does not include, the next greater than sign “>” immediately followed by a dollar sign “$” (both must exist). Stroboscope evaluates the preprocessor replacement expression, truncates the result, and uses it to replace the entire sequence that starts with “$<” and ends with “>$$”. Stroboscope then executes the statement.

The following two statements, for example, are identical:

```plaintext
COMBI MoveHoistFrom$$\log(100) * 5$$To$$<10+1>$$;
COMBI MoveHoistFrom10To11;
```
Both statements define a Combi and name it \textit{MoveHoistFrom10To11}. The preprocessor replacements in the first statement are shown in bold. The enclosed expressions that will be evaluated and substituted are underlined. In the second statement, the parts of the statement that are the result of pre-processing are shown in bold. Before Stroboscope executes the first statement it searches for preprocessor replacements and finds two of them. The first is \texttt{\$<Log[100]*5>\$"}. Stroboscope substitutes it with “10”. The second is \texttt{\$<10+1>\$"}. Stroboscope replaces it with “11”.

Preprocessor replacements are useful for the definition of repetitive network elements that follow a consistent pattern. Consider, for example, a model for high-rise construction where each floor is represented by an Activity succeeded by a Queue. For each floor, the associated Activity represents the work done by a crew. After the work is finished, the crew enters the Queue where it waits for an elevator to take it to another floor. The following code fragment illustrates how the Normals, Queues, and links connecting them could be defined for a 5 story building:

\begin{verbatim}
COMPTYPE Crew;
SAVEVALUE CurFlr 1;
WHILE CurFlr<=5;
    NORMAL WorkInFlr$<CurFlr>$;
    QUEUE WaitForElevAtFlr$<CurFlr>$ Worker;
    LINK CR$<CurFlr>$ _1 WorkInFlr$<CurFlr>$ WaitForElevAtFlr$<CurFlr>$;
    ASSIGN CurFlr CurFlr+1;
WEND;
\end{verbatim}

The code above defines 5 Normals, 5 Queues, and 5 links. The statements as they would be executed if the above code were submitted to Stroboscope for processing appear below (WHILE and WEND statements are omitted):

\begin{verbatim}
COMPTYPE Crew;
SAVEVALUE CurFlr 1;
NORMAL WorkInFlr1;
QUEUE WaitForElevAtFlr1 Crew;
LINK CR1_1 WorkInFlr1 WaitForElevAtFlr1;
ASSIGN CurFlr CurFlr+1;
NORMAL WorkInFlr2;
QUEUE WaitForElevAtFlr2 Crew;
LINK CR2_1 WorkInFlr2 WaitForElevAtFlr2;
ASSIGN CurFlr CurFlr+1;
NORMAL WorkInFlr3;
QUEUE WaitForElevAtFlr3 Crew;
LINK CR3_1 WorkInFlr3 WaitForElevAtFlr3;
ASSIGN CurFlr CurFlr+1;
\end{verbatim}
NORMAL WorkInFlr4;
QUEUE WaitForElevAtFlr4 Crew;
LINK CR4_1 WorkInFlr4 WaitForElevAtFlr4;
ASSIGN CurFlr CurFlr+1;
NORMAL WorkInFlr5;
QUEUE WaitForElevAtFlr5 Crew;
LINK CR5_1 WorkInFlr5 WaitForElevAtFlr5;
ASSIGN CurFlr CurFlr+1;

15.2 Example With Automatically Generated Code :

Shuttle Bus Problem

The example below builds upon the “bus-stop” example used to explain the assembly and disassembly of compound resources in section 13.4.15. Figure 34 shows the commuter transportation system for a large construction site.

Workers arrive at one of two commuter lots where they wait for the bus to take them to one of three work locations. For simplicity, it is assumed that only one bus is available. Workers spend a certain amount of time at that work location and then wait for the bus to pick them up and return them to the parking lot at which they had arrived originally. When the bus arrives at a bus stop, the workers that arrived at their destination unload one at a time. After all have unloaded, workers waiting to board enter the bus one at a time.

The arrivals at the two commuter lots are a Poisson process. The time spent by workers at the three work areas follow a normal distribution. The time for a worker to enter (load) and leave (unload) the bus are uniformly distributed.

Each commuter lot and work area has a bus stop. Loading and unloading of workers at each stop are very similar. Commuter lots differ from work areas slightly. Workers arrive from the “outside world” at commuter lots. When workers get off the bus at a work area they start working. After work, they wait for the bus to pick them up so they can go to the lot where they left their vehicle. When they get off a bus at a commuter lot they return to the “outside world”.

15.3 Solution

Figure 35 shows the fragment of this problem’s network that models commuter lot 2 (bus stop 4). Notice that the names for the network nodes and links contain the number 4 (which is the number of the bus stop). This network fragment shares BusFrom3To4 with the fragment that models bus stop 3. This network fragment also
shares **BusFrom4To5** with the fragment that models bus stop 5. The Normal named **WorkOrLeave4** represents the departure of the worker to the “outside world”.

Figure 36 shows the fragment of this problem’s network that models work area 3 (bus stop 5). It is very similar to Figure 35, except that the number 5 is embedded in the nodes and links instead of the number 4. Another difference is that Figure 36 does not
have an *InterArrive5 Combi* to feed *WorkersWait5*. Instead, workers enter *WorkersWait5* from *WorkOrLeave5*. The Normal named *WorkOrLeave5* represents the work performed by the worker in work area 3 (bus stop 5). *BusFrom4To5* appears in both Figure 35 and Figure 36.

Notice that *WorkOrLeaveX* represents work (i.e. work) or departure (i.e., leave) depending on whether the stop is a work area or a commuter lot. The dual purpose name allows the definition of the *WorkOrLeaveX* Normals and the links connected to them with the same statement.

In general, the five bus stops are very similar. The model takes advantage of this similarity to automatically generate the network definition and element attributes for all bus stops (using WHILE loops with preprocessor replacements).

In addition, solution is parametric so that it is easy to extend the model for a bus system with *N* commuter lots and *M* work areas. Furthermore, the model includes options that allow the creation of a “bus log” and a graph of the number of workers in the system as a function of time. The “bus log” and population graph will be described later on.

The general structure of the model is to use the first part for a definition of the system (from the perspective of someone using the model, not someone creating the model). The second part contains the implementation that generates the actual Stroboscope statements based on the data of the first section.

The model begins with the definition of variables that indicate which model options are in effect:

```plaintext
/**Statement Execution Switches***************************************************************************/
VARIABLE PrintBusLog 1; /1 to print a bus log and 0 otherwise
VARIABLE GraphPopulation 1; /1 for a population graph and 0 otherwise
```

The next part contains only two statements. They simply specify the seed for the default random number generator and the duration of the simulation.

```plaintext
/**Simulation Run Parameters***************************************************************************/
/SEED 9111964; /uncomment or replace as necessary
VARIABLE SimulationLength 10; / (days)
```

The next part defines the characteristics of the bus (or related to the bus):
/**Bus characteristics**************************************************
VARIABLE WorkerLoadTime Uniform[15,25]/3600; / (hours)
VARIABLE WorkerUnloadTime Uniform[16,24]/3600; / (hours)
VARIABLE BusCapacity 30;
VARIABLE BusSpeed 65; / (km/hr)

The next part defines the characteristics of the commuter lots:

/**Commuter Lot Data**************************************************
VARIABLE NumCommLots 2; /number of commuter lots
ARRAY MapCommLotsToStop NumCommLots {1 4};//Lots = stops 1 & 4

/Define one variable of form ArrivalRateAtLotX for each commuter lot,
/where X is the commuter lot number
VARIABLE ArrivalRateAtLot1 19;/ (Arrivals/Hour)
VARIABLE ArrivalRateAtLot2 15;/ (Arrivals/Hour)

/Define one variable of form DestIfArvAtLotX for each commuter lot,
/where X is the commuter lot number
VARIABLE DestIfArvAtLot1 'Rnd[] < 0.28 ? 2 : LastRnd[]<0.76 ? 3 : 5';
VARIABLE DestIfArvAtLot2 'Rnd[] < 0.35 ? 2 : LastRnd[]<0.56 ? 3 : 5';

MapCommLotsToStop is an Array that will be used to determine the bus stop number indexed by the commuter lot number. MapCommLotsToStop[1], for example, returns the bus stop number for commuter lot 2 (recall that Arrays are zero based). The code generator that appears later expects that variables of the form ArrivalRateAtLotN and DestIfArvAtLotN be defined for each commuter lot N. These variables indicate the arrival rate to the corresponding lot and the destination (i.e., bus stop) of the workers that arrive.

The next part of the model defines the characteristics of the work areas:

/**Work Area Data******************************************************
VARIABLE NumWorkAreas 3;
ARRAY MapWorkAreasToStop NumWorkAreas {2 3 5};//Work areas = stops 2,3,5

/Define one variable of form TimeAtWorkArea X for each area,
/where X is the work area number
VARIABLE TimeAtWorkArea1 Normal[8,0.5];// (hours)
VARIABLE TimeAtWorkArea2 Normal[7,0.75];// (hours)
VARIABLE TimeAtWorkArea3 Normal[9,0.35];// (hours)

The MapWorkAreasToStop Array is analogous to MapCommLotsToStop used in the commuter lot data. Similarly, the code generator expects that variables of the form TimeAtWorkAreaM be defined for each work area M. The variables indicate the amount of time that workers spend when they work in the area.
The user input data concludes with the distance between bus stops:

/**Distance Between Bus Stops***************************************************************************/
VARIABLE NumStops NumCommLots+NumWorkAreas;
ARRAY DistToNextStat NumStops {1.84 1.37 2.01 1.36 1.33}; / (km)

The definition of NumStops should not be modified by the user but appears here for convenience since it is necessary for the dimensioning of DistToNextStat. DistToNextStat is an Array that contains the distances between bus stops (indexed by bus stop number). For example DistToNextStat[3] returns the distance between bus stops 4 and 5 (recall that Arrays are zero based).

The model described so far contains all the data necessary to model the bus system as described. The statements that follow do not assume a specific number of commuter lots or work areas, or any other specific data such as arrival rates, working times, or distance between stations. As long as the first part of the model is prepared according to the guidelines described, the statements that follow should generate an appropriate network, run a simulation, and display output correctly. As a result, only the preceding statements need to be modified to add or remove bus stops, or to change any other model parameter. A banner makes this clear:

/**Implementation, Do not modify statements below this line*************/
=======================================================================

The implementation of the model begins with the definition of the resource types:

/**Resource Types***************************************************************************/
COMTYPE Bus;
SAVEPROPS Bus Speed Capacity Content NumToUnloadNow;
VARPROP Bus IsFull Content==Capacity;

COMTYPE Worker;
SAVEPROPS Worker Lot WorkArea Destination;

The model proceeds with the definition of some helper Variables, SaveValues, and Filters:

/**Helper SaveValues, Variables, and Filters***************************************************************************/
SAVEVALUE CurStop 1;
VARIABLE NextStop 'CurStop < NumStops ? CurStop+1 : 1';
VARIABLE PrevStop 'CurStop == 1 ? NumStops : CurStop-1';

SAVEVALUE CurCommLot 1;
SAVEVALUE CurWorkArea 1;

FILTER WorkersToUnloadNow Worker Destination==CurStop;
FILTER BusesWithSpace Bus !IsFull;

The model uses \textit{CurStop}, \textit{CurCommLot}, and \textit{CurWorkArea} several times as indexes in WHILE loops. In addition, the model uses \textit{CurStop} to store the current location of the bus as the simulation runs. \textit{NextStop} and \textit{PrevStop} are defined for convenience in the determination of the next and previous stop as a function of the current stop.

The model now defines the nodes and Collectors for all the bus stops:

/**Define Nodes and Collectors common to Commuter Lots and Work Areas***/

ASSIGN CurStop 1;
WHILE CurStop<=NumStops;
    DISASSEMBLER LookInsideBus$<CurStop>$ Bus;
    DYNAFORK CheckIfAnyToUnload$<CurStop>$ Bus;
    QUEUE ReadyToUnload$<CurStop>$ Bus;
    COMBI UnloadWorker$<CurStop>$;
    DISASSEMBLER DetachWorker$<CurStop>$ Bus;
    NORMAL WorkOrLeave$<CurStop>$;
    QUEUE BusWaits$<CurStop>$ Bus;
    QUEUE WorkersWait$<CurStop>$ Worker;
    COMBI LoadWorker$<CurStop>$;
    ASSEMBLER AttachWorker$<CurStop>$ Bus;
    COMBI BusFrom$<CurStop>$To$<NextStop>$;

    COLLECTOR Couldn'tBoardAt$<CurStop>$;

ASSIGN CurStop CurStop+1;
WEND;

The WHILE loop executes the statements in its body once for each bus stop. (Each statement is executed 5 times.) The preprocessor replacements are in bold face to make them easy to visualize. Note that the \textit{BusFrom$<CurStop>$To$<NextStop>$} Combis are defined after the corresponding \textit{LoadWorker$<CurStop>$} Combis in order to give higher priority to \textit{LoadWorker$<CurStop>$}. The Collectors named \textit{Couldn'tBoardAt$<CurStop>$} will keep statistics on the number of times that workers cannot board the bus because it is full. The WHILE loop defines a total of 55 network nodes and 5 Collectors.
/**Define Links common to commuter lots and work areas******************

ASSIGN CurStop 1;
WHILE CurStop<=NumStops;

/BUS Cycle
LINK B$<CurStop>$ _1
   BusFrom$<PrevStop>$To$<CurStop>$ LookInsideBus$<CurStop>$;
DISASMBASELINK B$<CurStop>$ _2
   LookInsideBus$<CurStop>$ CheckIfAnyToUnload$<CurStop>$;
LINK B$<CurStop>$ _3
   CheckIfAnyToUnload$<CurStop>$ ReadyToUnload$<CurStop>$;
LINK B$<CurStop>$ _4
   ReadyToUnload$<CurStop>$ UnloadWorker$<CurStop>$;
LINK B$<CurStop>$ _5
   UnloadWorker$<CurStop>$ DetachWorker$<CurStop>$;
DISASMBASELINK B$<CurStop>$ _6
   DetachWorker$<CurStop>$ CheckIfAnyToUnload$<CurStop>$;
LINK B$<CurStop>$ _7
   CheckIfAnyToUnload$<CurStop>$ BusWaits$<CurStop>$;
LINK B$<CurStop>$ _8
   BusWaits$<CurStop>$ LoadWorker$<CurStop>$;
ASMBASELINK B$<CurStop>$ _9
   LoadWorker$<CurStop>$ AttachWorker$<CurStop>$;
LINK B$<CurStop>$ _10
   AttachWorker$<CurStop>$ BusWaits$<CurStop>$;
LINK B$<CurStop>$ _11
   BusWaits$<CurStop>$ BusFrom$<CurStop>$To$<NextStop>$;

/WORKER Cycle
LINK W$<CurStop>$ _1
   DetachWorker$<CurStop>$ WorkOrLeave$<CurStop>$ Worker;
LINK W$<CurStop>$ _2
   WorkersWait$<CurStop>$ LoadWorker$<CurStop>$;
LINK W$<CurStop>$ _3
   LoadWorker$<CurStop>$ AttachWorker$<CurStop>$ Worker;

ASSIGN CurStop CurStop+1;
WEND;

Stroboscope executes the statements in the WHILE loop once for each bus stop.

Fourteen links are defined in every iteration (for a total of 70 links).

The model proceeds with the definition of the network element attributes:

ASSIGN CurStop 1;
WHILE CurStop<=NumStops;

/Update the CurStop SaveValue and the Bus SaveProp NumToUnloadNow
/when the bus goes through LookInsideBusX. LookInsidedBusX does
/not really disassemble, it just allows us to look inside the
/bus.
ONDISASSEMBLY LookInsideBus$<CurStop>$ ASSIGN CurStop $<CurStop>$;
ONDISASSEMBLY LookInsideBus$<CurStop>$ ASSIGN NumToUnloadNow
   LookInsideBus$<CurStop>$ .WorkersToUnloadNow.Count;

/Go to BusWaitsX only if there are no Workers to unload in the
/current stop.
STRENGTH B$<CurStop>$ _3 NumToUnloadNow;

Copyright © 1996 Julio C. Martínez
STRENGTH B$<CurStop>$._7 !NumToUnloadNow;

DURATION UnloadWorker$<CurStop>$ WorkerUnloadTime;

/Make sure that only one worker (of those that need to get off at
/this station) is detached in DetachWorkerX
RELEASEWHERE W$<CurStop>$._1 Destination==<CurStop>$;
RELEASEUNTIL W$<CurStop>$._1 WorkOrLeave<CurStop>$._Worker.Count;

/Update SaveProps of Bus after worker is detached
ONFLOW B$<CurStop>$._6 ASSIGN Content Content-1;
ONFLOW B$<CurStop>$._6 ASSIGN NumToUnloadNow NumToUnloadNow-1;

/Allow LoadWorkerX to start only if a bus with free space is
/available in BusWaitsX. If so, make sure you get a bus that is
/not full.
ENOUGH B$<CurStop>$._8 BusWaits$<CurStop>$._BusesWithSpace.Count;
DRAWWHERE B$<CurStop>$._8 !IsFull;
DURATION LoadWorker$<CurStop>$ WorkerLoadTime;

/Update SaveProps of Bus after worker is attached
ONASSEMBLY AttachWorker$<CurStop>$ ASSIGN Content AttachWorker$.<CurStop>$.Worker.Count;

/If some workers could not board the bus, collect statistics on the
/number of workers that could not board.
BEFOREDRAWS BusFrom$<CurStop>$To$<NextStop>$
  COLLECT Couldn'tBoardAt$<CurStop>$
  PRECOND WorkersWait$<CurStop>$._CurCount
  WorkersWait$<CurStop>$._CurCount;

/Determine the time it takes for the bus to get to the next station
DURATION BusFrom$<CurStop>$To$<NextStop>$
  'DistToNextStat[$<CurStop>-1]$'/
  BusFrom$<CurStop>$To$<NextStop>$._Bus.Speed';

  ASSIGN CurStop CurStop+1;
WEND;

Note that the first OnDisassembly action for LookInsideBus$<CurStop>$ uses
CurStop as the action target and $<CurStop>$ as the target argument. This is because
CurStop controls the WHILE loops during statement preprocessing, and also keeps track
of the bus’ location during simulation (i.e., during the execution of a SIMULATE or
SIMULATEUNTIL). Stroboscope actually executes the post-processed OnDisassembly
statement five times as follows:

ONDISASSEMBLY LookInsideBus1 ASSIGN CurStop 1;
ONDISASSEMBLY LookInsideBus2 ASSIGN CurStop 2;
ONDISASSEMBLY LookInsideBus3 ASSIGN CurStop 3;
ONDISASSEMBLY LookInsideBus4 ASSIGN CurStop 4;
ONDISASSEMBLY LookInsideBus5 ASSIGN CurStop 5;
Note also that although the duration expressions for the $BusFromXToY$ Combis are constant, the entire expressions cannot be placed inside the preprocessor replacements. This is because preprocessor replacements truncate to integers and in this case the resulting travel time is fractional.

The model proceeds with the definition of the nodes, links, and element attributes that exist in commuter lots but not in work areas:

```plaintext
/**Define Nodes, Links, and attributes exclusive to Commuter Lots**********
ASSIGN CurCommLot 1;
WHILE CurCommLot<=NumCommLots;
    ASSIGN CurStop MapCommLotsToStop[CurCommLot-1];
    COMBI InterArrive$<CurStop>$_;
    LINK W$<CurStop>$_0 InterArrive$<CurStop>$ WorkersWait$<CurStop>$;
    /Allow only once instance of InterArriveX at a time
    SEMAPHORE InterArrive$<CurStop>$_ !InterArrive$<CurStop>$_.CurInst;
    /The arrivals follow a Poisson process
    DURATION InterArrive$<CurStop>$
        Exponential[1/ArrivalRateAtLot$<CurCommLot>$];
    /Create a Worker just before each instance of InterArriveX ends
    BEFOREEND InterArrive$<CurStop>$ $ GENERATE 1 Worker;
    /Update the properties of workers as soon as they arrive
    ONRELEASE W$<CurStop>$_0 ASSIGN Lot$<CurStop>$;
    ONRELEASE W$<CurStop>$_0 ASSIGN WorkArea
        DestIfArvAtLot$<CurCommLot>$;
    ONRELEASE W$<CurStop>$_0 ASSIGN Destination WorkArea;
    ASSIGN CurCommLot CurCommLot+1;
WEND;
```

The model now defines the nodes, links, and element attributes that exist in work areas but not in commuter lots:

```plaintext
/**Define Nodes, Links, and attributes exclusive to Work Areas**********
ASSIGN CurWorkArea 1;
WHILE CurWorkArea<=NumWorkAreas;
    ASSIGN CurStop MapWorkAreasToStop[CurWorkArea-1];
    LINK W$<CurStop>$_0 WorkOrLeave$<CurStop>$ WorkersWait$<CurStop>$;
    DURATION WorkOrLeave$<CurStop>$ TimeAtWorkArea$<CurWorkArea>$;
    /Update the Worker’s Destination SaveProp, she’s going back to the
    /lot now.
    ONRELEASE W$<CurStop>$_0 ASSIGN Destination Lot;
```
ASSIGN CurWorkArea CurWorkArea+1;
WEND;

At this point, the entire model is defined and the bus stops are connected. The model could simply initialize the Queue that will initially hold the bus, simulate, and then print the standard report. Before doing that, the model contains code that prints a “bus log” and/or a population graph.

The first option is really a debugging aid. It fills up a table with one row each time the bus gets to a stop. The information in each row indicate the stop number, time of arrival to the stop, number of workers inside the bus, number of workers waiting to get in the bus, and number of workers that will get off the bus. Additionally, the table shows the number of workers that arrived at the bus stop while other workers were getting off or getting in, the time of departure, the number of workers in the bus at departure, and the number of workers that could not get in the bus because they did not fit:

/**Code to print a Bus Log if desired****************************************************************************
IF PrintBusLog;
OUTFILE BusLog "BusLog.sto";
PRINT BusLog "STOP -------At Arrival------|---While In Stop---|----At Departure----
Time InBus WtBoard |GotOff Arvd GotOn |    Time InBus NoFit
================================================================================|
ASSIGN CurStop 1;
WHILE CurStop< =NumStops;

/Define SaveValues to keep track of Workers that arrive and
(that get on the bus.
SAVEVALUE nArrived$<CurStop>$ 0;
SAVEVALUE nGotOn$<CurStop>$ 0;

/Print information that is available when the bus arrives
ONFLOW B$<CurStop>$ _2 PRINT BusLog
"%4.0f %8.3f %5.0f %7.0f |%6.0f 
$<CurStop>$
SimTime
Content
WorkersWait$<CurStop>$.$CurCount
NumToUnloadNow;

/When the bus arrives set the number of arrivals and count of
(workers that got on the bus to zero
ONFLOW B$<CurStop>$ _2 ASSIGN nArrived$<CurStop>$ 0;
ONFLOW B$<CurStop>$ _2 ASSIGN nGotOn$<CurStop>$ 0;
INCREMENT number of arrivals every time a worker arrives
ONENTRY WorkersWait<CurStop>
  ASSIGN nArrived<CurStop>
  nArrived<CurStop>+1;

INCREMENT the number of workers that boarded every time a worker enters the bus
ONDRAW W<CurStop>_2 ASSIGN nGotOn<CurStop>
  nGotOn<CurStop>+1;

Print the remaining information when the bus leaves,
except at the very beginning of the simulation so that the first row of the table does not start in the middle
ONDRAW B<CurStop>_11 PRINT BusLog PRECOND SimTime
  "%4.0f %5.0f |%8.3f %5.0f %5.0f\n"
  nArrived<CurStop>
  nGotOn<CurStop>
  SimTime
  Content
  WorkersWait<CurStop>.CurCount;

  ASSIGN CurStop CurStop+1;
WEND;
ENDIF;

The second option presents a plot of the number of workers in the system as a function of time. This graph can be used to visually determine when the system has reached steady state (i.e., the population is more or less constant). It can also be used to see if the arrival rate is too large for the system (i.e., the population increases constantly):

/** Code to graph the population of the system if desired ****************
IF GraphPopulation;

OUTFILE PopulationGraph "PopGraph.sto";
PRINT PopulationGraph "Hour\tCurPp\tAvePp\tSDPp\n";

/Create a Combi that sequentially starts and ends every hour and
/update the graph every time it ends
COMBI PopGrapher;
SEMAPHORE PopGrapher !PopGrapher.CurInst;
DURATION PopGrapher 1;
ONEND PopGrapher PRINT PopulationGraph
  "%4f\t%.4f\t%.4f\t%.4f\n"
  SimTime
  Worker.CurPp
  Worker.AvePp
  Worker.SDPp;

ENDIF;
The graph is really a tab delimited table. Any modern spreadsheet program can open the “PopGraph.sto” file and create a graph from the data with minimal effort. An example of such a graph appears later.

At this point, the model initializes the Queue that will initially contain the bus. It then transfers the Variables defined in the input section at the beginning of the model to the bus properties:

```plaintext
/**Initialize Queues and transfer bus properties

INIT BusWaits1 1;
ASSIGN BusWaits1.Bus.Speed BusSpeed;

Notice that values can be directly assigned to the properties of the bus in
BusWaits1 because there is only one bus. If there is more than one bus in the same
Queue, another method must be used to assign initial values to the properties of the bus
(e.g., Filters or OnEntry actions).

The model concludes with the statements that run the simulation and print a
custom report:

/**Simulate and print report

SIMULATEUNTIL SimTime>=SimulationLength*24;
PRINT StdOutput "Number of days simulated : %.2f\n" SimTime/24;
PRINT StdOutput "Stop-|Waiting Times-(min)|--Departures-|No Fit In Bus-
Av Avg StdDv Max Total LftWr Tot Ave Max
=====|====================|=============|===============
"
ASSIGN CurStop 1;
WHILE CurStop<=NumStops;
PRINT StdOutput "%4.0f | %5.2f %5.2f %6.2f | %5.0f  %4.0f | %4.0f %5.2f %3.0f\n"
CurStop
WorkersWait$<CurStop>$$.Worker.AvVstTm*60
WorkersWait$<CurStop>$$.Worker.SDAvVst*60
WorkersWait$<CurStop>$$.Worker.MxVstTm*60
BusFrom$<CurStop>$To$<NextStop>$$.TotInst
Couldn'tBoardAt$<CurStop>$$.nSamples
Couldn'tBoardAt$<CurStop>$$.SumVal
Couldn'tBoardAt$<CurStop>$$.AveVal
Couldn'tBoardAt$<CurStop>$$.MaxVal;
ASSIGN CurStop CurStop+1;
WEND;

Copyright © 1996 Julio C. Martínez 294
Running the model of the commuter transportation system produces as main output a table that looks as follows:

<table>
<thead>
<tr>
<th>Stop</th>
<th>Avg</th>
<th>StdDev</th>
<th>Max</th>
<th>Total LftWr</th>
<th>Tot Ave Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14.34</td>
<td>9.16</td>
<td>49.25</td>
<td>493</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>15.41</td>
<td>9.80</td>
<td>50.12</td>
<td>493</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>14.71</td>
<td>9.44</td>
<td>44.75</td>
<td>492</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>15.16</td>
<td>9.57</td>
<td>56.94</td>
<td>492</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>15.52</td>
<td>10.18</td>
<td>72.81</td>
<td>492</td>
<td>10</td>
</tr>
</tbody>
</table>

Execution Time = 23.125 seconds
Processor Time = 22.9531 seconds

The results show that the average worker waiting time at stop 1 was 14.34 minutes with a standard deviation of 9.16 minutes. The longest wait was 49.25 minutes. The bus stopped at the first bus stop 493 times. On 9 of the 493 times there were workers that could not board the bus because it was full. For these 9 occasions, the maximum number of workers that could not board was 5. On average 2.22 workers could not board on each of these 9 occasions. A total of 20 workers could not board the bus at stop 1 the first time they saw the bus.

The “bus log” is very useful for debugging purposes. It can be examined to make sure that the system is operating as intended. Although this information can be obtained from the standard trace provided automatically by Stroboscope, the “bus log” presents only the information that matters. The following are portions of the “bus log” created by a run of this model:
The graph in Figure 37 was produced almost automatically by a spreadsheet program from the data in file “PopGraph.sto”. Notice from the current population curve

![Figure 37 - Population Graph](image-url)
that the system is fairly stable after about 10 hours of operation. The curves for average population and standard deviation of the population do not stabilize until much later because they incorporate data from the first 10 hours (when the population was building up).

In continuous systems that eventually reach a steady state, it is convenient to “discard” the data collected while the system is “warming up”. This is discussed in the next chapter.

### 15.4 Extended Solution

The model for the commuter transportation system in the previous section is limited to one bus. Furthermore, it is not defined in a completely parametric fashion. It requires the definition of special variables such as $\text{ArrivalRateAtLotX}$, $\text{DestIfArvAtLotX}$, and $\text{TimeAtWorkAreaX}$. The model was presented first because it is easier to understand and serves as an introduction to the model below.

This section shows a completely parametric model that can handle any number of buses and that optionally requires that buses spend at least a specified minimum amount of time at each bus stop. (For example, buses may need to spend at least 3 minutes at bus stop 4.) The solution also includes an option that does not allow buses to unload/load in parallel at the same bus stop (i.e., one bus must unload and load before the next bus unloads and loads).

The model presented here is an advanced example of a code generator. It assumes an in-depth understanding of the material presented in the earlier chapters of this manuscript.

Highlights of this model as compared to the model of the previous section are:

- The new model uses an Array to define the relative probability that a worker arriving at commuter lot $X$ will have bus stop $Y$ as a destination.

- These relative probabilities are normalized and changed into a cumulative distribution by new code in the “do not touch below this line” section of the code.
• Another subsequent section of code uses the cumulative, normalized probabilities, to build variables of the form $DestIfArvAtLotX$.

• Several parts of the code that were implemented with Filters are now implemented with VarProps. This is simply to illustrate different ways of modeling similar situations.

• While the simulation is running, the bus stop at which a bus is stopped is stored in a SaveProp of the bus. The $CurStop$ variable used as a preprocessing aid in the previous model cannot be re-used because there is more than one bus.

• A Semaphore prevents the $BusFromXToY$ Combis from starting before the minimum amount of waiting time has elapsed. The new model includes Normals with names of the form $MinWaiterX$ (see Figure 38 and Figure 39). These Normals start when a bus arrives at the corresponding bus stop. They have a duration equal to the minimum waiting time for that bus stop. Their purpose is to trigger a Combi Instantiation Phase at their termination. This ensures that Stroboscope will examine whether a bus can depart from a bus stop exactly when the minimum amount of waiting time has elapsed.

![Diagram](image)

*Figure 38 - Commuter Lot 2 (Bus Stop 4) With Minimum Waiting Time*
Figure 39 - Work Area 3 (BusStop 5) With Minimum Waiting Time

The specific data shown in the solution for this section appears in Figure 40. The data differs from the data in the previous section as follows:

- This version uses two buses, each with half the capacity of the bus in the solution of the previous section. One bus is originally in bus stop 1 and the other in bus stop 4. In addition, the buses are slower. One of them travels at 40 km/hr and the other at 35 km/hr.

- Buses must spend at least 3 minutes in commuter lots and 1 minute in work areas, even if there are no workers to unload or load.

- In addition to the lower bus speeds and minimum stopping time requirement, the new model also has arrival rates at the commuter lots that are higher than before.
15.4.1 Extended Solution Code

/**Statement Execution Switches***************************************************************************/

VARIABLE GraphPopulation 0; /*use 1 to graph population, 0 otherwise*/
VARIABLE AllowParallelLds 1; /*use 1 to allow parallel loading/unloading*/
VARIABLE UseMinWaits 1; /*use 1 to enforce minimum wait times at bus stops, 0 otherwise*/
/**Simulation Run Parameters***************************************************************************/
SEED 9111964; /*comment or replace as necessary*/
VARIABLE SimulationLength 10; /* (days)*/
/**Number of Buses, Work Areas, and Commuter Lots***************************************************************************/
VARIABLE NumBuses 2; /*number of buses*/
VARIABLE NumCommLots 2; /*number of commuter lots*/
VARIABLE NumWorkAreas 3; /*number of work areas*/
VARIABLE NumStops NumCommLots+NumWorkAreas;
/**Bus characteristics***************************************************************************/
/* Use one row for each bus. The columns should contain the following data in the order shown:*/
/* Load Time low (sec), Load Time high (sec), Unload Time low (sec),*/
/* Unload Time high (sec), Bus Capacity, Bus Speed (km/hr),*/
/* Initial Bus Stop*/
ARRAY BusData NumBuses 7
{ 15 25 16 24 15 40 1
  15 25 16 24 15 35 4};
/* LdTmLow LdTmHigh UnLdTmLow UnLdTmHigh Cap Speed InitStop*/
/**Commuter Lot Data***************************************************************************/
/* The values in index X indicate the bus stop number that corresponds to commuter lot X+1*/
ARRAY MapCommLotsToStop NumCommLots {1 4}; /*Lots = stops 1 & 4*/
/* The values in index X indicate the arrival rate in workers per hour at commuter lot X+1*/
ARRAY ArrivalRates NumCommLots { 26 19};
/* Use one row for each commuter lot. Column X should contain the relative probability that the destination is bus stop X+1.*/
/* The implementation will normalize and calculate cumulatives from these numbers.*/
ARRAY Destinations NumCommLots NumStops
{ 0 28 48 0 24
  0 35 21 0 44};
/**Work Area Data***************************************************************************/
/* The values in index X indicate the bus stop number that corresponds to work area X+1*/
ARRAY MapWorkAreasToStop NumWorkAreas (2 3 5); /*Work areas = stops 2,3,5*/
/ Use one row for each work area. The columns should contain the
/ following data in the order shown:
/ Mean Work Time (hr), Standard Deviation of Work time (hr)
ARRAY WorkTimes NumWorkAreas 2
{ 8 0.50
  7 0.75
  9 0.35};
/ MeanWrkTm StdDevWrkTm

/**Distance Between Bus Stops*************************************************************************
/ The values in index X indicate the distance between stop X+1 and X+2
ARRAY DistToNextStat NumStops {1.84 1.37 2.01 1.36 1.33}; / (km)

IF UseMinWaits;
/ The values in index X indicate the minimum time that a bus must
/ spend at bus stop X+1 before it may depart
ARRAY MinWaitingTimes NumStops { 3 1 1 3 1}; / (min)
ENDIF;

=======================================================================
/**Implementation, Do not modify statements below this line*************
=======================================================================

/**Helper SaveValues and Variables**************************************************************************
SAVEVALUE CurStop 1;
VARIABLE NextStop 'CurStop < NumStops ? CurStop+1 : 1';
VARIABLE PrevStop 'CurStop == 1 ? NumStops : CurStop-1';

SAVEVALUE CurBus 1;
SAVEVALUE CurCommLot 1;
SAVEVALUE CurWorkArea 1;

/**Create Variables for determination of destinations**************************************************************************
/Normalize and make Cumulative the Destinations Array
SAVEVALUE SumOfProbs 0;

/In the following WHILE loop, CurCommLot is zero-based to match the
/Destinations Array, so when CurCommLot is 3, for example, it refers to
/Commuter lot number 4
ASSIGN CurCommLot 0;
WHILE CurCommLot<NumCommLots;
    /Calculate in SumOfProbs the sum of the relative probabilites in the
    /row that corresponds to the current commuter lot
    ASSIGN SumOfProbs Destinations[CurCommLot,0];
    ASSIGN CurStop 1;
    WHILE CurStop<NumStops;
        ASSIGN SumOfProbs SumOfProbs+Destinations[CurCommLot,CurStop];
        ASSIGN CurStop CurStop+1;
    WEND;
    /Normalize the relative probabilities and change them to cumulative

/form
ASSIGN Destinations CurCommLot 0
       Destinations[CurCommLot,0]/SumOfProbs;
ASSIGN CurStop 1;
WHILE CurStop<NumStops;
   ASSIGN Destinations CurCommLot CurStop
        'Destinations[CurCommLot,CurStop-1]+
        Destinations[CurCommLot,CurStop]/SumOfProbs';
   ASSIGN CurStop CurStop+1;
WEND;
ASSIGN CurCommLot CurCommLot+1;
WEND;

/Build the destination variables of the form DestIfArvAtLotX.
/
/In the previous model we defined them explicitly in the user input
/section. For example:
/   DestIfArvAtLot2 = 'Rnd[1]<=0.28 ? 2 : LastRnd[1]<=0.76 ? 3 : 5'.
/
/Observe that DestIFArvAtLot2 is defined to return 2,3, or 5. Bus stops
/1 and 4 are not included because we know that workers do not want to go
/there. If we were to include the chained conditionals for bus stops 1
/and 4 also (since we do not know beforehand that workers do not want to
/go there), the conditional expression would look as follows:
/   DestIFArvAtLot2 = 'Rnd[1]<=0 ? 1 : LastRnd[1]<=0.28 ? 2 :
       LastRnd[1]<=0.76 ? 3 LastRnd[1]<=0.76 ? 4 : 5'
/
/Since the number of chained conditionals depends on the number of bus
/stops and may be varied in the input section, we need to mimic the
/conditional with a chain of variables. The code below will create a
/series of variables that look as follows:
/   DestIFArvAtLot2 = Rnd[1]<=0 ? 1 : Dest2_2;
/   Dest2_2 = LastRnd[1]<=0.28 ? 2 : Dest2_3;
/   Dest2_3 = LastRnd[1]<=0.76 ? 3 : Dest2_4;
/   Dest2_4 = LastRnd[1]<=0.76 ? 4 : Dest2_5;
/   Dest2_5 = 5;
/Obviously these variables need to be defined in reverse order because
/one must be defined before it is used in the definition of another.
/Also note that the variables that the code below builds do not have
/names as above, but instead obtain them from the Destinations Array.
/(Which was converted from relative probabilities to normalized
/cumulatives in the previous code)

ASSIGN CurCommLot 1;
WHILE CurCommLot<NumCommLots;

   /If, for example, NumStops is 5 and CurCommLot is 2, we first
   /define the variable Dest2_5 = 5
   VARIABLE Dest$<CurCommLot>$$_<NumStops>$_$ NumStops;

   ASSIGN CurStop NumStops-1;
   WHILE CurStop>1;
      /If, for example, CurCommLot is 2 and CurStop is 4, we define
      /Dest2_5 = LastRnd[1]<=Destinations[1,3] ? 4 : Dest2_5

Copyright © 1996 Julio C. Martínez
VARIABLE Dest$<CurCommLot>$<CurStop>$
'LastRnd[]<=Destinations[$<CurCommLot-1>$,$<CurStop-1>$] ?
$<CurStop>$ : 
Dest$<CurCommLot>$<CurStop+1>$';

ASSIGN CurStop CurStop-1;
WEND;

/If, for example, CurCommLot is 2, we define the variable
/DestIfArvAtLot2 = Rnd[]<=Destinations[1,0] ? 1 : Dest2_2
VARIABLE DestIfArvAtLot$<CurCommLot>$
'Rnd[]<=Destinations[$<CurCommLot-1>$,0] ?
1 : 
Dest$<CurCommLot>$';

ASSIGN CurCommLot CurCommLot+1;
WEND;

/**Resource Types******************************************************************************
COMPTYPE Bus;
SAVEPROPS Bus Content CurStop ArrivalTm NumToUnloadNow;
VARPROP Bus LoadTime Uniform[BusData[ResNum-1,0],BusData[ResNum-1,1]]/3600;
VARPROP Bus UnloadTime Uniform[BusData[ResNum-1,2],BusData[ResNum-1,3]]/3600;
VARPROP Bus Capacity BusData[ResNum-1,4];
VARPROP Bus Speed BusData[ResNum-1,5];
VARPROP Bus NotFull Content<Capacity;

IF UseMinWaits;
VARPROP Bus WaitTime SimTime-ArrivalTm;
VARPROP Bus WaitedLongEnough WaitTime>=MinWaitingTimes[CurStop-1]/60;
ENDIF;

COMPTYPE Worker;
SAVEPROPS Worker Lot WorkArea Destination;

IF UseMinWaits;
GENTYPE Signal;
ENDIF;

/**Define Nodes common to Commuter Lots and Work Areas**************************************
ASSIGN CurStop 1;
WHILE CurStop<=NumStops;

IF UseMinWaits;
NORMAL MinWaiter$<CurStop>$;
ENDIF;

DISASSEMBLER LookInsideBus$<CurStop>$ Bus;
DYNAFORK CheckIfAnyToUnload$<CurStop>$ Bus;
QUEUE ReadyToUnload$<CurStop>$ Bus;
COMBI UnloadWorker$<CurStop>$;
DISASSEMBLER DetachWorker$<CurStop>$ Bus;
NORMAL WorkOrLeave$<CurStop>$;
QUEUE BusWaits$<CurStop>$ Bus;
QUEUE WorkersWait$<CurStop>$ Worker;
COMBI LoadWorker$<CurStop>$;
ASSEMBLER AttachWorker$<CurStop>$ Bus;
COMBI BusFrom$<CurStop>$To$<NextStop>$;

COLLECTOR Couldn'tBoardAt$<CurStop>$;

FILTER WorkersToUnloadNow$<CurStop>$ Worker
   Destination==LookInsideBus$<CurStop>$ .CurStop;

ASSIGN CurStop CurStop+1;
WEND;

/**Define Links common to commuter lots and work areas***************

ASSIGN CurStop 1;
WHILE CurStop<=NumStops;

/Bus Cycle
LINK B$<CurStop>$ _1
   BusFrom$<PrevStop>$To$<CurStop>$ LookInsideBus$<CurStop>$;
DISASSEMBLELINK B$<CurStop>$ _2
   LookInsideBus$<CurStop>$ CheckIfAnyToUnload$<CurStop>$;
LINK B$<CurStop>$ _3
   CheckIfAnyToUnload$<CurStop>$ ReadyToUnload$<CurStop>$;
LINK B$<CurStop>$ _4
   ReadyToUnload$<CurStop>$ UnloadWorker$<CurStop>$;
LINK B$<CurStop>$ _5
   UnloadWorker$<CurStop>$ DetachWorker$<CurStop>$;
DISASSEMBLELINK B$<CurStop>$ _6
   DetachWorker$<CurStop>$ CheckIfAnyToUnload$<CurStop>$;
LINK B$<CurStop>$ _7
   CheckIfAnyToUnload$<CurStop>$ BusWaits$<CurStop>$;
LINK B$<CurStop>$ _8
   BusWaits$<CurStop>$ LoadWorker$<CurStop>$;
ASSEMBLELINK B$<CurStop>$ _9
   LoadWorker$<CurStop>$ AttachWorker$<CurStop>$;
LINK B$<CurStop>$ _10
   AttachWorker$<CurStop>$ BusWaits$<CurStop>$;
LINK B$<CurStop>$ _11
   BusWaits$<CurStop>$ BusFrom$<CurStop>$To$<NextStop>$;

/Worker Cycle
LINK W$<CurStop>$ _1
   DetachWorker$<CurStop>$ WorkOrLeave$<CurStop>$ Worker;
LINK W$<CurStop>$ _2
   WorkersWait$<CurStop>$ LoadWorker$<CurStop>$;
LINK W$<CurStop>$ _3
   LoadWorker$<CurStop>$ AttachWorker$<CurStop>$ Worker;

IF UseMinWaits;
   /Signal Cycle
LINK S$<CurStop>$ _1
   BusFrom$<PrevStop>$To$<CurStop>$
      MinWaiter$<CurStop>$ Signal;
ENDIF;
ASSIGN CurStop CurStop+1;
WEND;

/** Define attributes for elements in commuter lots and work areas******

ASSIGN CurStop 1;
WHILE CurStop<=NumStops;
    IF UseMinWaits;
        DURATION MinWaiter<$CurStop$> MinWaitingTimes[$<CurStop-1>$]/60;
        ENDIF;
    ONDISASSEMBLY LookInsideBus<$CurStop$> ASSIGN CurStop $<CurStop>$;
    ONDISASSEMBLY LookInsideBus<$CurStop$> ASSIGN NumToUnloadNow
        LookInsideBus<$CurStop$>.WorkersToUnloadNow$<CurStop>$ .Count;
    IF UseMinWaits;
        ONDISASSEMBLY LookInsideBus<$CurStop$> ASSIGN ArrivalTm SimTime;
        ENDIF;
    STRENGTH B$<CurStop>$ _3 NumToUnloadNow;
    STRENGTH B$<CurStop>$ _7 !NumToUnloadNow;
    DURATION UnloadWorker$<CurStop>$
        UnloadWorker$<CurStop$>.Bus.UnloadTime;
    RELEASEWHERE W$<CurStop>$ _1 Destination===$<CurStop>$ ;
    RELEASEUNTIL W$<CurStop>$ _1 WorkOrLeave$<CurStop$>.Worker.Count;
    ONFLOW B$<CurStop>$ _6 ASSIGN Content Content-1;
    ONFLOW B$<CurStop>$ _6 ASSIGN NumToUnloadNow NumToUnloadNow-1;
    ENOUGH B$<CurStop>$ _8 BusWaits$<CurStop$>.NotFull.SumVal;
    DRAWWHERE B$<CurStop>$ _8 NotFull;
    DURATION LoadWorker$<CurStop>$
        LoadWorker$<CurStop$>.Bus.LoadTime;
    ONASSEMBLY AttachWorker$<CurStop>$ ASSIGN Content
        AttachWorker$<CurStop$>.Worker.Count;
    IF UseMinWaits;
        SEMAPHORE BusFrom$<CurStop$>$To$<NextStop>$
            BusWaits$<CurStop$>.WaitedLongEnough.SumVal;
        ENDIF;
    BEFOREDRAWS BusFrom$<CurStop$>$To$<NextStop>$
        COLLECT CouldnntBoardAt$<CurStop>$
            WorkersWait$<CurStop$>.CurCount
            WorkersWait$<CurStop$>.CurCount;
    IF UseMinWaits;
        DRAWWHERE B$<CurStop>$ _11 WaitedLongEnough;
        ENDIF;
DURATION BusFrom$<CurStop>$To$<NextStop>$
  'DistToNextStat[$<CurStop-1>$]/
    BusFrom$<CurStop>$To$<NextStop>$$.Bus.Speed';

IF !AllowParallelLds;
  SEMAPHORE UnloadWorker$<CurStop>$
    '!UnloadWorker$<CurStop>$$.CurInst &
      !LoadWorker$<CurStop>$$.CurInst';
  SEMAPHORE LoadWorker$<CurStop>$
    '!UnloadWorker$<CurStop>$$.CurInst &
      !LoadWorker$<CurStop>$$.CurInst';
ENDIF;

ASSIGN CurStop CurStop+1;
WEND;

/**Define Nodes, Links, and attributes exclusive to Commuter Lots******
 ASSIGN CurCommLot 1;
 WHILE CurCommLot<=NumCommLots;
   ASSIGN CurStop MapCommLotsToStop[CurCommLot-1];
   COMBI InterArrive$<CurStop>$;
   LINK W$<CurStop>$_0 InterArrive$<CurStop>$ WorkersWait$<CurStop>$;
   SEMAPHORE InterArrive$<CurStop>$ !InterArrive$<CurStop>$$.CurInst;
   DURATION InterArrive$<CurStop>$
     Exponential[1/ArrivalRates[$<CurCommLot-1>$]];
   BEFOREEND InterArrive$<CurStop>$ GENERATE 1 Worker;
   ONRELEASE W$<CurStop>$_0 ASSIGN Lot $<CurStop>$;
   ONRELEASE W$<CurStop>$_0 ASSIGN WorkArea
     DestIfArvAtLot$<CurCommLot>$;
   ONRELEASE W$<CurStop>$_0 ASSIGN Destination WorkArea;
   ASSIGN CurCommLot CurCommLot+1;
 WEND;

/**Define Nodes, Links, and attributes exclusive to Work Areas**********
 ASSIGN CurWorkArea 1;
 WHILE CurWorkArea<=NumWorkAreas;
   ASSIGN CurStop MapWorkAreasToStop[CurWorkArea-1];
   LINK W$<CurStop>$_0 WorkOrLeave$<CurStop>$ WorkersWait$<CurStop>$;
   DURATION WorkOrLeave$<CurStop>$
     'Normal[WorkTimes[$<CurWorkArea-1>$,0],
      WorkTimes[$<CurWorkArea-1>$,1]]';
   ONRELEASE W$<CurStop>$_0 ASSIGN Destination Lot;
ASSIGN CurWorkArea CurWorkArea+1;
WEND;

/**Code to graph the population of the system if desired******************
IF GraphPopulation;

OUTFILE PopulationGraph "PopGraph.sto";
PRINT PopulationGraph "Hour\tCurPp\tAvePp\tSDPp\n";

/>Create a Combi that sequentially starts and ends every hour and
/update the graph every time it ends
COMBI PopGrapher;
SEMAPHORE PopGrapher !PopGrapher.CurInst;
DURATION PopGrapher 1;
ONEND PopGrapher PRINT PopulationGraph
"%.0f\t%.4f\t%.4f\t%.4f\n"
SimTime
Worker.CurPp
Worker.AvePp
Worker.SDPp;
ENDIF;

/**Initialize Queues and transfer bus properties**************************
FILTER LastBus Bus ResNum==Bus.TotPp;
ASSIGN CurBus 1;
WHILE CurBus<=NumBuses;
   INIT BusWaits$<BusData[CurBus-1,6]$ 1;
   ASSIGN BusWaits$<BusData[CurBus-1,6]$ .LastBus.CurStop
       BusData[CurBus-1,6];
   ASSIGN CurBus CurBus+1;
WEND;

/**Simulate and print report********************************************
SIMULATEUNTIL SimTime>=SimulationLength*24;
PRINT StdOutput "Number of days simulated : %.2f\n\n" SimTime/24;
PRINT StdOutput "Summary statistics about waiting for the bus\n\n";
PRINT StdOutput "
Stop-|Waiting Times-(min)|--Departures-|--No Fit In Bus-
     | Avg StdDv Max | Total LftWr | Tot Ave Max
====|====================|=============|===============
 ASSIGN CurStop 1;
WHILE CurStop<=NumStops;
PRINT StdOutput
"%4.0f | %5.2f %5.2f %6.2f | %5.0f %4.0f | %4.0f %5.2f %3.0f
"

CurStop
WorkersWait$<CurStop>$$.Worker.AvVstTm*60
WorkersWait$<CurStop>$$.Worker.SDAvVst*60
WorkersWait$<CurStop>$$.Worker.MxVstTm*60
BusFrom$<CurStop>$$.To$<NextStop>$$.TotInst
CouldntBoardAt$<CurStop>$$.nSamples
'CouldntBoardAt$<CurStop>$$.nSamples ?
    CouldntBoardAt$<CurStop>$$.SumVal : -1'
'CouldntBoardAt$<CurStop>$$.nSamples ?
    CouldntBoardAt$<CurStop>$$.AveVal : -1'
'CouldntBoardAt$<CurStop>$$.nSamples ?
    CouldntBoardAt$<CurStop>$$.MaxVal : -1';

ASSIGN CurStop CurStop+1;
WEND;

15.4.2 Extended Solution Output

A run of the extended model produces the following output:

Stroboscope Model MultiBus.str (203336448)

Number of days simulated : 10.00

Summary statistics about waiting for the bus

<table>
<thead>
<tr>
<th>Stop-</th>
<th>-Waiting Times- (min)</th>
<th>--Departures--</th>
<th>-No Fit In Bus-</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg StdDv Max</td>
<td>Total LftWr</td>
<td>Tot Ave Max</td>
</tr>
<tr>
<td>1</td>
<td>12.04 7.23 41.77</td>
<td>1083</td>
<td>80 282 3.52 9</td>
</tr>
<tr>
<td>2</td>
<td>12.67 7.59 39.96</td>
<td>1083</td>
<td>80 251 3.14 9</td>
</tr>
<tr>
<td>3</td>
<td>12.94 7.86 50.57</td>
<td>1083</td>
<td>109 372 3.41 10</td>
</tr>
<tr>
<td>4</td>
<td>11.90 7.51 36.11</td>
<td>1084</td>
<td>67 211 3.15 10</td>
</tr>
<tr>
<td>5</td>
<td>12.56 7.61 42.31</td>
<td>1082</td>
<td>98 316 3.22 11</td>
</tr>
</tbody>
</table>

---------------------------------
Execution Time = 44.656 seconds
Processor Time = 44.3906 seconds

Observe that with smaller buses (the total number of seats being the same), and despite slower bus speeds and higher traffic, the average and maximum waiting times improve. Figure 41 shows the population graph for the extended example. The graph shows that this system can handle a larger work force.
Figure 41 - Population Graph With Two Small Buses
Chapter 16
Performing Multiple Replications

This chapter introduces the concepts and statements necessary to automatically perform multiple replications of the same or similar simulation models. It includes examples that enhance those presented in earlier sections and illustrate several methods of performing multiple replications.

Detailed explanations of the concepts presented here and their application can be found in (Law and Kelton 1991) and (Schriber 1990). This chapter simply presents the Stroboscope tools that enable us to perform multiple replications.

Performing multiple replications in Stroboscope is very simple. It requires the use of two new statements and an extension to the statements that define SaveValues and Collectors. The first of the two new statements, RESETSTATS, clears the statistics kept by the system without changing the state of the simulation. The second of the new statements, CLEAR, clears the statistics kept by the system and additionally resets the state of the simulation (i.e., returns SimTime to zero, kills all Activity instances currently in the FEL without terminating them, and destroys all resources held in Queues). The statements that define SaveValues and Collectors have an enhanced syntax that allow us to make the objects defined with the statements persistent. Persistent SaveValues and Collectors are immune to the CLEAR and RESETSTATS statements.

The statements and concepts described in this chapter can be used to perform multiple replications in numerous ways. The specific forms of multiple replications shown here are the most common and are only examples.
16.1 Persistent SaveValues and Collectors

When Stroboscope executes a CLEAR or RESETSTATS statement, the Collectors in the model discard all the data they had collected. Additionally, in the case of the CLEAR statement only, Stroboscope reassigns original values to SaveValues (i.e., SaveValues attain the value with which they were defined).

It is usually necessary to collect statistics across replications or to use SaveValues to control While-loops that enclose a CLEAR statement. The Collectors and SaveValues used in these cases must not be affected by a statistical reset or model clear.

Persistent SaveValues and Collectors are immune to the CLEAR and RESETSTATS statements. SaveValues and Collectors are made persistent by appending an asterisk “*” to their names in the statements that define them. The asterisk is not part of the name, but rather an indicator that the SaveValue or Collector is persistent.

For example:

/Define a persistent weighted collector -- append an asterisk
WGTCOLLECTOR GroundRating*;

/Use the collector in statements normally (without the asterisk)
ONSTART ChangeGroundClass ASSIGN GroundRating
    PrevGroundClass PrevStepLength;

16.2 The Confidence Function

Simulations are typically run to measure the performance of a system. For example, an earth-moving operation may be modeled to estimate the cost per cubic meter of moved earth that corresponds to a specific equipment configuration. The answer obtained from a single run of a simulation model is often stochastic. Running the same model with a different set of random numbers generally produces different results. In general, the objective is to determine the expected value of the result and its variability.

Confidence intervals bracket the expected value of a series of point samples with a certain level of confidence. For example, if the 90% Confidence interval for the cost per cubic meter of moved earth spans from $2.90/m³ to $3.15/m³, there is a 90% chance
that the true mean is in that range. If 1000 different 90% confidence intervals are obtained, on average 100 of them will not contain the true expected value.

The desired width of a confidence interval usually dictates how many times a simulation model is run. For example, an earth-moving operation may be simulated until the width of the 90% confidence interval for the cost per cubic meter is at most $0.10.

The width of a confidence interval depends on the standard deviation, number of samples, and desired confidence level as shown by the following formula:

\[
\text{CIHalfWidth} = \frac{\text{StdDev}}{\sqrt{\text{nSamples}}} \times t_{\text{Inv}}\left(\frac{1-\text{ConfLevel}}{2}\right, \text{nSamples}-1
\]

Stroboscope offers the Confidence function, based on the formula above, to determine the half-width of a confidence interval:

\[
\text{Confidence}[\text{StandardDeviation, ConfidenceLevel, NumberOfSamples}];
\]

The following code collects 10 statistical samples and then prints the 90% confidence interval on the expected value of the samples:

```
COLLECTOR UnitCost;
COLLECT UnitCost 2.90;
COLLECT UnitCost 2.97;
COLLECT UnitCost 2.82;
COLLECT UnitCost 3.15;
COLLECT UnitCost 3.02;
COLLECT UnitCost 2.79;
COLLECT UnitCost 2.92;
COLLECT UnitCost 3.05;
COLLECT UnitCost 2.89;
COLLECT UnitCost 2.94;
PRINT StdOutput
"90% Confidence Interval on Unit Cost is : [%.2f, %.2f]"
'UnitCost.AveVal-
    Confidence[UnitCost.SDVal,0.90,UnitCost.nSamples]'n
'UnitCost.AveVal+
    Confidence[UnitCost.SDVal,0.90,UnitCost.nSamples]';
```

Although the above code does not run a simulation, it can be submitted to Stroboscope to produce the following output:

```
Stroboscope Model Strobo1 (1631584456)
90% Confidence Interval on Unit Cost is : [2.88, 3.01]
------------------------------------------
Execution Time = 0.094 seconds
Processor Time = 0.015625 seconds
```
16.3 The RESETSTATS statement

The RESETSTATS statement is used for continuously ongoing processes that never terminate. In such processes, operations are always taking place.

An example of a continuous process that never stops is the commuter transportation system described in section 15.2. Figure 41 shows the population of the system as a function of time in the extended solution that uses 2 buses. The population is zero only at the beginning of the very first day of operation. The population then rises steadily until after about 12 hours of operation. The population of the system then stabilizes and oscillates around the long term average, which is about 400. The statistics collected while the system is “warming up” distort those statistics collected after the system is in steady state. Thus, even though the population of the system is fairly stable after 12 hours of operation, the average population continues to increase slightly and the standard deviation of the population continues to decrease. This is because the “noise” collected during the first 12 hours affects the steady data collected thereafter.

In such situations, it is convenient to discard the data collected while the system is warming up. The RESETSTATS control statement does precisely this.

The RESETSTATS control statement clears the statistics kept by the system but does not otherwise alter the state of the simulation. The resources currently in Queues remain there. The Activity instances in the Future Events List do not change. Stroboscope, however, does reset the statistics kept by Queues, Activities, Collectors, and the system in general. RESETSTATS is a control statement that takes no arguments:

```
Syntax:   RESETSTATS;
Example:  RESETSTATS;
```

“RelTime” and “ActivityName.RelTotInst” are two new system-maintained variables that are useful after the execution of a statistical reset. RelTime keeps the simulation time elapsed since the last statistical reset. ActivityName.RelTotInst keeps the total number of instances of Activity ActivityName that have been created since the last statistical reset. These new variables are related to SimTime and ActivityName.TotInst; prior to the first statistical reset they are interchangeable with their counterparts.
Continuous statistics such as average waiting times in Queues and resource populations consider only the time elapsed since the statistical reset.

After a statistical reset, discrete statistics such as Activity inter-instantiation rates and customer visit times to hotels consider only new samples. The data for these new samples may straddle the instant of the statistical reset. Suppose that, in a run of the commuter transportation system of section 15.2 and shown in Figure 35 and Figure 36, Stroboscope executes a RESETSTATS statement at hour 12.00. The first worker to leave WorkersWait2 after the reset does so at hour 12.05. The worker had entered WorkersWait2 at hour 11.78. The first sample included in the visit statistics for workers to WorkersWait2 is 12.05-11.78=0.27 hours (not 12.05-12.00=0.05 hours). The 0.27 reflects 0.22 hours before the reset and 0.05 hours after the reset. Obviously, sampling a value of 0.05 would distort the statistics for worker visit times to WorkersWait2.

16.3.1 Batch Means Replications - Generic Format

The batch means method is a popular technique used to perform multiple replications of non-terminating processes. The first step in such a method is to warm-up the system until it reaches steady state and then discard the data collected during the warm-up period. The remainder of the simulation is broken up into batches of a predetermined length. At the end of each batch, the measures of performance of the system are used as statistical samples and the data collected during the batch is discarded. The last step is to build confidence intervals on the expected value of the statistical samples.

The most difficult issue in the application of the batch means method is the determination of the length of the warm-up period and batches. The warm-up period must be long enough so that the system attains steady state. The batch length must be long enough so that there is no correlation between batches. In practice, the length of the warm-up period and batches are determined graphically. It is usually better to over-estimate the length of the warm-up and batches than to under-estimate them.

The simplicity with which Stroboscope can perform batch means replications is illustrated by the following generic code structure:
/ Define the model’s input parameters
/ Include WarmUpTime, BatchLength, and NumBatches
/ i.e.:
VARIABLE WarmUpTime 12;
VARIABLE BatchLength 24;
VARIABLE NumBatches 20;

/ Define the network

/ Define element attributes

/ Initialize Queues

/ Warm-up the model
SIMULATEUNTIL SimTime>=WarmUpTime;

/ Define variables that return performance measures of interest.
/ i.e.:
VARIABLE ProductionRate Production/RelTime;

/ Define one persistent collector for each statistic of interest.
/ i.e.:
COLLECTOR ProdRateStats*;

/ Replicate and collect statistics in a loop
/ i.e.:
WHILE ProdRateStats.nSamples<NumBatches;

/ ResetStats at beginning of loop so that, if desired, detailed
/ statistics on the last batch are available for a report. This also
/ clears the statistics collected during warm-up (the first time this
/ body is executed).
RESETSTATS;

SIMULATEUNTIL RelTime>=BatchLength;

COLLECT ProdRateStats ProductionRate;

WEND;

/ Present results of replications and/or last batch
REPORT; /this will do both

The sections that follow illustrate model warm-up and batch means replications
by enhancing the commuter transportation system presented in section 15.2. Since the
commuter transportation system is highly parametric and the number of bus stops is not
known a priori, the simplicity with which batch-means can be implemented may be
hidden (due to the many loops and preprocessor replacements). Refer to the code above
as a starting point for simple models.
16.3.2 Simulation Warm-up Example

The following enhancements to the model presented in section 15.4.1 illustrate system warm-up. The modifications at the beginning of the model include the definition WarmUpSystem in the “statement execution switches” section and WarmUpLength in the “simulation run parameters” section:

/**Statement Execution Switches*******************************************************************************************/
VARIABLE GraphPopulation 1; /use 1 to graph population, 0 otherwise
VARIABLE AllowParallelLds 1; /use 1 to allow parallel loading/unloading
  /of several buses at the same bus stop
VARIABLE UseMinWaits 1; /use 1 to enforce minimum wait times at bus stops, 0 otherwise
VARIABLE WarmUpSystem 1; /use 1 to warm up the system before collecting statistics, 0 otherwise
/**Simulation Run Parameters*******************************************************************************************/
SEED 9111964; /comment or replace as necessary
IF WarmUpSystem;
  VARIABLE WarmUpLength 12; / (hours)
ENDIF;
VARIABLE SimulationLength 10; / (days)

The remainder of the model is the same until just before the original SIMULATEUNTIL in the “simulate and print report” section. This section is enhanced to run the simulation for a warm-up period and then reset the statistics before continuing:

/**Simulate and print report*******************************************************************************************/
IF WarmUpSystem;
  /run the simulation until the system warms up
  SIMULATEUNTIL SimTime>=WarmUpLength;
  /discard statistics collected so far
  RESETSTATS;
ENDIF;

/run the simulation until the desired time has elapsed
SIMULATEUNTIL SimTime>=SimulationLength*24;
/rest is just as before
Running the multiple bus model with these modifications discards the statistics collected during the first 12 hours. A run of the model with the same simulation seed produces the following result:

Stroboscope Model MultiBus.STR (1614346640)
Number of days simulated : 10.00

Summary statistics about waiting for the bus

<table>
<thead>
<tr>
<th>Stop-</th>
<th>-Waiting Times-(min)-</th>
<th>--Departures--</th>
<th>-No Fit In Bus-</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg StdDv Max</td>
<td>Total LftWr</td>
<td>Tot Ave Max</td>
</tr>
<tr>
<td>1</td>
<td>12.18 7.22 41.77</td>
<td>1083 77</td>
<td>263 3.42 9</td>
</tr>
<tr>
<td>2</td>
<td>12.70 7.59 39.96</td>
<td>1083 79</td>
<td>247 3.13 9</td>
</tr>
<tr>
<td>3</td>
<td>12.98 7.87 50.57</td>
<td>1083 108</td>
<td>367 3.40 10</td>
</tr>
<tr>
<td>4</td>
<td>12.08 7.51 36.11</td>
<td>1084 65</td>
<td>204 3.14 10</td>
</tr>
<tr>
<td>5</td>
<td>12.58 7.62 42.31</td>
<td>1082 96</td>
<td>309 3.22 11</td>
</tr>
</tbody>
</table>

Execution Time = 48.313 seconds
Processor Time = 47.9375 seconds

Figure 42 shows the population graph for the warmed-up model. The current population curve is identical to that of Figure 41. The average and standard deviation curves show accurate values much earlier because these statistics do not consider the warm-up period. In fact, the standard deviation curve in the model that is not warmed-up does not even get close to the long run value.

16.3.3 Example of Batch Means Replications

The commuter transportation system also provides a good example for batch means replications. Let us assume that it is necessary to estimate the daily maximum worker waiting time at the bus stops. Thus a range not wider than 5 minutes must be established such that it will include the mean daily maximum waiting times at the bus stops with a 95% confidence level.

In a real commuter transportation system the maximum waiting time at each bus stop can be recorded at the end of each day. After data for a few weeks is accumulated, it is used to determine the mean daily maximum waiting times and their standard deviation. These numbers establish 95% confidence intervals. If the confidence intervals are wider
than 5 minutes, more data is collected until confidence intervals of the appropriate width are obtained.

The procedure described above for a real system is exactly the same procedure that is used in a simulation model. The code that follows is based on the two-bus system described in section 15.4 and enhanced in the previous subsection. The code performs replications until the 95% confidence interval of the daily maximum waiting time at a designated control bus stop is narrower than 5 minutes. The confidence intervals for the other stops are calculated, but their width may be wider than 5 minutes.

Continuing with the parametric design of the model, the first part is modified to include the definition of $\text{MultipleRepls}$ in the section labeled “Statement Execution Switches”; and the definition of $\text{ConfLevel}$, $\text{BatchLength}$, $\text{MaxCIWidth}$, and $\text{ControlStop}$ in the section labeled “Simulation Run Parameters”:

*Figure 42 - Population Graph With 2 Small Buses and Statistical Reset at 12 Hours*
/**Statement Execution Switches*************************************************************************

VARIABLE GraphPopulation 0; /use 1 to graph population, 0 otherwise
VARIABLE AllowParallelLds 1; /use 1 to allow parallel loading/unloading
   /of several buses at the same bus stop
VARIABLE UseMinWaits 1; /use 1 to enforce minimum wait times at bus
   /stops, 0 otherwise
VARIABLE WarmUpSystem 1; /use 1 to warm up the system before
   /collecting statistics, 0 otherwise
VARIABLE MultipleRepls 1; /use 1 to perform multiple replications
   /using batch means, 0 otherwise

/**Simulation Run Parameters*************************************************************************

SEED 9111964; /comment or replace as necessary
IF WarmUpSystem;
   VARIABLE WarmUpLength 12; / (hours)
ENDIF;
IF MultipleRepls;
   VARIABLE ConfLevel 0.95;
   VARIABLE BatchLength 1; / (days)
   VARIABLE MaxCIWidth 5; / (minutes)
   VARIABLE ControlStop 1; /Bus Stop # whose CI must be < MaxCIWidth
ELSE;
   VARIABLE SimulationLength 10; / (days)
ENDIF;

The remainder of the model is the same up to the point where the
simulation is run and the report printed:

/ **Simulate and print report*************************************************************************

IF WarmUpSystem;
   /run the simulation until the system warms up
   SIMULATE UNTIL SimTime>=WarmUpLength;
   /discard statistics collected so far
   RESETSTATS;
ENDIF;
IF MultipleRepls;
   /Print a description of the table that will follow
   PRINT StdOutput "Max daily wait time at each stop in minutes\n\n";
   /Print blank spaces to use up the first column of table
   /column headings
   PRINT StdOutput "          ";
/This loop defines statistics collectors for the maximum daily waiting times at each bus stop. The loop also prints the column heading for each bus stop.
ASSIGN CurStop 1;
WHILE CurStop<=NumStops;

/Note the asterisk at the end of the collector name that makes it persistent and immune to statistical resets
COLLECTOR MaxDailyWaitAt$<CurStop>$_*$

/There is no "\n" at the end of the format string so that whatever is printed in the next loop continues in same line
PRINT StdOutput " | Stop %2.0f" CurStop;

ASSIGN CurStop CurStop+1;
WEND;

/Mark the end of the column titles and print the underline that corresponds to the first column
PRINT StdOutput "\n========";

/Loop to print the underlines that correspond to other columns
ASSIGN CurStop 1;
WHILE CurStop<=NumStops;

PRINT StdOutput "=|===== ===";

ASSIGN CurStop CurStop+1;
WEND;

/Mark the end of the row that contains the underlines
PRINT StdOutput "\n"

/put into a variable the confidence width of the daily maximum waiting time at the control bus stop
VARIABLE ControlConfidenceWidth
'Confidence[MaxDailyWaitAt$<$ControlStop>$$.SDVal,
ConfLevel,
MaxDailyWaitAt$<$ControlStop>$$.nSamples]*2';

/put into a variable the condition to continue replicating. Note that the condition ensures that at least 2 replications have been performed because the Confidence function is not defined for less than 2 samples.
VARIABLE ContinueReplicating
'MaxDailyWaitAt$<$ControlStop>$$.nSamples<2 | ControlConfidenceWidth>MaxCIWidth';

/The replications loop
WHILE ContinueReplicating;

/RelTime gives time since last ResetStats, thus this simulates one batch
SIMULATEUNTIL RelTime>=BatchLength*24;

/Print the day number in the first column, the day # is the number of samples collected plus 1.
PRINT StdOutput "  Day %4.0f"
MaxDailyWaitAt$<$ControlStop>$$.nSamples+1;
/loop to collect the maximum waiting times incurred during this
/batch at each bus stop.
/At the same time print the maximum waiting time for the day
/in the corresponding column.
ASSIGN CurStop 1;
WHILE CurStop<=NumStops;

  /Collect maximum waiting time
  COLLECT MaxDailyWaitAt$<CurStop>$
       60*WorkersWait$<CurStop>$.Worker.MxVstTm;

  /Print maximum waiting time, note that there is no "\\n"
  PRINT StdOutput " | %7.2f"
       60*WorkersWait$<CurStop>$.Worker.MxVstTm;

  ASSIGN CurStop CurStop+1;
WEND;

/Mark the end of the row that contains the data for this batch
PRINT StdOutput "\\n";

/reset statistics before doing another replication
RESETSTATS;
WEND;

/Print the underline that corresponds to the first column
PRINT StdOutput "=|========";

/Loop to print underlines under the other columns
ASSIGN CurStop 1;
WHILE CurStop<=NumStops;

   PRINT StdOutput "=|========"; /no "\\n"
   ASSIGN CurStop CurStop+1;
WEND;

/Break line and print "Average" as the label for the data that
/appears in this row
PRINT StdOutput "\\nAverage ";

/Loop to print Average for each bus stop
ASSIGN CurStop 1;
WHILE CurStop<=NumStops;

   /no line break after print
   PRINT StdOutput " | %7.2f" MaxDailyWaitAt$<CurStop>$$.AveVal;

   ASSIGN CurStop CurStop+1;
WEND;

/Break line and print "Std. Dev." as the label for the data that
/appears in this row
PRINT StdOutput "\\nStd. Dev. ";
ASSIGN CurStop 1;
WHILE CurStop<=NumStops;

  /no line break after print
  PRINT StdOutput " | %7.2f" MaxDailyWaitAt$<CurStop>$ SDVal;

  ASSIGN CurStop CurStop+1;
WEND;

/Break line and print "XX.XX% CIL" as the label for the data that
/appears in this row - Low side of Confidence Interval
PRINT StdOutput "\n%5.2f%% CIL" ConfLevel*100;

/Loop to print left side of Confidence Interval for each bus stop
ASSIGN CurStop 1;
WHILE CurStop<=NumStops;

  /no line break after print
  PRINT StdOutput " | %7.2f" '(MaxDailyWaitAt$<CurStop>$ AveVal-
      Confidence[ MaxDailyWaitAt$<CurStop>$ SDVal,
      ConfLevel,
      MaxDailyWaitAt$<CurStop>$ nSamples]);'

  ASSIGN CurStop CurStop+1;
WEND;

/Break line and print "XX.XX% CIH" as the label for the data that
/appears in this row - High side of Confidence Interval
PRINT StdOutput "\n%5.2f%% CIH" ConfLevel*100;

/Loop to print Right side of Confidence Interval for each bus stop
ASSIGN CurStop 1;
WHILE CurStop<=NumStops;

  /no line break after print
  PRINT StdOutput " | %7.2f" '(MaxDailyWaitAt$<CurStop>$ AveVal+
      Confidence[ MaxDailyWaitAt$<CurStop>$ SDVal,
      ConfLevel,
      MaxDailyWaitAt$<CurStop>$ nSamples]);'

  ASSIGN CurStop CurStop+1;
WEND;

ELSE; /not performing replications, just 1 single run

  /run the simulation until the desired time has elapsed
  SIMULATEUNTIL SimTime>=SimulationLength*24;

  PRINT StdOutput
    "Number of days simulated : %.2f\n\n" SimTime/24;

  PRINT StdOutput
    "Summary statistics about waiting for the bus\n\n";
The following is an output from the enhanced model:

**Stroboscope Model MBusRepl.STR (1083311520)**

Maximum daily waiting time at each bus stop in minutes

<table>
<thead>
<tr>
<th>Stop</th>
<th>Day 1</th>
<th>Day 2</th>
<th>Day 3</th>
<th>Day 4</th>
<th>Day 5</th>
<th>Day 6</th>
<th>Day 7</th>
<th>Day 8</th>
<th>Day 9</th>
<th>Day 10</th>
<th>Day 11</th>
<th>Day 12</th>
<th>Day 13</th>
<th>Day 14</th>
<th>Day 15</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>34.15</td>
<td>39.96</td>
<td>37.26</td>
<td>35.04</td>
<td>28.21</td>
<td>33.57</td>
<td>32.66</td>
<td>26.87</td>
<td>26.03</td>
<td>27.00</td>
<td>25.60</td>
<td>25.89</td>
<td>42.58</td>
<td>41.32</td>
<td>32.33</td>
</tr>
<tr>
<td>3</td>
<td>30.82</td>
<td>29.96</td>
<td>28.32</td>
<td>34.07</td>
<td>26.81</td>
<td>38.07</td>
<td>45.26</td>
<td>50.57</td>
<td>26.72</td>
<td>28.85</td>
<td>25.60</td>
<td>27.48</td>
<td>34.32</td>
<td>47.89</td>
<td>32.33</td>
</tr>
<tr>
<td>4</td>
<td>27.44</td>
<td>36.11</td>
<td>34.31</td>
<td>33.83</td>
<td>26.44</td>
<td>30.21</td>
<td>31.55</td>
<td>29.92</td>
<td>27.37</td>
<td>30.16</td>
<td>27.21</td>
<td>39.10</td>
<td>38.89</td>
<td>29.43</td>
<td>32.33</td>
</tr>
<tr>
<td>5</td>
<td>28.16</td>
<td>36.17</td>
<td>36.74</td>
<td>32.33</td>
<td>27.92</td>
<td>42.31</td>
<td>31.91</td>
<td>30.50</td>
<td>29.33</td>
<td>40.41</td>
<td>26.50</td>
<td>27.34</td>
<td>40.60</td>
<td>38.24</td>
<td>32.33</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>29.73</td>
<td>32.56</td>
<td>30.82</td>
<td>30.16</td>
<td>27.37</td>
<td>31.55</td>
<td>29.92</td>
<td>29.33</td>
<td>30.16</td>
<td>27.21</td>
<td>39.10</td>
<td>38.89</td>
<td>30.50</td>
<td>32.33</td>
</tr>
<tr>
<td></td>
<td>Std. Dev.</td>
<td>4.20</td>
<td>5.86</td>
<td>4.07</td>
<td>5.30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>95.00% CIL</td>
<td>27.41</td>
<td>29.32</td>
<td>31.68</td>
<td>33.38</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>95.00% CIH</td>
<td>32.05</td>
<td>35.81</td>
<td>30.45</td>
<td>36.32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Execution Time = 70.734 seconds
Processor Time = 68.7188 seconds
A close look at the results in the column for bus stop 1 may suggest that the first batch includes some transient data (i.e., the warm-up period was not long enough). Running the model with a warm-up period of 24 hours rather than 12 hours (same random number seeds) produces the following result:

<table>
<thead>
<tr>
<th>Stop 1</th>
<th>Stop 2</th>
<th>Stop 3</th>
<th>Stop 4</th>
<th>Stop 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Day 1</td>
<td>33.62</td>
<td>27.31</td>
<td>30.82</td>
<td>36.11</td>
</tr>
<tr>
<td>Day 2</td>
<td>26.87</td>
<td>39.96</td>
<td>28.32</td>
<td>34.31</td>
</tr>
<tr>
<td>Day 3</td>
<td>32.28</td>
<td>35.04</td>
<td>34.07</td>
<td>33.83</td>
</tr>
<tr>
<td>Day 4</td>
<td>27.64</td>
<td>28.21</td>
<td>27.67</td>
<td>26.74</td>
</tr>
<tr>
<td>Day 5</td>
<td>29.86</td>
<td>27.43</td>
<td>27.91</td>
<td>30.21</td>
</tr>
<tr>
<td>Day 6</td>
<td>27.48</td>
<td>33.57</td>
<td>38.07</td>
<td>30.39</td>
</tr>
<tr>
<td>Day 7</td>
<td>27.17</td>
<td>32.38</td>
<td>45.26</td>
<td>31.55</td>
</tr>
<tr>
<td>Day 8</td>
<td>26.63</td>
<td>26.87</td>
<td>50.57</td>
<td>29.92</td>
</tr>
</tbody>
</table>

Average: 28.94  31.35  35.34  31.63  33.70
Std. Dev.: 6.29  4.71  6.57  2.99  4.93
95.00% CIL: 26.07  27.41  26.11  29.13  29.58
95.00% CIH: 31.19  35.28  42.57  34.13  37.82

Execution Time = 40.812 seconds
Processor Time = 39.7656 seconds

16.4 The CLEAR Statement

The CLEAR statement is used to perform multiple replications of simulation models that have a natural finish.

Most construction processes are of this type (i.e., they start when work begins and come to a natural end when the work is complete). Performance measures of such processes are usually stochastic. For example, the cost per cubic meter of moved earth in an earth-moving operation may vary from job to job even if both jobs are practically identical. In such cases, the interest is the expected cost per cubic meter and its variability, rather than the outcome of one particular unit cost realization in a past project.

Simulation can be used to estimate the expected value and variance of the outcome for a terminating system. To do this, several simulations are run (from the start
to the finish of the process) and the outcome of each run is used as a statistical sample. The samples are then used to build a confidence interval on the expected value of the outcome.

The CLEAR statement completely clears the results of a previous simulation run so that another run can start. When Stroboscope executes a CLEAR statement it performs the following tasks:

- Resets all the statistics kept by the system except for those kept by persistent Collectors.

- Reassigns to SaveValues the values with which they were defined. The values of persistent SaveValues do not change.

- Destroys all existing Activity instances without terminating them, and empties the FEL.

- Destroys all the resources in the system.

- Resets the simulation clock (SimTime) and the relative clock (RelTime) to zero.

Essentially, after Stroboscope executes a CLEAR statement, the model is in the same condition as it was before Queues were initialized and the simulation was run.

16.4.1 Examples of Replications in Terminating Simulations

The CLEAR statement can be placed in a loop to simulate a model with different parameters each time. It can also be used to simulate a model with the same parameters every time so that confidence intervals on the measures of performance of interest can be established. Using a loop within a loop and statement preprocessing, the CLEAR statement can be used for both cases described above simultaneously.

Two of the examples that follow enhance the earth-moving operation of section 11.5. A third example implements a fully parametric probabilistic Critical Path Method model.

In order to enhance the original model of section 11.5 to include multiple replications, it is only necessary to modify the parts that define problem parameters,
initialize Queues, run the simulation, and print results. In addition, the statements that define Variables that aid in the computation of results need to be placed before the statements that initialize Queues.

The network for the earth-moving operation is reproduced in Figure 43. The code for the model described in section 11.5.1 has been reorganized as described above and is presented below:

```plaintext
/** Problem parameters and decision variables 
/  In multiple replications some additional parameters may be needed 
/  and others may need to be replaced 
*/
VARIABLE NumPshrs 3; / number of pushers 
VARIABLE Num651E 2; / number of big scrapers 
VARIABLE Num621E 9; / number of small scrapers 
VARIABLE PshrInitCost 900; / pusher initial cost ($) 
VARIABLE PshrHrCost 55; / pusher hourly cost ($/hr) 
VARIABLE OthrHrCost 200; / other hourly cost ($/hr) 
VARIABLE EarthWgt 15.7; / weight of earth (kN/m³) 
VARIABLE ShrkFct 0.95; / earth shrinkage factor 
VARIABLE InitDst 1000; / initial haul distance (M) 
VARIABLE FinDist 5000; / final haul distance (M) 
VARIABLE RdCrsSct 12.5; / road cross sectional area (M²) 
VARIABLE RollRst 0.03; / rolling resistance 
VARIABLE Grade 0.02; / hauling slope (uphill) 

/** Resource Types */
GENTYPE Earth; / ER 
GENTYPE Pusher; / PS
```

---

**Figure 43 - Earth-Moving Operation With Scrapers**
CHARTYPE Scraper Wgt Pow Cap MaxV HrCost Eff InitCost;/SC
/ kN kW m3 kM/hr $/hr $
SUBTYPE Scraper S621E 299 256 10.7 51 48 0.80 750 ;
SUBTYPE Scraper S651E 583 410 24.5 55 103 0.83 1300 ;
SAVEPROPS Scraper AmountLoaded;

/**Auxiliary global Variables and scraper VarProps
VARIABLE DstPerBV 'ShrkFct/RdCrsSct'; / (M/M2)
VARIABLE Distance 'InitDst+DstPerBV*MvdEarth.CurCount'; / (M)
VARPROP Scraper OptLdTm '125*(1+0.48*Ln[0.08*Distance/Pow])';
VARPROP Scraper OptPayLd 'Cap*(1+Cap/60*Ln[Distance/5000])';

/**Attributes of Load and related links
VARPROP Scraper ActLdTm 'OptLdTm*Pertpg[0.95,1,1.1]';
VARIABLE SpotTime 'Pertpg[24,36,95]';
DURATION Load 'SpotTime+Load.Scraper.ActLdTm';
ONRELEASE SC2 ASSIGN AmountLoaded 'OptPayLd*Max[Normal[1,0.3],0.25]';

/**Attributes of BkTrack and related links
DURATION BkTrack '15+0.40*Haul.Scraper.OptLdTm';

/**Attributes of Haul and related links
VARPROP Scraper GrossWgt 'Wgt+AmountLoaded*EarthWgt';
VARPROP Scraper HaulFrc 'GrossWgt*(Grade+RollRst)';
VARPROP Scraper ThHaulV 'Min[Pow*Eff/HaulFrc,MaxV/3.6]';
VARPROP Scraper ThHaulTm 'Distance/ThHaulV';
DURATION Haul 'Haul.Scraper.ThHaulTm*Normal[1,0.25]';

/**Attributes of Dump and related links
DURATION Dump 'Pertpg[24,36,78]';
RELEASEAMT ER1 'Dump.Scraper.AmountLoaded';

/**Attributes of Return and related links
VARPROP Scraper RetFrc Wgt*(RollRst-Grade);
VARPROP Scraper ThRetV 'Min[Pow*Eff/RetFrc,MaxV/3.6]';
VARPROP Scraper ThRetTm 'Distance/ThRetV';
DURATION Return 'Return.Scraper.ThRetTm*Normal[1,0.15]';

/**Variables to compute results of interest.
/ These were originally placed after the SIMULATEUNTIL statement.
/ Some helpers may need to be added for the multiple replication cases.
VARIABLE TotSetupCst 'NumPshrs*PshrInitCost+Num621E*S621E.InitCost+'
Num651E*S651E.InitCost';
VARIABLE TotHrs 'SimTime/3600';
VARIABLE TotHrCst    'NumPshrs*PshrHrCost+Num621E*S621E.HrCost+ 
                    Num651E*S651E.HrCost+OthrHrCost';
VARIABLE TotalCst    'TotSetupCst+TotHrs*TotHrCst';
VARIABLE Product     'MvdEarth.CurCount/TotHrs';
VARIABLE UnitCost    'TotalCst/MvdEarth.CurCount';

/**The sections that follow will be placed in a loop, but before 
/  entering the loop we will have to print the header of the output 
/  table here

/**Initialize Queues
INIT    Pshrs    NumPshrs;
INIT    Scrprs   Num651E  S651E;
INIT    Scrprs   Num621E  S621E;

/**Simulate until entire road segment is built
SIMULATEUNTIL 'Distance>=FinDist';

/**Print results
/  In the cases for multiple replications we will print all the values 
/  in one line. One line per replication
PRINT StdOutput "Number of pushers          : %7.0f\n" NumPshrs;
PRINT StdOutput "Number of 651E scrapers    : %7.0f\n" Num651E;
PRINT StdOutput "Number of 621E scrapers    : %7.0f\n" Num621E;
PRINT StdOutput "Total earth moved          : %10.2f bcm\n" MvdEarth.CurCount;
PRINT StdOutput "Total time required        : %10.2f hours\n" TotHrs;
PRINT StdOutput "Total cost of road segment : %10.2f $\n" TotalCst;
PRINT StdOutput "Production rate          : %10.2f bm3/hr\n" Product;
PRINT StdOutput "Unit cost              : %10.2f $/bm3\n" UnitCost;
PRINT StdOutput "Av. pusher wait         : %10.2f secs\n" Pshrs.AveWait;
PRINT StdOutput "Av. scraper wait         : %10.2f secs\n" Scrprs.AveWait;

Running this model produces output that shows the results from a single run. An 
example appears below:
**16.4.1.1 Replications that Simulate the Same Model with Different Configurations**

This example modifies the scraper model to run several replications, each with a different fleet configuration.

The “Model Parameters and Decision Variables” section needs to be modified. Instead of specifying the number of big haulers with \textit{Num651E} and the number small haulers with \textit{Num621E}, the number of alternatives is defined in Variable \textit{NumAlternatives} and the alternatives themselves in the \textit{Alternatives} Array. Each row in \textit{Alternatives} represents a combination of big and small haulers. The number of pushers remains fixed at three:

```plaintext
/** Problem parameters and decision variables */
// Instead of defining Num651E and Num621E, we define the number of alternatives to try and an Array containing the alternatives
VARIABLE NumAlternatives 14; /number of scraper combinations

// Each row in Alternatives contains a combination of scrapers.
// Column 1 is # of big scrapers, Column 2 is # of small scrapers
ARRAY Alternatives NumAlternatives 2
{ 1 5
  1 6
  1 7
  1 8
  1 9
  1 10
  1 11
  2 4
  2 5
  2 6
...}
```
The remaining problem parameters are unchanged

The section labeled “Variables to Compute Results of Interest” contains definitions of variables that use the “old” variables Num651E and Num621E. These, however, have been replaced with the Alternatives Array. Thus, at the beginning of this section Num651E and Num621 need to be redefined. A persistent SaveValue to hold the index into the Alternatives Array that contains the current alternative must also be defined:

/**Variables to compute results of interest.
/  Note that CurAlternative is persistent and that Num651E and Num621E
/  need to be defined here so they can be used to compute cost and other
/  outcomes of interest
SAVEVALUE CurAlternative* 0;
VARIABLE Num651E Alternatives[CurAlternative,0]; /number of 651E’s
VARIABLE Num621E Alternatives[CurAlternative,1]; /number of 621E’s

/ The remainder of the section is unchanged

Finally, the sections of code that initialize Queues, run the simulation, and print results need to be placed in a loop. Each iteration simulates an alternative and prints the results for that alternative in a row of a table. The table header is printed before entering the loop.

SEED '09111964'; /comment out or replace seed if needed

/  Print the header of the output table
PRINT StdOutput "
Pshr 651E 621E   Time  TCost  Rate  UCost  PshW  ScrW
hr $/bm3/hr $/bm3  secs  secs
====================================================================
"

/  Loop once for each alternative until all have been simulated
WHILE CurAlternative<NumAlternatives;

/  Clear the model for the next alternative
CLEAR;

/  Initialize the queues
INIT  Pshrs  NumPshrs;
INIT  Scrprs  Num651E  S651E;
INIT  Scrprs  Num621E  S621E;
/ Simulate until the 4 km segment is complete
SIMULATEUNTIL 'Distance>=FinDist';

/ Print row with results
PRINT StdOutput
"%4.0f %4.0f %4.0f %6.2f %6.0f %6.2f %5.3f %5.1f %5.1f\n"
NumPshrs
Num651E
Num621E
TotHrs
TotalCst
Product
UnitCost
Pshrs.AveWait
Scrprs.AveWait;

/ Increment index to current alternative
ASSIGN CurAlternative CurAlternative+1;
WEND;

The output of the model that tries several alternatives is shown below:

Stroboscope Model Scrpers5.str (1156356672)

<table>
<thead>
<tr>
<th>Pshr</th>
<th>651E</th>
<th>621E</th>
<th>Time</th>
<th>TCost</th>
<th>Rate</th>
<th>UCost</th>
<th>PshW</th>
<th>ScrW</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>hr</td>
<td>$</td>
<td>$/bm3/hr</td>
<td>secs</td>
<td>secs</td>
<td>$/bm3</td>
<td>secs</td>
<td>secs</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>166.11</td>
<td>125356</td>
<td>316.86</td>
<td>2.382</td>
<td>149.6</td>
<td>8.0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>145.63</td>
<td>118594</td>
<td>361.45</td>
<td>2.253</td>
<td>100.0</td>
<td>14.7</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>132.38</td>
<td>115680</td>
<td>397.64</td>
<td>2.198</td>
<td>63.7</td>
<td>27.0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>123.21</td>
<td>114978</td>
<td>427.17</td>
<td>2.185</td>
<td>100.0</td>
<td>14.7</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>116.77</td>
<td>115839</td>
<td>450.82</td>
<td>2.201</td>
<td>21.7</td>
<td>82.1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>111.60</td>
<td>117300</td>
<td>471.71</td>
<td>2.228</td>
<td>10.9</td>
<td>125.8</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>109.71</td>
<td>121521</td>
<td>479.85</td>
<td>2.308</td>
<td>4.6</td>
<td>177.1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>147.34</td>
<td>120718</td>
<td>357.27</td>
<td>2.293</td>
<td>153.8</td>
<td>7.4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>131.24</td>
<td>115487</td>
<td>401.13</td>
<td>2.194</td>
<td>103.3</td>
<td>15.0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>14</td>
<td>119.73</td>
<td>112649</td>
<td>439.63</td>
<td>2.140</td>
<td>67.1</td>
<td>25.8</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>110.83</td>
<td>111068</td>
<td>475.02</td>
<td>2.110</td>
<td>41.0</td>
<td>43.5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>106.42</td>
<td>112927</td>
<td>494.66</td>
<td>2.145</td>
<td>23.0</td>
<td>76.1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>17</td>
<td>103.60</td>
<td>115956</td>
<td>508.05</td>
<td>2.203</td>
<td>11.9</td>
<td>116.9</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>101.63</td>
<td>119616</td>
<td>517.88</td>
<td>2.273</td>
<td>4.9</td>
<td>172.5</td>
<td></td>
</tr>
</tbody>
</table>

Execution Time = 75.875 seconds
Processor Time = 74.2813 seconds

The results above, based on a single run for each alternative, indicate that the
lowest cost fleet configuration tried consists of three pushers, two 651E scrapers, and
seven 621E scrapers.

It is not necessary to specify the different alternatives to be tried using an Array.
The statements that initialize Queues, run the simulation and print results, for example,
can be placed in a nested loop controlled by $\text{Num621E}$ and $\text{Num651E}$. The following code achieves this:

\[
/ \quad \text{In the problem parameters section use the following instead of} \\
/ \quad \text{defining Alternatives and NumAlternatives:} \\

\text{VARIABLE MinBigScrapers 1;} \\
\text{VARIABLE MaxBigScrapers 2;} \\
\text{VARIABLE MinSmallScrapers 4;} \\
\text{VARIABLE MaxSmallScrapers 11;} \\
/ \quad . . .
\]

/ Change the definition of Num621E and Num651E to \\
/ persistent SaveValues, CurAlternative no longer needed:

\text{SAVEVALUE Num651E* MinBigScrapers;} \\
\text{SAVEVALUE Num621E* MinSmallScrapers;}
/ \quad . . .

/ Change the loop that does the replications into a nested loop. \\
/ The body of the inner loop changes only to increment Num621E instead \\
/ of CurAlternative. Num651E is incremented in the outer loop.

\text{WHILE Num651E<=MaxBigScrapers;}

\text{ASSIGN Num621E MinSmallScrapers;}
\text{WHILE Num621E<=MaxSmallScrapers;}

/ Clear the model for the next alternative \\
\text{CLEAR;}

/ Initialize the queues \\
\text{INIT Pshrs NumPshrs;} \\
\text{INIT Scrprs Num651E S651E;} \\
\text{INIT Scrprs Num621E S621E;}

/ Simulate until the 4 km segment is complete \\
\text{SIMULATEUNTIL 'Distance>=FinDist';}

/ Print row with results \\
\text{PRINT StdOutput} \\
\text{"%4.0f %4.0f %4.0f %6.2f %6.0f %6.2f %5.3f %5.1f %5.1f\n"} \\
\text{NumPshrs} \\
\text{Num651E} \\
\text{Num621E} \\
\text{TotHrs} \\
\text{TotalCst} \\
\text{Product} \\
\text{UnitCost} \\
\text{Pshrs.AveWait} \\
\text{Scrprs.AveWait;

ASSIGN Num621E Num621E+1;
ASSIGN Num651E Num651E+1;
WEND;

This version is almost the same as the first except that 16 alternatives are tried
instead of 14. The results in this case look as follows:

Stroboscope Model Scrpers6.str (1642343744)

<table>
<thead>
<tr>
<th>Pshr</th>
<th>651E</th>
<th>621E</th>
<th>Time</th>
<th>TCost</th>
<th>Rate</th>
<th>UCost</th>
<th>PshW</th>
<th>ScrW</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
<td>193.36</td>
<td>134616</td>
<td>272.27</td>
<td>2.557</td>
<td>222.2</td>
<td>3.6</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>5</td>
<td>165.43</td>
<td>124873</td>
<td>318.19</td>
<td>2.372</td>
<td>150.4</td>
<td>7.4</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>6</td>
<td>145.48</td>
<td>118486</td>
<td>361.78</td>
<td>2.251</td>
<td>99.8</td>
<td>14.1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>7</td>
<td>132.97</td>
<td>116161</td>
<td>395.88</td>
<td>2.207</td>
<td>61.5</td>
<td>26.1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>8</td>
<td>121.84</td>
<td>113811</td>
<td>432.00</td>
<td>2.162</td>
<td>38.3</td>
<td>49.1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>9</td>
<td>115.54</td>
<td>114736</td>
<td>455.56</td>
<td>2.180</td>
<td>21.9</td>
<td>79.6</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>10</td>
<td>112.19</td>
<td>117857</td>
<td>469.18</td>
<td>2.239</td>
<td>11.5</td>
<td>127.3</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>11</td>
<td>109.90</td>
<td>121712</td>
<td>478.93</td>
<td>2.312</td>
<td>4.5</td>
<td>177.9</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
<td>146.77</td>
<td>120284</td>
<td>358.65</td>
<td>2.285</td>
<td>152.4</td>
<td>7.5</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>5</td>
<td>129.91</td>
<td>114408</td>
<td>405.19</td>
<td>2.173</td>
<td>102.0</td>
<td>12.3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>6</td>
<td>119.22</td>
<td>112210</td>
<td>441.56</td>
<td>2.132</td>
<td>66.9</td>
<td>25.4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>7</td>
<td>110.78</td>
<td>111024</td>
<td>475.17</td>
<td>2.109</td>
<td>40.6</td>
<td>45.7</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>8</td>
<td>106.77</td>
<td>113268</td>
<td>492.96</td>
<td>2.152</td>
<td>24.0</td>
<td>77.3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>9</td>
<td>102.83</td>
<td>115189</td>
<td>511.91</td>
<td>2.188</td>
<td>12.3</td>
<td>119.1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>10</td>
<td>100.42</td>
<td>118338</td>
<td>524.32</td>
<td>2.248</td>
<td>5.1</td>
<td>168.7</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>11</td>
<td>100.95</td>
<td>124498</td>
<td>521.43</td>
<td>2.365</td>
<td>1.7</td>
<td>229.6</td>
</tr>
</tbody>
</table>

Execution Time = 87.985 seconds
Processor Time = 86.1406 seconds

### 16.4.1.2 Replications that Simulate a Model with the Same Parameters Each Time

This example modifies the scraper model of section 16.4.1 to run several
replications, each with the same configuration of big and small loaders. The interest here
is to obtain confidence intervals on the results of the alternative that seems to be the
cheapest (i.e., three pushers, 2 big scrapers, and 7 small scrapers).

The “Model Parameters and Decision Variables” section is first extended by
defining Variables for the maximum width and desired confidence level of the unit cost:
/**Problem parameters and decision variables

VARIABLE NumPshrs 3; /number of pushers
VARIABLE Num651E 2; /number of big scrapers
VARIABLE Num621E 7; /number of small scrapers
VARIABLE ConfIntWidth 0.01; /confidence interval width
VARIABLE ConfLevel 0.95; /confidence level

/ The rest of the section is unchanged

Somewhere before the loop that performs the multiple replications, it is necessary to define persistent Collectors that keep statistics on the results of each run:

/ Define persistent Collectors for outcomes of interest

COLLECTOR TimeStats*; /Time to complete project in hours
COLLECTOR TCostStats*; /Total cost of project in $
COLLECTOR RateStats*; /Production rate in bm3/hr
COLLECTOR UCostStats*; /Unit cost in $/bm3
COLLECTOR PshWStats*; /Average pusher waiting time in seconds
COLLECTOR ScrWStats*; /Average scraper waiting time in seconds

Also before the replications loop, it is necessary to print information regarding the specific fleet used, confidence level of the results, and the header for the table that will contain the results of each run:

/ Print General information about this run

PRINT StdOutput "Number of pushers %7.0f\n" NumPshrs;
PRINT StdOutput "Number of 651E scrapers %7.0f\n" Num651E;
PRINT StdOutput "Number of 621E scrapers %7.0f\n" Num621E;
PRINT StdOutput "Confidence level of results: %6.2f%\n\n" ConfLevel*100;

/ Print the header of the output table

PRINT StdOutput "Run  Time  TCost   Rate UCost  PshW  ScrW
hr   $   bm3/hr $/bm3 secs secs
===========================================\n";

The model continues with the While-block that performs the multiple replications. After each replication, the statistics of interest are collected and a row containing the results of the run is printed:

SEED '09111964'; /comment out or replace seed if needed
/ Loop until desired interval is accomplished
WHILE 'UCostStats.nSamples<3 | 
    Confidence[UCostStats.SDVal,ConfLevel,UCostStats.nSamples] < 
    ConfIntWidth/2';

/ Clear the model for the next replication
CLEAR;

/ Initialize the queues
INIT Pshrs NumPshrs;
INIT Scrprs Num651E S651E;
INIT Scrprs Num621E S621E;

/ Simulate until the 4 km segment is complete
SIMULATEUNTIL 'Distance>=FinDist';

/ Collect statistics of interest for this run
COLLECT TimeStats TotHrs;
COLLECT TCostStats TotalCst;
COLLECT RateStats Product;
COLLECT UCostStats UnitCost;
COLLECT PshWStats Pshrs.AveWait;
COLLECT ScrWStats Scrprs.AveWait;

/ Print row with results
PRINT StdOutput
  "%4.0f %6.2f %6.0f %6.2f %5.3f %5.1f %5.1f\n"
  UCostStats.nSamples
  TotHrs
  TotalCst
  Product
  UnitCost
  Pshrs.AveWait
  Scrprs.AveWait;
WEND;

After all the replications have been performed, the average, standard deviation, 
and low and high ends of the confidence interval for each of the statistics of interest are 
presented:

PRINT StdOutput "===========================================\n"
PRINT StdOutput " Ave %6.2f %6.0f %6.2f %5.3f %5.1f %5.1f\n"
  TimeStats.AveVal
  TCostStats.AveVal
  RateStats.AveVal
  UCostStats.AveVal
  PshWStats.AveVal
  ScrWStats.AveVal;

PRINT StdOutput " SD %6.2f %6.0f %6.2f %5.3f %5.1f %5.1f\n"
  TimeStats.SDVal
  TCostStats.SDVal
  RateStats.SDVal
  UCostStats.SDVal
  PshWStats.SDVal
  ScrWStats.SDVal;
A run of this model replicates the earth-moving operation as many times as necessary to obtain the desired confidence interval. The output that follows corresponds to a case that required 6 replications:

Stroboscope Model Scrpers7.str (1059687360)

<table>
<thead>
<tr>
<th>Run</th>
<th>Time (hr)</th>
<th>TCost ($111434)</th>
<th>Rate (473.19 $/bm3/hr)</th>
<th>UCost (42.117 $/bm3)</th>
<th>PshW (41.6 secs)</th>
<th>ScrW (44.7 secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>111.23</td>
<td>473.19</td>
<td>2.117</td>
<td>41.6</td>
<td>44.7</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>111.41</td>
<td>472.44</td>
<td>2.120</td>
<td>42.0</td>
<td>45.2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>111.28</td>
<td>473.03</td>
<td>2.118</td>
<td>41.8</td>
<td>43.2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>111.69</td>
<td>471.50</td>
<td>2.124</td>
<td>40.9</td>
<td>44.5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>111.35</td>
<td>472.70</td>
<td>2.119</td>
<td>40.7</td>
<td>46.1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>112.14</td>
<td>469.39</td>
<td>2.133</td>
<td>41.3</td>
<td>46.6</td>
<td></td>
</tr>
</tbody>
</table>

| Ave  | 111.52    | 472.04         | 2.122                  | 41.4                 | 45.0             |
| SD   | 0.34      | 1.43           | 0.006                  | 0.5                  | 1.2              |
| CIL  | 111.15    | 470.54         | 2.116                  | 40.8                 | 43.8             |
| CIH  | 111.88    | 473.54         | 2.128                  | 41.9                 | 46.3             |

Execution Time = 34.109 seconds
Processor Time = 33.375 seconds
16.4.1.3 Replicating a Parametric Probabilistic CPM Model

This example implements a parametric probabilistic CPM model. The network for the construction of a fast food outlet shown in Figure 44 is used to illustrate the model. The CPM network and specific data are defined in the input section of the code shown below:

![CPM Network for Construction of Fast Food Outlet](image)

Figure 44 - CPM Network for Construction of Fast Food Outlet

/***********************************************************************
/ Input Data Area, Replace as appropriate
/
/ The current data is from pg. 344 in "Precedence and Arrow Networking
/ Techniques for Construction" by Robert B. Harris, 1978. John Wiley &
/ Sons. It is summarized as follows:
/
/ Number Name Predecessors P0,Mode,P100
/ 1 Base Slab 3,6,12
/ 2 Wall Panels 1 4,6,9
/ 3 Parking Area 1 6,15,20
/ 4 Roof Trusses 2 1,2,5
/ 5 Landscaping 3 3,5,10
/ 6 Roofing 4 1,3,5
/ 7 Windows 6 1,2,4
/ 8 Doors 6 1,2,4
/ 9 Counter 7,8 3,8,10
/ 10 Walk-in Refrigerator 7,8 2,5,8
***************************************************************************/
/ 11     Sign                    5            3,4,6
/ 12     Counter Equipment     9            1,2,4
/ 13     Kitchen Equipment     7,8          4,10,15
/ 14     Floor Coverings       10,9,13      2,4,8
/ 15     Tables and Furnishings 14           5,10,15

/ The seed for the default stream. Comment out or change as appropriate.
SEED 9111964;

/ The number of replications to perform.
VARIABLE nReplications 1000;

/ This model defines a dummy START and a dummy END Activity
/ automatically. Thus it is not necessary to define these explicitly so
/ that the network has a single start and a single end node.

/ The number of Activities in the CPM network. Do not include the START
/ and END Activities that the model will create for you in this number.
/ Activities are numbered with consecutive numbers beginning with 1 (we
/ do not particularly care about the numbers assigned to the dummy START
/ and dummy END).
VARIABLE nActs 15;

/ The number of precedence relationships. They are assumed to be Finish
/ to Start. Do not include relationships from the START and to the END
/ activities.
VARIABLE nPrecedences 19;

/ Assume durations of CPM activities are given as in PERT.
/ Use one row for the duration parameters of each CPM activity.
/ Place P0 in first column, Mode in second, and P100 in third.
/ The first row, for example, indicates that the duration of CPM
/ Activity 1 is Pert[3,6,12]
ARRAY DurParams nActs 3
{
  3   6   12
  4   6   9
  6  15  20
  1   2   5
  3   5  10
  1   3   5
  1   2   4
  1   2   4
  3   8  10
  2   5   8
  3   4   6
  1   2   4
  4  10  15
  2   4   8
  5  10  15
};
Use one row for each Precedence relationship (i.e., each row is a link for an "Activity-on-Node" network).
Place predecessor activity in column 1 and successor activity in column 2. Do not include relationships to the START and from the END.
I.e., The first row indicates that there is a finish to start relationship from Activity 1 to Activity 2 (Base Slab to Wall Panels)

```
ARRAY Precedences nPrecedences 2
{
  1 2
  1 3
  2 4
  3 5
  4 6
  6 7
  6 8
  7 9
  8 9
  7 10
  8 10
  5 11
  9 12
  7 13
  8 13
  10 14
  9 14
  13 14
  14 15
};
```

The remaining part of the model is completely parametric and does not need to be changed in order to model a different CPM network. This is because the input data above provides a complete definition of the project.

16.4.1.3.1 Modeling a Generic CPM Network - A Example of a Stroboscope Code Generator

The following Stroboscope code determines the number of successors and predecessors for each CPM Activity. It then creates the Stroboscope network that corresponds to the CPM network. Thus, the model continues as follows:

```
/ nSuccessors array will keep the number of successors of CPM Activity i in element i-1
ARRAY nSuccessors nActs; /all elements initially zero

/ nPredecessors array will keep the number of Predecessors of Activity i in element i-1
ARRAY nPredecessors nActs; /all elements initially zero

/the number of activities with no predecessors
SAVEVALUE nStartingActs* 0;

/the number of activities with no successors
SAVEVALUE nEndingActs* 0;
```
/ this nested loop fills-in nSuccessors and nPredecessors with the appropriate values, it also determines nStartingActs and nEndingActs
SAVEVALUE i 0;
SAVEVALUE j 0;
WHILE i<nActs;
    ASSIGN j 0;
    WHILE j<nPrecedences;
        ASSIGN nPredecessors i
        'nPredecessors[i]+(i+1 == Precedences[j,1])';
        ASSIGN nSuccessors i
        'nSuccessors[i]+(i+1 == Precedences[j,0])';
    ASSIGN j j+1;
    WEND;
    ASSIGN nStartingActs nStartingActs+!nPredecessors[i];
    ASSIGN nEndingActs nEndingActs+!nSuccessors[i];
    ASSIGN i i+1;
WEND;

The model maps each CPM Activity to a series of Stroboscope nodes. On the forward pass, a CPM Activity is represented by a Consolidator followed by a Normal. On the backward pass, a CPM Activity is represented by another Consolidator. Figure 45, for example, shows the representation of Activity #9, “Counter”.

The Consolidator used for the forward pass for CPM Activity “X” is named \textit{fpConX}, the Normal that succeeds the Consolidator is named \textit{CpmX} and the link that joins the two is named \textit{ILX}. The Consolidator used for the backward pass for CPM Activity “X” is named \textit{bpConX}.

The operation of the model is straightforward. The ConsolidateWhen for \textit{fpConX} is set so that it consolidates after it has received one resource from each of the predecessors (through links named \textit{FSY}_X, where “Y” is the CPM Activity number of a predecessor). The release amount of \textit{ILX} is set to one so that \textit{CpmX} starts with only one resource. Thus, \textit{CpmX} releases 1 resource to each of its successors (through links named \textit{FSX}_Z, where “Z” is the CPM Activity number of a successor).

The purpose of \textit{bpConX} is to determine the late start and finish dates for CPM Activity “X” (using the same duration as in the forward pass). Its ConsolidateWhen is set so that it consolidates after it has received one resource from each of its successors (through links named \textit{bpLZ}_X, where Z is the CPM Activity number of a successor). At this time the late dates of the successors have been determined and are used to determine
the late dates of “X”. Since a CPM Activity can have many successors and therefore receive several resources, the release amounts for the links that leave \(bpConX\) need to be set to 1 (these links are named \(bpLX\_Y\), where \(Y\) is the CPM Activity number of a predecessor).

**Figure 45 - Representation of CPM Activity # 9 "Counter"**
Starting CPM Activities have no predecessors. The model creates a Combi named $Cpm0$ to act as their predecessor. $Cpm0$ is preceded by a Queue named $Start$. Initializing $Start$ with one resource allows the forward pass to begin.

Similarly, ending CPM Activities have no successors. The model creates a Consolidator named $End$ to act as their successor. $End$ consolidates when it has received a resource from each of the ending CPM Activities. This marks the end of the forward pass. $End$ then releases one resource to the backward pass Consolidator (i.e., $bpConY$) of each of the ending CPM Activities. This triggers the backward pass calculations.

The code below builds a Stroboscope network as described above. Additionally, the code defines some auxiliary SaveValues, persistent Collectors, and action events to gather statistics about each CPM Activity:

---

/ The only resource type in this network, basically a signal
GENTYPE Time;

/ Queue and Combi combination to represent the START Activity
QUEUE Start Time;
COMBI Cpm0;
LINK StartLink Start Cpm0;

/ Consolidator for END Activity, it finishes when all the Activities
/ with no successors have finished. At that point it triggers the
/ beginning of the backward pass.
CONSOLIDATOR End;
CONSOLIDATEWHEN End End.Time.Count>=nEndingActs;

/ A persistent Collector for the duration of the project. It samples
/ every time Consolidator "End" finishes.
COLLECTOR ProjectDur*;
ONEND End COLLECT ProjectDur SimTime;

/ Big loop where we define everything related to CPM Activities
ASSIGN i 1;
WHILE i<=nActs;

/ Each CPM Activity "i" is represented by a Consolidator "fpConi"
/ linked to a Normal "Cpmi" by a link "ILi", to act on the forward
/ pass. Another Consolidator "bpConi" acts on the backward pass.
CONSOLIDATOR fpCon$i$>
NORMAL Cpm$i$>
LINK IL$i$> fpCon$i$> Cpm$i$> Time;
CONSOLIDATOR bpCon$i$>

/ On the forward pass, CPM Activity "i" starts when Consolidator
/ "fpConi" ends. "fpConi" ends when it has received one resource
/ from each predecessor Activity (i.e., all predecessors have
/ finished). While "fpConi" may receive several resources, it only
/ passes one to "Cpmi". The parameters for the duration of the CPM
/ Activity are taken from the DurParams array.
CONSOLIDATEWHEN fpCon<i>:
    fpCon<i>.Time.Count>=<Max[1,nPredecessors[i-1]]>;
RELEASEAMT IL<i> 1;
DURATION Cpm<i>:
    'Pert[ DurParams[$<i-1>$,0],
        DurParams[$<i-1>$,1],
        DurParams[$<i-1>$,2]]';

/ The backward pass calculations for CPM Activity "i" are performed
/ when Consolidator "bpConi" ends. "bpConi" ends after all the
/ successors have ended (i.e., its backward pass calculations are
/ performed after the backward pass calculations of its successors).
CONSOLIDATEWHEN bpCon<i>:
    bpCon<i>.Time.Count>=<Max[1,nSuccessors[i-1]]>;

/ Each CPM Activity "i" has two SaveValues "LFi" and "FFi" that
/ represent the late finish and free float of the CPM Activity on
/ each replication. These values are initially set very high and are
/ reduced by appropriate actions to be defined later.
SAVEVALUE LF<i> 10^5;
SAVEVALUE FF<i> 10^5;

/ CPM Activities that have no predecessors must be linked from the
/ starting Activity, Cpm0.
IF !nPredecessors[i-1];
    LINK FS0_<i> Cpm0 fpCon<i> Time;
ENDIF;

/ CPM Activities that have no successors must be linked to the end
/ Activity, Consolidator End.
IF !nSuccessors[i-1];
    LINK FS_<i> End Cpm<i> End Time;
    LINK bpLEnd_<i> End bpCon<i> Time;
    RELEASEAMT bpLEnd_<i> 1;

/ Just before the end of the End Consolidator (i.e., the
/ earliest time at which we know the duration of the project in
/ the current replication), we update the Free Float and Late
/ Finish of the Activities that have no successors.
BEFOREEND End ASSIGN FF<i> SimTime-(Cpm<i>.FirstStart+Cpm<i>.AveDur);
BEFOREEND End ASSIGN LF<i> SimTime;

ENDIF;

/ Define persistent Collectors to keep the statistics of interest
/ for each of the CPM Activities
COLLECTOR EarlyStart<i>;
COLLECTOR LateStart<i>;
COLLECTOR EarlyFinish<i>;
COLLECTOR LateFinish<i>;
COLLECTOR Duration<i>;
COLLECTOR Critical<i>;
COLLECTOR TotalFloat<i>;
COLLECTOR FreeFloat<i>;

/ Collect the statistics when the backward pass for the Activity is done. Most of the numbers come straight from pre-defined system-maintained variables (except for the late finish and free float). Note that the value sampled for Criticality has a tolerance of 10^{-5} to compensate for accumulated floating point round-offs.

ONEND bpCon \$<i>\$ COLLECT EarlyStart \$<i>\$ Cpm \$<i>\$.FirstStart;
ONEND bpCon \$<i>\$ COLLECT LateStart \$<i>\$ LF \$<i>\$ - Cpm \$<i>\$.AveDur;
ONEND bpCon \$<i>\$ COLLECT EarlyFinish \$<i>\$ Cpm \$<i>\$.FirstStart + Cpm \$<i>\$.AveDur;
ONEND bpCon \$<i>\$ COLLECT LateFinish \$<i>\$ LF \$<i>\$ - Cpm \$<i>\$.AveDur;
ONEND bpCon \$<i>\$ COLLECT Duration \$<i>\$ Cpm \$<i>\$.AveDur;

ONEND bpCon \$<i>\$ COLLECT Critical \$<i>\$ 'Abs[LF \$<i>\$ - (Cpm \$<i>\$.FirstStart + Cpm \$<i>\$.AveDur)] < 10^{-5}';

ONEND bpCon \$<i>\$ COLLECT TotalFloat \$<i>\$ LF \$<i>\$ - (Cpm \$<i>\$.FirstStart + Cpm \$<i>\$.AveDur);
ONEND bpCon \$<i>\$ COLLECT FreeFloat \$<i>\$ FF \$<i>\$;

ASSIGN i i+1;
WEND;

/ Helper variables to make preprocessor replacements more readable, they assume "i" is the index into Precedences for the current relationship.
VARIABLE Pred Precedences \[i,0\];
VARIABLE Succ Precedences \[i,1\];

/ Loop where we establish the precedence relationships between the CPM Activities and define actions that update the free float and late finish of the predecessor based on data from the successor.
ASSIGN i 0;
WHILE i < nPrecedences;

/ the link that connects the predecessor CPM Activity to the successor CPM Activity on the forward pass. The predecessor automatically releases only one resource because the link that enters it was set to release 1 resource.
LINK FS \$<Pred>\$_\$<Succ>\$ Cpm \$<Pred>\$ fpCon \$<Succ>\$ Time;

/ The link that connects the successor CPM Activity to the predecessor CPM Activity on the backward pass. The release amount must be set to 1.
LINK bpL \$<Succ>\$_\$<Pred>\$ bpCon \$<Succ>\$ bpCon \$<Pred>\$ Time;
RELEASEAMT bpL \$<Succ>\$_\$<Pred>\$ 1;

/ We define an action so that when a successor CPM Activity starts the free float of the predecessor is updated (FF starts out high and is decremented to match the minimum of the lags of each precedence relationship – there is a separate action for each precedence, one defined per loop).
ONSTART Cpm \$<Succ>\$ ASSIGN FF \$<Pred>\$
'Min[FF \$<Pred>\$, SimTime - (Cpm \$<Pred>\$.FirstStart + Cpm \$<Pred>\$.AveDur)];'

/ Define an action that updates the late finish of the predecessor CPM Activity (LF starts out high and is decremented depending on the late finish and duration of the predecessor from each precedence relationship).
BEFOREEND bpCon \$<Succ>\$ ASSIGN LF \$<Pred>\$
Min[LF \$<Pred>\$, LF \$<Succ>\$ - Cpm \$<Succ>\$.AveDur];
ASSIGN i i+1;
WEND;

When performing multiple replications, Stroboscope flushes StdError after each replication (i.e., Stroboscope writes out any text accumulated in StdError’s buffer, regardless of the number of characters accumulated). This forces text similar to the following to be written out after each run:

Stopped at 45.6955 (Lack of Resources)
Model Cpm.STR Cleared
Simulating(9111964)...

The time required to run a typical probabilistic CPM simulation is very short compared to the time required to flush StdError. This is in contrast to most other simulations. As a consequence, the computer may spend 90% of its time flushing StdError and only 10% of its time processing the simulation model.

The SILENTREPLICATE statement can be used to tell Stroboscope that StdError should only be flushed when the entire model has been processed. In simulations of CPM networks this saves a significant amount of time:

Syntax: SILENTREPLICATE [LogicalExpression];
Example: SILENTREPLICATE;
Example: SILENTREPLICATE 0;
Example: SILENTREPLICATE 1;

If LogicalExpression is TRUE or omitted, Stroboscope flushes StdError only when model processing is complete. If LogicalExpression is FALSE, Stroboscope flushes StdError at the beginning and end of each execution of a SIMULATE or SIMULATEUNTIL.

The probabilistic CPM model concludes as follows:

/ Each Cpm simulation is very short. Flushing StdError after every run
/ wastes too much time.
SILENTREPLICATE;

/ The loop that performs the replications
WHILE ProjectDur.nSamples<nReplications;

/ Simply clear, initialize, and simulate. All statistics are sampled
/ on actions attached to events. These are a lot faster because they
/ are compiled as opposed to interpreted (they would be interpreted
/ if placed in the body of this loop as COLLECT statements).
CLEAR;
INIT Start 1;
SIMULATE;
WEND;

/ Need to replicate at least once to get any valid data
IF !ProjectDur.nSamples;
    PRINT StdOutput("No replications have run, report not printed");
ENDMODEL;
ENDIF;

/ Print number of replications; Average, and CI on project duration
PRINT StdOutput
    "Number of replications performed      : %.0f\n" ProjectDur.nSamples;
PRINT StdOutput
    "Average Project Duration             : %.2f\n" ProjectDur.AveVal;
PRINT StdOutput
    PRECOND ProjectDur.nSamples>=2
    "90\% Confidence interval on project duration : [% .2f, % .2f]\n"
    ProjectDur.AveVal-
    Confidence[ProjectDur.SDVal, 0.90, ProjectDur.nSamples]
    ProjectDur.AveVal+
    Confidence[ProjectDur.SDVal, 0.90, ProjectDur.nSamples];
PRINT StdOutput "\n\n";

/ Print header of output table
PRINT StdOutput
    "Act   Time    ESD    LSD    EFD    LFD     FF     TF %%Critic\n";
PRINT StdOutput
    "============================================================\n"

/print one output row for each CPM Activity
ASSIGN i 1;
WHILE i<=nActs;

    PRINT StdOutput
        "%.0f %.2f %.2f %.2f %.2f %.2f %.2f %.2f %.2f%%\n"
        i
    Duration$<i>$$.AveVal
    EarlyStart$<i>$$.AveVal
    LateStart$<i>$$.AveVal
    EarlyFinish$<i>$$.AveVal
    LateFinish$<i>$$.AveVal
    FreeFloat$<i>$$.AveVal
    TotalFloat$<i>$$.AveVal
    100.0*Critical$<i>$$.AveVal;

    ASSIGN i i+1;
WEND;

Running the above model produces the following output:

Stroboscope Model Cpm.STR (510465520)
Number of replications performed          : 1000
Average Project Duration                  : 44.86
90\% Confidence interval on project duration : [44.70, 45.03]
### Act | Time | ESD | LSD | EFD | LFD | FF | TF | %Critic
---|---|---|---|---|---|---|---|---
1 | 6.61 | 0.00 | 0.00 | 6.61 | 6.61 | 0.00 | 0.00 | 100.00%
2 | 6.16 | 6.61 | 6.61 | 12.77 | 12.77 | 0.00 | 0.00 | 100.00%
3 | 14.40 | 6.61 | 20.82 | 21.01 | 35.22 | 0.00 | 14.21 | 0.00%
4 | 2.31 | 12.77 | 12.77 | 15.09 | 15.09 | 0.00 | 0.00 | 100.00%
5 | 5.49 | 21.01 | 35.22 | 26.50 | 40.71 | 0.00 | 14.21 | 0.00%
6 | 2.99 | 15.09 | 15.09 | 18.08 | 18.08 | 0.00 | 0.00 | 100.00%
7 | 2.19 | 18.08 | 18.35 | 20.26 | 20.54 | 0.27 | 0.27 | 49.40%
8 | 2.18 | 18.08 | 18.36 | 20.25 | 20.54 | 0.28 | 0.28 | 50.60%
9 | 7.53 | 20.54 | 23.03 | 28.06 | 30.56 | 0.00 | 2.49 | 14.00%
10 | 4.98 | 20.54 | 25.58 | 25.51 | 30.56 | 5.04 | 5.04 | 0.10%
11 | 4.15 | 26.50 | 40.71 | 30.65 | 44.86 | 14.21 | 14.21 | 0.00%
12 | 2.16 | 28.06 | 42.71 | 30.22 | 44.86 | 14.64 | 14.64 | 0.00%
13 | 9.88 | 20.54 | 20.67 | 30.42 | 30.56 | 0.14 | 0.14 | 85.90%
14 | 4.36 | 30.56 | 30.56 | 34.91 | 34.91 | 0.00 | 0.00 | 100.00%
15 | 9.95 | 34.91 | 34.91 | 44.86 | 44.86 | 0.00 | 0.00 | 100.00%

----------
Execution Time = 12.938 seconds
Processor Time = 11.5 seconds

### 16.5 Recap

The RESETSTATS and CLEAR statements, together with the ability to define persistent SaveValues and Collectors, allow us to perform multiple replications in numerous ways. The actual multiple replication strategies that can be implemented depend on how the provided tools are used. This chapter presented examples that illustrate the batch-means method for non-terminating processes, two different strategies for terminating processes, and a code generator that can create the Stroboscope model for any probabilistic CPM model.
Chapter 17

Random Number Generator, Stream Management and Variance Reduction

This chapter introduces issues that relate to the production of random numbers and the management of multiple random number streams. This chapter also includes examples that illustrate their use in the implementation of variance reduction techniques.

Detailed explanations of the concepts presented here and their application can be found in (Law and Kelton 1991) and (Schriber 1990). This chapter concentrates on the Stroboscope tools for managing random number streams and implementing variance reduction techniques.

17.1 Random Number Generator

Non-deterministic systems contain components that exhibit random behavior but that follow certain probability distributions. In order to model these components, simulation systems need to obtain numbers drawn randomly from appropriate probability distributions. These numbers are called random variates.

The specific random variates that are distributed uniformly between 0 and 1 are called random numbers. Random numbers are very important because they can be transformed into random variates that follow all other distributions (e.g., normal, exponential, triangular, beta).
Random numbers can be obtained through numerous means that may involve such things as published tables, rolling dice, shuffling cards, etc. Random numbers can also be obtained through mathematical algorithms called random number generators. Stroboscope obtains random numbers by means of a random number generator introduced by (Lehmer 1951), and further developed by numerous researchers. Specifically, Stroboscope uses a “prime modulus multiplicative linear congruential generator” (PMMLCG) with a modulus of $2^{31} - 1$ and a multiplier of 630,360,016. This generator was found by (Fishman and Moore 1986) to be among the five best available.

The random number generator produces a sequence of uniformly distributed integers in the range 1 to $2^{31} - 2$. In this discussion, these integers are called IRNs or integer random numbers. An IRN normalized by $2^{31} - 1$, produces a random number. Thus, the random numbers produced by this generator are fractions in the open range 0 to 1 (or in the closed range $1/(2^{31} - 1)$ to $(2^{31} - 2)/(2^{31} - 1)$).

The sequence of IRNs produced by this generator is defined by the recursive formula

$$IRN(i) = (630,360,016 * IRN(i-1)) \mod (2^{31} - 1)$$

Starting with any IRN (i.e., IRN(1)) in the range 1 to $2^{31} - 2$, the generator produces a sequence of $2^{31} - 2$ IRN’s (i.e., IRN(i), $1 < i < 2^{31} - 1$) that cover all the integers in the range 1 to $2^{31} - 2$ without repeating any of them. Subsequent IRN’s produced by the generator repeat the first sequence of IRN’s exactly (i.e., for any i, IRN(i+2^{31} - 2) = IRN(i)).

The random number generator can be conceived as a circular chain with $2^{31} - 2$ rings. Each ring is associated with an IRN and with the random number obtained by dividing the IRN by $2^{31} - 1$. Since the chain is circular, no ring marks its start or end point. For practical purposes, position 0 is assigned to an arbitrarily chosen ring. The remaining rings are assigned positions 1 to $2^{31} - 2$ in the order in which they are encountered by walking the chain ring by ring.

Figure 46 shows a chain of random numbers where position 0 has been assigned to the ring with IRN 9111964. Notice that rings with positions 0 and 2,147,483,646 are the same.
17.2 The Random Number Streams

A random number Stream (with a capital ‘S’) is a pointer to one of the rings in the random number chain. Retrieving a random number from a Stream moves the pointer to the next ring and returns the random number associated with the new ring. The sequence of random numbers retrieved from a Stream is a stream (‘s’ not capitalized).

17.2.1 The Default Stream

By default, Stroboscope maintains only one pointer to the circular chain of rings produced by its random number generator. This pointer is the default Stream. Stroboscope retrieves one or more random numbers from the default Stream every time it evaluates any of the statistical sampling functions that do not take a Stream number as an argument (e.g., Rnd, Normal, Pert, Pertpg).
When the Stroboscope engine initializes, it associates the default Stream with a
ring that it chooses randomly. The default Stream can be explicitly pointed to any ring in
the chain, provided that the ring’s IRN is known. This is done with the SEED control
statement:

Syntax: SEED IntegerBetween1and2147483646;
Example: SEED 9111964;

The IRN associated with the default Stream can be obtained through the CurSeed
pre-defined system-maintained variable.

The last random number retrieved from the default Stream can be obtained
through the LastRnd function. LastRnd is undefined if a random number has never been
retrieved from the default Stream. In this case, the function returns zero.

17.2.2 Additional Streams

Stroboscope can maintain any number of Streams in addition to the default. The
number of additional Streams needs to be specified with the STREAMS control
statement:

Syntax: STREAMS NumberOfAdditionalStreams;
Example: STREAMS 14;
Example: STREAMS 7;

When Stroboscope executes the STREAMS statement it first destroys all
previously existing non-default streams and then creates NumberOfAdditionalStreams
Streams in addition to the default Stream. The additional Streams are numbered starting
with 1 (the default Stream is number 0). The first example creates Streams 1 through 14.
The second example destroys Streams 1 through 14 and then creates Streams 1 through
7.

When Stroboscope creates a Stream, it points it to a ring that is 100,000 rings
away from the ring pointed by the previous Stream. Thus, when the Streams are created,
Stream 1 points to a ring that is 100,000 rings away from the ring to which the default
Stream (Stream 0) points. Stream 2 points to a ring that is 100,000 rings away from the
ring to which Stream 1 points, and 200,000 rings away from the ring to which the default Stream points. Stroboscope points the remaining Streams to rings in a similar fashion.

The SEED statement only affects the default Stream. Thus, a SEED statement that is executed after a STREAMS statement does not affect the non-default streams.

All the Streams can be associated explicitly with a series of evenly spaced rings in the chain. This is done with the SEEDALL statement:

Syntax: \[\text{SEEDALL } \text{IntFrom1to2147483646} \{\text{SeparationInHundredThousands}\}\];
Example: \text{SEEDALL 9111964 4};
Example: \text{SEEDALL 9111964};

The SEEDALL statement points the default Stream to the ring whose IRN is \text{IntFrom1to2147483646}. It then points all the remaining Streams to rings separated an equal number of rings from each other. The separation is 100,000 multiplied by \text{SeparationInHundredThousands}. If \text{SeparationInHundredThousands} is not specified, it is assumed to be one. The first example points the default Stream to the ring with IRN 9111964. The rings pointed by all remaining streams are such that they are 400,000 rings apart. The second example is similar, except that the separation is 100,000 rings.

It is also possible to point a specific Stream to a specific ring, provided that the IRN that corresponds to the ring is known. This is done with the SEEDN statement:

Syntax: \[\text{SEEDN } \text{Stream} \text{IntFrom1to2147483646}\];
Example: \text{SEEDN 12 9111964};

The first argument, \text{Stream}, identifies the Stream that will point to the ring with IRN \text{IntFrom1to2147483646}. The example points stream number 12 to the ring with IRN 9111964. All other streams remain pointing to their respective rings.

Stroboscope retrieves one or more random numbers from a Stream every time it evaluates any of the statistical sampling functions that take a Stream number as an argument (e.g., \text{sRnd}, \text{sNormal}, \text{sPert}, \text{sPertpg}). These functions are similar to the functions that use the default Stream. The difference is that the letter "s" is prefixed to their names and that they take one extra argument that specifies the Stream to be used (which can be Stream 0).

The \text{sSeed[Stream]} function returns the IRN of the ring currently pointed by Stream \text{Stream}. When \text{Stream} is zero, this function is equivalent to \text{CurSeed}. 
The sLastRnd[Stream] function returns the last random number retrieved from Stream Stream. sLastRnd is undefined if a random number has never been retrieved from Stream Stream. In this case sLastRnd returns zero. When Stream is zero, sLastRnd is equivalent to LastRnd.

17.2.3 Antithetic Sampling

By default, the random number associated with a ring is obtained by dividing the ring’s IRN by $2^{31}-1$. It is sometimes convenient for the implementation of variance reduction techniques to obtain antithetic random numbers (i.e., complementary or completely “opposite”). Antithetic random numbers are obtained by subtracting the non-antithetic (i.e., the default or standard) random number from 1. For example, the random number associated with a ring whose IRN is 9111964, is $9111964/(2^{31}-1) = 0.00424309$, the corresponding antithetic random number is $1-0.00424309 = 0.99575691$.

Note that if $U$ is uniformly distributed between 0 and 1, so is $1-U$. Therefore, antithetic random numbers are also distributed uniformly between 0 and 1.

The ANTITHETICS control statement tells Stroboscope whether it should use standard or antithetic random numbers:

Syntax: ANTITHETICS [LogicalExpression];
Example: ANTITHETICS 1;
Example: ANTITHETICS 0;
Example: ANTITHETICS;

If LogicalExpression returns TRUE or is omitted, Stroboscope uses antithetic random numbers. If LogicalExpression returns FALSE, Stroboscope uses non-antithetic random numbers. The first and third examples tell Stroboscope to use antithetic random numbers. The second example tells Stroboscope to use standard random numbers. If the ANTITHETICS statement has never been used in a model, Stroboscope uses standard random numbers.

The Antithetics[] function returns the current status of antithetic sampling. If antithetics are on, the function returns TRUE. If antithetics are off, the function returns FALSE.
17.2.4 Statistical Considerations

The various random number streams in a simulation model should not overlap. This is statistically necessary in order to maintain independence between the random numbers retrieved from different Streams. By default, Stroboscope Streams point to rings that are 100,000 rings apart. This means that up to 100,000 random numbers can be retrieved from one Stream before using random numbers already retrieved by the subsequent Stream. This limits the number of completely independent Streams to 21,474.

If the number of random numbers retrieved from a particular Stream exceeds 100,000, then the separation between Streams must be increased so that streams are long enough. In these cases, the SEEDALL statement should be used to increase the separation to an appropriate multiple of 100,000.

It is usually not appropriate to point the random number Streams to arbitrarily selected rings (except once, before any random numbers are retrieved from any of the Streams). It is quite possible that doing so creates a stream that overlaps another stream. This removes the independence between the streams.

The OffSetSeed[baseIRN,PositionsAway] function returns the IRN that corresponds to a ring that is PositionsAway rings away from the ring whose IRN is baseIRN. This function can be used to seed Streams in models that require very sophisticated Stream control. The function should be used sparingly, however, since it is very slow (the function jumps IRNs by 100,000’s and then jumps IRN’s one by one to arrive at the target IRN). The expression OffSetSeed[9111964,2^31-3], for example, returns 1750151992 (see Figure 46).

17.3 Variance Reduction Techniques

Variance reduction techniques are used to reduce the number of replications required to achieve a given level of confidence in the results of a simulation model. Sometimes, as in the case of Common Random Numbers, they are a requirement for the appropriate comparison of alternative system configurations.
Variance reduction techniques generally involve precise random number stream management. The intention is to induce high positive or negative correlation between pairs of simulation runs. At the same time, complete independence must be maintained between the different sources of uncertainty in a model and between different pairs of runs.

The following observations apply to the implementation of most variance reduction techniques:

- Each source of uncertainty in a model should have a separate random number stream.
- The random number streams should be long enough so that they do not overlap.
- If possible, the statistical sampling functions should consume a fixed number of random numbers to generate a sample. The preferred method is the inverse transform. Methods that use acceptance-rejection should be avoided because they consume a random number of random numbers when generating a sample.
- The statistical sampling functions should be monotonic. That is, they should produce large samples when using large random numbers and small samples when using small random numbers.

Table 14 summarizes important information about the Stroboscope built-in sampling functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Arguments</th>
<th>Monotonic</th>
<th>Random Numbers Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beta, sBeta</td>
<td>Alpha, Beta, Stream</td>
<td>No</td>
<td>Variable</td>
</tr>
<tr>
<td>Erlang, sErlang</td>
<td>Order, Mean, Stream</td>
<td>Yes</td>
<td>Order</td>
</tr>
<tr>
<td>Exponential, sExponential</td>
<td>Beta, Stream</td>
<td>Yes</td>
<td>1</td>
</tr>
<tr>
<td>Gamma, sGamma</td>
<td>Alpha, Beta, Stream</td>
<td>No</td>
<td>Variable</td>
</tr>
<tr>
<td>Normal</td>
<td>Mean, Std. Deviation</td>
<td>No</td>
<td>Variable</td>
</tr>
<tr>
<td>Function</td>
<td>Arguments</td>
<td>Monotonic</td>
<td>Random Numbers Used</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------------------</td>
<td>-----------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>sNormal</td>
<td>Mean, Std. Deviation,</td>
<td>Yes</td>
<td>2 every other sample</td>
</tr>
<tr>
<td></td>
<td>Stream</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NormalInv</td>
<td>Mean, Std. Deviation,</td>
<td>Yes</td>
<td>1 if Rnd[] or sRnd[] are used as</td>
</tr>
<tr>
<td></td>
<td>Cumulative</td>
<td></td>
<td>Cumulative</td>
</tr>
<tr>
<td>StdNormalInv</td>
<td>Cumulative</td>
<td>Yes</td>
<td>1 if Rnd[] or sRnd[] are used as</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Cumulative</td>
</tr>
<tr>
<td>Pert, sPert</td>
<td>P0, Mode, P100, Stream</td>
<td>No</td>
<td>Variable</td>
</tr>
<tr>
<td>Pertpg, sPertpg</td>
<td>P5, Mode, P95, Stream</td>
<td>No</td>
<td>Variable</td>
</tr>
<tr>
<td>Rnd, sRnd</td>
<td>Stream</td>
<td>Yes</td>
<td>1</td>
</tr>
<tr>
<td>ScaledBeta,</td>
<td>LowBound, HighBound,</td>
<td>No</td>
<td>Variable</td>
</tr>
<tr>
<td>sScaledBeta</td>
<td>Alpha, Beta, Stream</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Triangular,</td>
<td>LowBound, Peak,</td>
<td>Yes</td>
<td>1</td>
</tr>
<tr>
<td>sTriangular</td>
<td>HighBound, Stream</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uniform,</td>
<td>LowBound, HighBound,</td>
<td>Yes</td>
<td>1</td>
</tr>
<tr>
<td>sUniform</td>
<td>Stream</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note that Table 14 contains four different functions that can be used to sample from the normal distribution. The Normal function is the fastest version. Unfortunately, it uses an acceptance-rejection method. As consequence, it is not monotonic and uses a variable number of random numbers to produce a sample.

The sNormal function uses a method due to (Box and Muller 1958). This method retrieves two random numbers to produce two random variates. It can therefore be used to synchronize streams in the implementation of variance reduction techniques. Additionally, sNormal is monotonic. Unfortunately, the random variates produced by sNormal fall on a spiral space and are not truly identical and independently distributed (Law and Kelton 1991). Despite this, the Box and Muller method is widely used.

The StdNormalInv and NormalInv functions are inverse cumulative approximations (the other functions mentioned are not approximations) of the normal distribution. They are the slowest functions, but the most suitable for the implementation of variance reduction techniques.
The two subsections that follow illustrate how to use Stroboscope to implement the most common variance reduction techniques: Antithetic Variates, and Common Random Numbers. These, however, are only examples of the variance reduction techniques that can be implemented with Stroboscope. The stream management tools discussed in the preceding section and those used to perform multiple replications discussed in the preceding chapter can be used to implement practically any variance reduction technique.

### 17.3.1 Antithetic Variates

Antithetic variates is a technique that induces negative correlation in the inputs to a pair of simulation runs, with the hope that the outputs are also negatively correlated. The outcomes of each run in a pair are averaged to obtain a single observation. These observations are used to estimate the expected value of the corresponding outcome.

Negative correlation means that one run in a pair should experience the opposite luck as the other run. Consider as an example a classic earth-moving operation. If the first loading time in one run is very close to the pessimistic duration, in the other run the first loading time should be very close to the optimistic duration. Similarly, if the 15th haul time in one run is on the high side of its distribution, the 15th haul time in the other run should be on the low side of the distribution.

When two runs are negatively correlated, the hope is that the same measure of performance for each run in a pair falls on either side of the expected value of the measure of performance. Assume that the true expected value for the cost per cubic meter of moved earth is $2.90/m^3$. If the non-antithetic run in a pair gives a result below $2.90/m^3$, then the corresponding antithetic run tends to produce a result above $2.90/m^3$. The average of the two values tends to be closer to $2.90/m^3$ than either value.

When properly applied, the total number of runs (standard + antithetic) required to obtain a given confidence interval is smaller than the number required using independent runs. Suppose that it takes 200 independent runs to obtain a 95% confidence interval on unit cost of moved earth that is narrower than $0.10/m^3$. An equivalent confidence interval should require well less than 100 pairs of standard-antithetic runs.
Antithetic variates should only be used to estimate the expected value of a measure of performance. The standard deviation, minimum, and maximum values obtained through antithetic variates do not reflect the distribution of the outcome.

17.3.1.1 Antithetic Variates Example - Job Site Snack Bar

The following example is a typical queuing problem. The snack bar at a job site operates for eight hours every day with two employees. The two employees help laborers that arrive at the snack bar at a rate of 30 per hour. The arrivals are exponentially distributed. Each employee can help one laborer at a time. The time required to help a laborer follows a normal distribution with a mean of 3 minutes and a standard deviation of 48 seconds (0.8 minutes). Laborers that cannot be helped when they arrive, join a single queue.

Laborers insist that their productivity is low because they have to wait too long before being helped at the snack bar. To this end, it is necessary to estimate the expected average waiting time that laborers experience before being helped. The estimate should be expressed as a 95% confidence interval at most 30 seconds wide.

Solution

Figure 47 shows the network for the snack bar. The snack bar employees are modeled as generic resources of type Employee and the laborers as generic resources of type Laborer. L3 and LaborerLeaves are shown for completeness; they are not necessary to run a simulation and obtain the desired confidence interval.

![Snack Bar Network](image)

**Figure 47 - Snack Bar Network**

The first part of the model is the definition of the model parameters:
VARIABLE ArrivalRate 30; /: arrivals per hour
VARIABLE MeanServiceTime 3; /: minutes
VARIABLE ServiceTimeSD 0.8; /: minutes
VARIABLE NumEmployees 2;
VARIABLE TimeLimit 8; /: hours
VARIABLE ConfLevel 0.95;
VARIABLE CIWidth 0.5; /: minutes

The next part is the definition of the resource types:

GENTYPE Employee; /EP
GENTYPE Laborer; /LB

The network itself would be defined next. The corresponding code is not shown here because it can be inferred directly from Figure 47.

This model has two sources of uncertainty: the laborer interarrival time and the time necessary to help each laborer. In order to synchronize the standard run with the antithetic run in each pair, different random number Streams must be used for each source of uncertainty. The interarrival times will use Stream 0 (the default Stream) and the help time will use Stream 1.

The following statements define the attributes necessary to model the arrival of laborers to the snack bar:

SEMAPHORE LaborersArrive !LaborersArrive.CurInst;
DURATION LaborersArrive sExponential[60/ArrivalRate,0];
RELEASEAMT L1 '1';

The following statement defines the time required to help laborers:

DURATION HelpLaborer NormalInv[MeanServiceTime,ServiceTimeSD,sRnd[1]];

The NormalInv function is used instead of the sNormal function because the latter retrieves random numbers in pairs (see Table 14). The third argument, sRnd[1], obtains a random number from Stream 1. This number is used as the cumulative from which the corresponding normal random variate is obtained. This approach is convenient because it is monotonic and uses exactly one random number per sample.

The model concludes with the experimentation section that implements the multiple replications of standard-antithetic pairs. Although the model uses only Streams 0 and 1, it uses a Stream management technique that requires the definition of 3 Streams. Figure 48 illustrates the random number scheme used in this model. It assumes that the
default random number Stream in the first pair of runs is seeded with 9111964. This random number will be used as a reference point. It is assigned position # 0.

Stream 0
IRN=9111964
Pos = 0

Stream 1
IRN=1481557890
Pos = 100000

Stream 2
IRN=482095861
Pos = 200000

Figure 48 - Random Number Scheme in the Snack Bar Model

The average number of laborers that arrive at the snack bar is 30 laborers/hour * 8 hours = 240 laborers. If the system is stable (i.e., the employees can keep up with the arrivals), the average number of laborers helped will be less than but very close to the average number of arrivals. Thus, on average, 240 random numbers will be retrieved from each Stream (well below the 100,000 stream length) on each run.

The simulation runs for a fixed length of 8 hours. The number of laborers that arrive in 8 hours varies from run to run because the interarrival times are stochastic. For this reason, the actual number of random numbers retrieved from each Stream varies from run to run. Moreover, it is not possible to determine which of the standard or antithetic run in a pair of runs had the most arrivals.
Figure 48 shows the number of random numbers used by each Stream in the first two pairs of runs. (These numbers were obtained by actually running the model and counting the total number of instances of LaborersArrive and HelpLaborers. They cannot be determined exactly without first running the simulation.) The figure indicates that the first non-antithetic run uses 223 random numbers from Stream 0 and 222 random numbers from Stream 1. The antithetic run paired to it uses 235 random numbers from Stream 0 and 234 random numbers from Stream 1.

In order to achieve independence, it is necessary to use completely different random numbers in each pair of runs. The first non-antithetic run uses the random numbers in positions 1-223 and 100,001-100,222. The antithetic run paired to it uses the random numbers in positions 1-235 and 100,001-100,234.

The Streams for the subsequent run-pairs cannot be seeded arbitrarily. Doing so makes it possible for the Streams to retrieve random numbers from positions that have already been used in the first pair of runs. Instead, the following approaches can be used.

The first approach is the most efficient in terms of not wasting stream positions but is the most tedious to implement. In this approach each Stream is seeded so that it starts at the highest position used by the same Stream in the previous run-pair. In the snack bar model, this means seeding Streams 0 and 1 so that they start at positions 235 and 100,234. This approach could be used for subsequent runs as long as:

1) The position occupied by Stream 0 does not eventually reach 100,000. This could happen after about 416 runs. (The expected number of runs can increase from 416 if the Streams are separated by a larger number of rings.)

2) For each Stream, the model must keep track of whether the non-antithetic or antithetic run in the run-pair used more random numbers. The corresponding IRN needs to be stored and used to seed the same Stream in the next run-pair.

This approach allows us to run close to \((2^{31}-2)/480 = 4,473,924\) independent run pairs. This requires us to separate the Streams by \((2^{31}-2)/2 = 1073741823\) by using a statement such as ‘SEEDALL 9111964 10737;’, or slower but marginally better ‘SEEDN 1 OffSetSeed[CurSeed,(2^31-2)/2];’.

The second approach is less efficient in terms of wasting stream positions but is also less tedious to implement. In this approach the model seeds Stream 0 so that it starts
where the last Stream stopped. The remaining Streams are pointed to rings 100,000 rings
apart. In the snack bar model, this means seeding Streams 0 and 1 so that they start at
positions 100,234 and 200,234. This method allows us to run about \((2^{31}-2)/100,240 =
21,423\) independent replications of the snack bar.

If the non-antithetic and the antithetic runs in a pair retrieve the same number of
random numbers from the last Stream, the Streams in the next pair can simply start
where the last Stream ends (without having to check which of the standard or the
antithetic run used more random numbers). Since this is not the case in the snack bar, the
model would need to keep track of whether the standard or the non-antithetic run in the
last pair retrieved the most random numbers from the last Stream. (In the first approach,
mentioned earlier, this has to be done for every Stream.)

Instead of doing this, the model uses an extra Stream from which it does not
retrieve random numbers. This stream is simply used to determine the seed to use in the
next pair of runs. Thus, in the second run the model seeds Stream 0 so that it starts at
position 200,000 (the last and only position occupied by Stream 2). The model then seeds
Streams 1 and 2 so that they start at positions 300,000 and 400,000 (Stroboscope
automatically separates Streams by 100,000 positions with the SEEDALL statement).
This procedure continues with each subsequent run. This scheme for controlling random
number streams allows us to run about \((2^{31}-2)/200,000 = 10,737\) independent replications.

The following code implements the process described above and prints a table
with the results of each run-pair as well as overall statistics:

```
/Use one more stream than actually needed so that
/the last stream can be used to seed all the streams
/in the next pair of runs.
STREAMS 2;

/Define a persistent SaveValue to store the seed for the
/next pair of runs
SAVEVALUE SeedSave* 9111964;

/Define a persistent SaveValue to store the result
/of the non-antithetic run
SAVEVALUE AvLaborerWaitSave* 0;

/Create a persistent Collector to accumulate the
/statistics of interest
COLLECTOR AvLaborerWait*;
```
/Define a variable that returns the current half-width
/of the confidence interval
VARIABLE CIHalfWidth 'Confidence[AvLaborerWait.SDVal,
   ConfLevel,
   AvLaborerWait.nSamples]';

/*Print description of the output we will get*/
PRINT StdOutput "Average waiting times for service\n"

/*Print the header of the output table*/
PRINT StdOutput "Day Standard Antithetic Average\n";
PRINT StdOutput "-------------------------------\n";

/*Loop until the desired confidence is obtained*/
WHILE 'AvLaborerWait.nSamples<3 | CIHalfWidth*2>CIWidth';

/*the non-antithetic run*/
ANTITHETICS 0; /turn off antithetics
SEEDALL SeedSave; /seed all streams 100,000 RNs apart
CLEAR;

/*initialize Queues and run the simulation*/
INIT Employees NumEmployees;
SIMULATEUNTIL SimTime>=TimeLimit*60;

/*save the non-antithetic performance measure of interest*/
ASSIGN AvLaborerWaitSave LaborersWait.AveWait;

/*the antithetic run*/
ANTITHETICS 1; /turn on antithetics
SEEDALL SeedSave; /seed all streams with the same
   /seeds used for the non-antithetic run
CLEAR;

/*initialize Queues and run the simulation*/
INIT Employees NumEmployees;
SIMULATEUNTIL SimTime>=TimeLimit*60;

/*Calculate the average of the non-antithetic and
   /antithetic runs and collect statistics on it*/
COLLECT AvLaborerWait
   (AvLaborerWaitSave+LaborersWait.AveWait)/2;

/*Print a row with the current pair's results*/
PRINT StdOutput "%3.0f %8.4f %10.4f %7.4f\n"
   AvLaborerWait.nSamples
   AvLaborerWaitSave
   LaborersWait.AveWait
   (AvLaborerWaitSave+LaborersWait.AveWait)/2;

/*Transfer the seed for the next run from stream 2*/
ASSIGN SeedSave sSeed[2];
WEND;

/*Print the Average, Standard Deviation, and confidence
   /interval on the average laborer waiting time*/
PRINT StdOutput "-----------------------------\n"
PRINT StdOutput "Average %7.4f\n"
   AvLaborerWait.AveVal;
Running the snack bar model as defined above produces the following output:

```
Stroboscope Model SnackBr1.str (758693440)

Average waiting times for service

Day Standard Antithetic Average
-----------------------------------
  1  1.2460   1.6216   1.4338
  2  1.0823   2.6003   1.8413
  3  1.7596   2.0177   1.8887
  4  1.8906   1.2705   1.5805
  5  2.3636   1.9510   2.1573
  6  2.0662   1.2424   1.6543
  7  2.7526   1.5413   2.1470
  8  0.9567   3.2051   2.0809
-----------------------------------
Average                  1.8480
Standard Deviation       0.2730
95% CI Low               1.6198
95% CI High              2.0762
-----------------------------------
Execution Time = 2.906 seconds
Processor Time = 1.1875 seconds
```

The results indicate that after 8 pairs of standard-antithetic runs there is a 95% chance that the average waiting time for service at the snack bar is between 1.62 and 2.08 minutes.

The effectiveness of antithetic variates depends on the model to which it is applied. The following output corresponds to a run of the snack bar model that does not use antithetic variates. The same random number seed and scheme were used to make the results comparable:
Average waiting times for service

<table>
<thead>
<tr>
<th>Day</th>
<th>Average Laborer Wait</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.2460</td>
</tr>
<tr>
<td>2</td>
<td>1.0823</td>
</tr>
<tr>
<td>3</td>
<td>1.7596</td>
</tr>
<tr>
<td>4</td>
<td>1.8906</td>
</tr>
<tr>
<td>5</td>
<td>2.3636</td>
</tr>
<tr>
<td>6</td>
<td>2.0662</td>
</tr>
<tr>
<td>7</td>
<td>2.7526</td>
</tr>
<tr>
<td>8</td>
<td>0.9567</td>
</tr>
<tr>
<td>9</td>
<td>2.3493</td>
</tr>
<tr>
<td>10</td>
<td>1.6569</td>
</tr>
<tr>
<td>11</td>
<td>1.7120</td>
</tr>
<tr>
<td>12</td>
<td>1.8651</td>
</tr>
<tr>
<td>13</td>
<td>1.1753</td>
</tr>
<tr>
<td>14</td>
<td>1.1094</td>
</tr>
<tr>
<td>15</td>
<td>1.6333</td>
</tr>
<tr>
<td>16</td>
<td>1.1039</td>
</tr>
<tr>
<td>17</td>
<td>1.4858</td>
</tr>
<tr>
<td>18</td>
<td>4.5730</td>
</tr>
<tr>
<td>19</td>
<td>1.8759</td>
</tr>
<tr>
<td>20</td>
<td>1.7699</td>
</tr>
<tr>
<td>21</td>
<td>2.4176</td>
</tr>
<tr>
<td>22</td>
<td>1.8585</td>
</tr>
<tr>
<td>23</td>
<td>1.9313</td>
</tr>
<tr>
<td>24</td>
<td>3.0124</td>
</tr>
<tr>
<td>25</td>
<td>2.6504</td>
</tr>
<tr>
<td>26</td>
<td>2.4817</td>
</tr>
<tr>
<td>27</td>
<td>1.3059</td>
</tr>
<tr>
<td>28</td>
<td>2.3963</td>
</tr>
<tr>
<td>29</td>
<td>0.8664</td>
</tr>
<tr>
<td>30</td>
<td>3.9006</td>
</tr>
<tr>
<td>31</td>
<td>2.9167</td>
</tr>
<tr>
<td>32</td>
<td>2.9130</td>
</tr>
<tr>
<td>33</td>
<td>1.1139</td>
</tr>
<tr>
<td>34</td>
<td>1.8810</td>
</tr>
<tr>
<td>35</td>
<td>3.4552</td>
</tr>
<tr>
<td>36</td>
<td>1.8379</td>
</tr>
<tr>
<td>37</td>
<td>1.8902</td>
</tr>
<tr>
<td>38</td>
<td>1.4937</td>
</tr>
<tr>
<td>39</td>
<td>1.5985</td>
</tr>
<tr>
<td>40</td>
<td>1.7080</td>
</tr>
<tr>
<td>41</td>
<td>1.5490</td>
</tr>
<tr>
<td>42</td>
<td>0.9039</td>
</tr>
<tr>
<td>43</td>
<td>2.6910</td>
</tr>
</tbody>
</table>

Average: 1.9814
Standard Deviation: 0.8076
95% CI Low: 1.7329
95% CI High: 2.2299

Execution Time = 7.75 seconds
Processor Time = 3 seconds
The results indicate that after 43 standard runs there is a 95% chance that the average waiting time for service at the snack bar is between 1.73 and 2.23 minutes. With this particular model and seeds, antithetic variates are clearly effective. Antithetic variates obtained in 16 runs (8 pairs) the same confidence interval that required 43 standard runs. Note, however, that the standard deviation of 0.27 minutes obtained from the paired standard-antithetic runs underestimates the variability of the average waiting time considerably. The standard deviation of 0.81 minutes produced by the standard runs is a much better indicator of the variability.

17.3.2 Common Random Numbers

Common random numbers or CRN is a variance reduction technique useful for the comparison of alternative system configurations. CRN is an attempt to compare different systems under equal conditions so that any observed differences in performance are due to differences in system configurations and not due to different conditions.

CRN is described in detail in (Law and Kelton 1991) and (Schriber 1990). A very detailed example of CRN applied to the comparison of alternative tunneling construction methods, and the corresponding Stroboscope model, appears in (Ioannou and Martinez 1996).

CRN is similar to antithetic variates but differs in the following issues:

• Each run in a pair of runs corresponds to a different system configuration.

• CRN induces high positive correlation between the runs in a pair instead of high negative correlation.

• Instead of collecting statistics on the average of the measures of performance in each run in a pair, CRN collects statistics on their difference.

The tools necessary to implement CRN in Stroboscope have already been discussed. They are essentially the same as those used to implement antithetic variates.

An important issue in the implementation of CRN is that the simulation models for each alternative system configuration may be different. Implementing CRN requires us to switch models in order to make one run in a pair correspond to one alternative and...
to make the other run correspond to the other alternative. How to do this merits discussion since it may not be initially evident (even though the required tools have already been discussed).

In the simplest case, the alternatives differ only in the resources used. For example, one alternative in an earth-moving operation could be to use two small loaders and the other alternative could be to use one big loader. Switching models in these cases is fairly simple. It only involves initializing Queues differently for each configuration.

The replication loop in a typical case may look as follows:

```
SAVEVALUE SeedStore* 9111964;
SAVEVALUE CostWith2SmallLdrs* 0;
COLLECTOR DiffInCost*;
WHILE DiffInCost.nSamples<nReplications;
  /first alternative
  SEEDALL SeedStore;
  CLEAR;
  INIT LoadersWait 2 SmallLdr; /this differs between alternatives
  INIT HaulersWait NumHaulers;
  INIT SoilToMove AmountOfSoil;
  SIMULATE;
  ASSIGN CostWith2SmallLdrs TotalCost; /remember this result
  /second alternative
  SEEDALL SeedStore; /seeds for 2nd alternative are same as for 1st
  CLEAR;
  INIT LoadersWait 1 BigLdr; /this differs between alternatives
  INIT HaulersWait NumHaulers;
  INIT SoilToMove AmountOfSoil;
  SIMULATE;
  /collect statistics on the difference in cost
  COLLECT DiffInCost TotalCost-CostWith2SmallLdrs;
  ASSIGN SeedStore sSeed[4]; /select seed for the next pair of runs
WEND;
```

Another simple case is when the networks for the models being compared are the same, but some modeling element attributes differ between them. Suppose that in an earth-moving operation with scrapers, the alternatives are to use different queuing disciplines for the scrapers. One case gives preference to scrapers with high power and the other gives preference to scrapers with emptier tanks. The replication loop in a typical case may look as follows:
SAVEVALUE SeedStore* 9111964;
SAVEVALUE CostWithPowerDiscipline* 0;
COLLECTOR DiffInCost*;

WHILE DiffInCost.nSamples<nReplications;
  /preference to high powered scrapers
  SEEDALL SeedStore;
  CLEAR;
  DISCIPLINE Scrapers -Pow; /this differs between alternatives
  INIT Scrapers NumBigScrapers S651E;
  INIT Scrapers NumSmallScrapers S621E;
  INIT Pushers NumPushers;

  SIMULATEUNTIL EarthMoved>=AmntOfEarthToMove;
  ASSIGN CostWithPowerDiscipline TotalCost; /remember this result

  /preference to scrapers with emptier tanks
  SEEDALL SeedStore; /seeds for 2nd alternative are same as for 1st alternative
  CLEAR;
  DISCIPLINE Scrapers Fuel Left; /this differs between alternatives
  INIT Scrapers NumBigScrapers S651E;
  INIT Scrapers NumSmallScrapers S621E;
  INIT Pushers NumPushers;

  SIMULATEUNTIL EarthMoved>=AmntOfEarthToMove;
  /collect statistics on the difference in cost
  COLLECT DiffInCost CostWithPowerDiscipline-TotalCost;

  ASSIGN SeedStore sSeed[4]; /select seed for the next pair of runs

WEND;

In the above code, the only difference between the two alternatives is the Discipline attribute of the Scrapers Queue. Before running each alternative, the Discipline is set to the appropriate expression. The same procedure can be applied to change other modeling element attributes such as Semaphores, Durations, and DrawUntils.

Similarly simple is the case where the difference between the alternatives depends exclusively on data. In such cases instead of setting different attributes, the data is changed between runs. Assume that two different construction methods are evaluated in the model for the construction of a tunnel. In one method, the initial support changes when the geology changes. The cost to change the initial support is obtained by using the current and next ground classes as indexes into a matrix. In the other method, the initial support never changes, and thus the cost is never incurred. The replication loop in a typical case may look as follows:
SAVEVALUE  SeedStore* 9111964;
SAVEVALUE  CostWithNATM* 0;
COLLECTOR  DiffInCost*;

/There are three GroundClasses
ARRAY  CostToSwitchInitialSupport 3 3; / create array filled with zeroes

WHILE  DiffInCost.nSamples<nReplications;
    /Adaptive method
    SEEDALL  SeedStore;
    CLEAR;
    ASSIGN  CostToSwitchInitialSupport { 0 400 700
                                          200 0  550
                                          300 150 0
                                      } ; /this differs between alternatives
    SIMULATE;
    ASSIGN  CostWithNATM  CostOfTunnel; /remember this result

    /Conventional method
    SEEDALL  SeedStore; /seeds for 2'nd alternative are same as for 1'
    CLEAR;
    ASSIGN  CostToSwitchInitialSupport { 0 0  0
                                          0 0  0
                                          0 0  0
                                      } ; /this differs between alternatives
    SIMULATE;
    /collect statistics on the difference in cost
    COLLECT  DifInCost  CostWithNATM-CostOfTunnel;
    ASSIGN  SeedStore  sSeed[4]; /select seed for the next pair of runs
WEND;

The most complex case is when the networks for each method are completely different. In order to switch models in these cases, two separate networks are defined in the same model file. (This is similar to the case where a network fragment represents a clock, and a separate network fragment represents the actual model, both in the same file.) Care must be taken so that there is no conflict between the names of the modeling elements of one network and the other. Assume that two different methods of moving earth are being compared. One method uses pushers and scrapers while the other uses loaders and trucks. The skeleton for a CRN model file in this case may look as follows:

/Define the earth-moving operation with scrapers and pushers
/entirely, including modeling element attributes. But no Queue
/initializations. Prefix all modeling element names with "s_" just
/in case there is a similarly named element in the network for the
/other alternative.
/Define the earth-moving operation with loaders and trucks entirely, including modeling element attributes. But no Queue initializations. Prefix all modeling element names with "t_" just in case there is a similarly named element in the network for the other alternative.

SAVEVALUE SeedStore* 9111964;
SAVEVALUE CostWithScrapers* 0;
COLLECTOR DiffInCost*;

WHILE DiffInCost.nSamples<nReplications;

/using scrapers and pushers
SEEDALL SeedStore;
CLEAR;
INIT s_Scrapers NumBigScrapers S651E;
INIT s_Scrapers NumSmallScrapers S621E;
INIT s_Pushers NumPushers;

/if we don’t initialize the Queues related to the other alternative, nothing will happen in that part of the model while this part is simulating

SIMULATEUNTIL EarthMoved>=AmntOfEarthToMove;
ASSIGN CostWithScrapers s_TotalCost; /remember this result

/using loaders and trucks
SEEDALL SeedStore; /seeds for 2nd alternative are same as for 1st
CLEAR;
INIT t_LoadersWait 1 BigLdr;
INIT t_HaulersWait NumHaulers;
INIT t_SoilToMove AmountOfSoil;

/if we don’t initialize the Queues related to the other alternative, nothing will happen in that part of the model while this part is simulating

SIMULATE;
/collect statistics on the difference in cost
COLLECT DiffInCost t_TotalCost-CostWithScrapers;

ASSIGN SeedStore sSeed[4]; /select seed for the next pair of runs

WEND;

The examples presented above illustrate several approaches that can be used to implement CRN. Stroboscope provides tools that allow us to implement numerous other approaches. The possibilities depend on the imagination of the modeler.
Chapter 18
Extending Stroboscope — Add-Ons

The Stroboscope simulation language as described in the preceding chapters is very powerful and allows us to model a wide range of discrete event systems. It is possible, however, to extend Stroboscope beyond the capabilities presented earlier. This chapter describes how to create and use statements and functions written with standard compiled languages such as C or C++.

Custom statements and functions reside in Dynamic Link Libraries or DLLs. A DLL that contains one or more statements or functions developed for Stroboscope is a Stroboscope Add-On. Stroboscope Add-Ons can be designed for use with specific models, or they can be general extensions that become part of the Stroboscope language and can subsequently be used for any model.

This chapter consists of two parts. The first explains how to access the statements and functions defined in an Add-On. It presents an example that uses an Add-On for probabilistic analysis of CPM networks. The second part explains how to write Add-Ons using compiled languages. It includes the source code (in C++) for the CPM Add-On used in the example of the first part.

18.1 Using Stroboscope Add-Ons

In order to use the statements and functions defined in an Add-On it is necessary to know the name and location of the Add-On DLL. This is required to load the Add-On or to declare aliases to the statements and functions defined in the Add-On. The use of
the specific statements and functions defined in an Add-On are defined by the developer of the Add-On. Typically, and Add-On will have a reference manual or user’s guide that explains how to use it.

Full-fledged Add-Ons are loaded into Stroboscope with the LOADADDON statement:

Syntax: LOADADDON AddOnDllName;
Example: LOADADDON CpmAddOn.dll;
Example: LOADADDON E:\Strobos\Add-Ons\CpmAddOn.dll;

*AddOnDllName* is the name and location of the Add-On DLL. It is not necessary to use the fully qualified path name of the Add-On if the Add-On is contained in the current directory, in the directory that contains the Stroboscope engine, or in a directory in the standard executable search path for the computer. Both examples load an Add-On contained in file “CpmAddOn.dll”. LOADADDON statements are executed in the order in which they appear in the model file, but before executing any other statements.

Once an Add-On is loaded, the statements, functions, and pre-defined variables defined within it are available for use as if they were built into the Stroboscope language.

The “CpmAddOn” Add-On is actually included with Stroboscope and its source code is presented at the end of this chapter. Reference information about the Add-On is presented below.

### 18.1.1 Probabilistic CPM Add-On for Stroboscope — Reference

The Probabilistic CPM Add-on for Stroboscope is contained in file “CpmAddOn.dll”. This Add-On allows us to define CPM networks with stochastic durations and to obtain various statistics about the project and its Activities. The expressions for the durations of the CPM Activities can use the pre-defined variables provided by the Add-On. The CPM network to be modelled does not have to have a single starting activity and a single finishing activity (a common CPM requirement). The CPM Add-On precedes every network with a dummy START and concludes it with a dummy FINISH.
18.1.1.1 CPM Add-On Statements

The CPM Add-On includes the following statements:

18.1.1.1.1 CPMACTIVITY

Syntax:  
CPMACTIVITY CpmActivityName DurationExpression;
Example:  
CPMACTIVITY ParkingArea Pert[6,15,20];

CPMACTIVITY defines CpmActivityName as a CPM Activity with a duration determined by DurationExpression. The example defines CPM Activity ParkingArea with a duration sampled from a Beta distribution defined by P0=6, Mode=15, and P100=20.

When the Add-On executes the CPMACTIVITY statement it creates a Consolidator linked to a Normal. These represent the CPM Activity. The Add-On additionally defines several persistent Collectors to keep statistics related to the CPM Activity.

18.1.1.1.2 PRECEDENCE

Syntax:  
PRECEDENCE Predecessor Successor;
Example:  
PRECEDENCE ParkingArea Landscaping;

PRECEDENCE defines a finish-to-start relationship from Predecessor to Successor, both of which must have been defined previously with the CPMACTIVITY statement. The example indicates that Landscaping cannot start until ParkingArea has finished.

When the Add-On executes the PRECEDENCE statement, it defines a link from the Normal attached to the predecessor CPM Activity to the Consolidator attached to the successor CPM Activity.

18.1.1.1.3 CPMREPLICATE

Syntax:  
CPMREPLICATE NumberOfReplications;
Example:  
CPMREPLICATE 1000;
CPMREPLICATE simulates the CPM network $NumberOfReplications$ times and then prints the report described in the CPMREPORT statement below.

When the Add-On executes the CPMREPLICATE statement it enters a loop that repeatedly clears the model and executes the DOCPM statement described below. The Add-On then executes the CPMREPORT statement.

### 18.1.1.1.4 DOCPM

**Syntax:** DOCPM;

**Example:** DOCPM;

DOCPM does a time-based forward pass (i.e., as the CPM Activities start and finish the simulation clock advances) and then an instantaneous backward pass of the CPM network. This statement simply initializes a private Queue defined by the Add-On and then executes the SIMULATE statement. It is useful when a CPM network and a standard Stroboscope network are combined and related to each other. (I.e., the durations for CPM Activities access resource or state information from the Stroboscope network or vice-versa.) CPMREPLICATE cannot be used for this purpose because it clears the model on its own without giving the opportunity to initialize Queues related to the Stroboscope network.

### 18.1.1.5 CPMREPORT

**Syntax:** CPMREPORT;

**Example:** CPMREPORT;

CPMREPORT prints a report showing the average and 90% confidence interval on the project duration. Additionally, CPMREPORT prints the average duration, early start date, late start date, early finish date, late finish date, free float, total float, and criticality, for each CPM Activity.

### 18.1.1.2 CPM Add-On Pre-Defined Variables

The CPM Add-On defines and maintains the following variables. They all access information about a CPM Activity:
18.1.1.2.1  CPMActivity.Critical

Returns TRUE if CPMActivity was critical in the current run of the CPM network. This value is not defined before the backward pass is complete and therefore can only be used for post-simulation calculations after each run.

18.1.1.2.2  CPMActivity.Duration

Returns the duration of CPMActivity in the current run of the CPM network. This value is not defined before CPMActivity has started.

18.1.1.2.3  CPMActivity.EarlyFinish

Returns the early finish of CPMActivity in the current run of the CPM network. This value is not defined before CPMActivity has started.

18.1.1.2.4  CPMActivity.EarlyStart

Returns the early start of CPMActivity in the current run of the CPM network. This value is not defined before CPMActivity has started.

18.1.1.2.5  CPMActivity.Finished

Returns TRUE if CPMActivity has already finished in the current run of the CPM network. This value is always defined.

18.1.1.2.6  CPMActivity.FreeFloat

Returns the free float for CPMActivity in the current run of the CPM network. This value is not defined before the backward pass is complete and therefore can only be used for post-simulation calculations after each run.

18.1.1.2.7  CPMActivity.GoingOn

Returns TRUE if CPMActivity is currently going on. This value is always defined.
18.1.1.2.8 CPMActivity.LateFinish

Returns the late finish for CPMActivity in the current run of the CPM network. This value is not defined before the backward pass is complete and therefore can only be used for post-simulation calculations after each run.

18.1.1.2.9 CPMActivity.LateStart

Returns the late start for CPMActivity in the current run of the CPM network. This value is not defined before the backward pass is complete and therefore can only be used for post-simulation calculations after each run.

18.1.1.2.10 CPMActivity.Started

Returns TRUE if CPMActivity has already started in the current run of the CPM network. This value is always defined.

18.1.1.2.11 CPMActivity.TotalFloat

Returns the total float for CPMActivity in the current run of the CPM network. This value is not defined before the backward pass is complete and therefore can only be used for post-simulation calculations after each run.

18.1.1.3 CPM Add-On Persistent Collectors

The CPM Add-On defines and maintains the following persistent statistics Collectors. These Collectors are standard Stroboscope Collectors. They expose their average, standard deviation, number of samples, minimum, and maximum values through pre-defined variables like any other Collector:

18.1.1.3.1 CR_CPMActivityName

This Collector samples a value of one every time CPMActivityName is critical and a value of zero otherwise. The average value of the samples indicates the percent of the runs in which CpmActivityName was critical.

18.1.1.3.2 Dur_CPMActivityName

This Collector holds statistics about the duration of CPMActivityName.
18.1.1.3.3 \textit{EF\textsubscript{CPMActivityName}}

This Collector holds statistics about the early finish of \textit{CPMActivityName}.

18.1.1.3.4 \textit{ES\textsubscript{CPMActivityName}}

This Collector holds statistics about the early start of \textit{CPMActivityName}.

18.1.1.3.5 \textit{FF\textsubscript{CPMActivityName}}

This Collector holds statistics about the free float of \textit{CPMActivityName}.

18.1.1.3.6 \textit{LF\textsubscript{CPMActivityName}}

This Collector holds statistics about the late finish of \textit{CPMActivityName}.

18.1.1.3.7 \textit{LS\textsubscript{CPMActivityName}}

This Collector holds statistics about the late start of \textit{CPMActivityName}.

18.1.1.3.8 \textit{ProjectDur}

This Collector holds statistics about the duration of the project.

18.1.1.3.9 \textit{TF\textsubscript{CPMActivityName}}

This Collector holds statistics about the total float of \textit{CPMActivityName}.

18.1.2 Example Using the CPM Add-On

The following model uses the CPM Add-On to analyze the fast food outlet construction described in section 16.4.1.3 and shown in Figure 44. Note that the only standard Stroboscope statement that is essential for this model is \texttt{LOADADDON}. Most of the model consists of statements defined by the Add-On.

\begin{verbatim}
VARIABLE nReplications 1000;
SEED 9111964;
STREAMS 15;
LOADADDON "e:\Strobos\Add-Ons\CpmAddOn.dll";

CPMACTIVITY BaseSlab sPert[3,6,12,1];
CPMACTIVITY WallPanels sPert[4,6,9,2];
CPMACTIVITY ParkingArea sPert[6,15,20,3];
CPMACTIVITY RoofTrusses sPert[1,2,5,4];
CPMACTIVITY Landscaping sPert[3,5,10,5];
CPMACTIVITY Roofing sPert[1,3,5,6];
\end{verbatim}
CPMACTIVITY Windows sPert[1,2,4,7];
CPMACTIVITY Doors sPert[1,2,4,8];
CPMACTIVITY Counter sPert[3,8,10,9];
CPMACTIVITY WIFridge sPert[2,5,8,10];
CPMACTIVITY Sign sPert[3,4,6,11];
CPMACTIVITY CounterEq sPert[1,2,4,12];
CPMACTIVITY KitchenEq sPert[4,10,15,13];
CPMACTIVITY FloorCvrngs sPert[2,4,8,14];
CPMACTIVITY Furnishings sPert[5,10,15,15];

PRECEDENCE BaseSlab WallPanels;
PRECEDENCE BaseSlab ParkingArea;
PRECEDENCE WallPanels RoofTrusses;
PRECEDENCE ParkingArea Landscaping;
PRECEDENCE RoofTrusses Roofing;
PRECEDENCE Roofing Windows;
PRECEDENCE Roofing Doors;
PRECEDENCE Windows Counter;
PRECEDENCE Doors Counter;
PRECEDENCE Windows WIFridge;
PRECEDENCE Doors WIFridge;
PRECEDENCE Landscaping Sign;
PRECEDENCE Counter CounterEq;
PRECEDENCE Windows KitchenEq;
PRECEDENCE Doors KitchenEq;
PRECEDENCE WIFridge FloorCvrngs;
PRECEDENCE Counter FloorCvrngs;
PRECEDENCE KitchenEq FloorCvrngs;
PRECEDENCE FloorCvrngs Furnishings;

CPMREPLICATE nReplications;

If the above model is submitted to Stroboscope for processing (and Stroboscope can find “CpmAddOn.dll”), the following output is produced:

Stroboscope Model CPM2.STR (1008348808)

Number of replications performed : 1000
Average Project Duration         : 44.86
90% CI on project duration       : [44.70,45.03]

<table>
<thead>
<tr>
<th>CPM Activity</th>
<th>Time</th>
<th>ESD</th>
<th>LSD</th>
<th>EFD</th>
<th>LFD</th>
<th>FF</th>
<th>TF</th>
<th>%Critic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Doors</td>
<td>2.18</td>
<td>18.08</td>
<td>18.36</td>
<td>20.25</td>
<td>20.54</td>
<td>0.28</td>
<td>0.28</td>
<td>50.60%</td>
</tr>
<tr>
<td>ParkingArea</td>
<td>14.40</td>
<td>6.61</td>
<td>20.82</td>
<td>21.01</td>
<td>35.22</td>
<td>-0.00</td>
<td>14.21</td>
<td>0.00%</td>
</tr>
<tr>
<td>Roofing</td>
<td>2.99</td>
<td>15.09</td>
<td>15.09</td>
<td>18.08</td>
<td>18.08</td>
<td>0.00</td>
<td>0.00</td>
<td>100.00%</td>
</tr>
<tr>
<td>FloorCvrngs</td>
<td>4.36</td>
<td>30.56</td>
<td>30.56</td>
<td>34.91</td>
<td>34.91</td>
<td>0.00</td>
<td>0.00</td>
<td>100.00%</td>
</tr>
<tr>
<td>Counter</td>
<td>7.53</td>
<td>20.54</td>
<td>23.03</td>
<td>28.06</td>
<td>30.56</td>
<td>0.00</td>
<td>2.49</td>
<td>14.00%</td>
</tr>
<tr>
<td>Windows</td>
<td>2.19</td>
<td>18.08</td>
<td>18.35</td>
<td>20.26</td>
<td>20.54</td>
<td>0.27</td>
<td>0.27</td>
<td>49.40%</td>
</tr>
<tr>
<td>RoofTrusses</td>
<td>2.31</td>
<td>12.77</td>
<td>12.77</td>
<td>15.09</td>
<td>15.09</td>
<td>0.00</td>
<td>0.00</td>
<td>100.00%</td>
</tr>
<tr>
<td>BaseSlab</td>
<td>6.61</td>
<td>0.00</td>
<td>0.00</td>
<td>6.61</td>
<td>6.61</td>
<td>0.00</td>
<td>0.00</td>
<td>100.00%</td>
</tr>
<tr>
<td>WIFridge</td>
<td>4.98</td>
<td>20.54</td>
<td>25.58</td>
<td>25.51</td>
<td>30.56</td>
<td>5.04</td>
<td>5.04</td>
<td>0.10%</td>
</tr>
<tr>
<td>Furnishings</td>
<td>9.95</td>
<td>34.91</td>
<td>34.91</td>
<td>44.86</td>
<td>44.86</td>
<td>0.00</td>
<td>0.00</td>
<td>100.00%</td>
</tr>
<tr>
<td>KitchenEq</td>
<td>9.88</td>
<td>20.54</td>
<td>20.67</td>
<td>30.42</td>
<td>30.56</td>
<td>0.14</td>
<td>0.14</td>
<td>85.90%</td>
</tr>
<tr>
<td>CounterEq</td>
<td>2.16</td>
<td>28.06</td>
<td>42.71</td>
<td>30.22</td>
<td>44.86</td>
<td>14.64</td>
<td>14.64</td>
<td>0.00%</td>
</tr>
<tr>
<td>Sign</td>
<td>4.15</td>
<td>26.50</td>
<td>40.71</td>
<td>30.65</td>
<td>44.86</td>
<td>14.21</td>
<td>14.21</td>
<td>0.00%</td>
</tr>
<tr>
<td>WallPanels</td>
<td>6.16</td>
<td>6.61</td>
<td>6.61</td>
<td>12.77</td>
<td>12.77</td>
<td>-0.00</td>
<td>0.00</td>
<td>100.00%</td>
</tr>
<tr>
<td>Landscaping</td>
<td>5.49</td>
<td>21.01</td>
<td>35.22</td>
<td>26.50</td>
<td>40.71</td>
<td>0.00</td>
<td>14.21</td>
<td>0.00%</td>
</tr>
</tbody>
</table>
18.1.3 Additional Methods of Accessing Statements and Functions Defined in Add-Ons

It is also possible to register specific statements and functions defined in an Add-On without loading the entire Add-On. This method could be used to register functions or statements that were not specifically designed for Stroboscope (e.g., functions from a standard mathematical library). It can also be used when a model needs to use two or more Add-Ons that expose statements or functions with the same names.

In order to load a function as a statement, the following conditions must be met:

- The function must be exported from the DLL and the name used by the DLL to export the function must be known.

- The function must use the standard calling convention (__stdcall).

- The function must return a 32-bit integer and take a pointer to a null terminated constant ANSI string as argument.

- The function must return the value 0 to indicate failure.

The STATEMENT statement is used to register a statement with Stroboscope:

Syntax: STATEMENT NameInStrobo DllName NameInDll;

Example: STATEMENT CD kernel32.dll SetCurrentDirectoryA;

_NameInStrobo_ is the statement keyword that will be used in the model to invoke the external statement. _DllName_ is the name of the Add-On DLL. _NameInDll_ is the name for the statement as exported by the DLL. The example registers the Windows function _SetCurrentDirectoryA_, defined in “kernel32.dll”, as a statement that can be accessed by a Stroboscope model using the _CD_ statement keyword. The following statement could subsequently be used in a model to change the current directory:

CD D:\temp; / switch to directory “\temp” in drive D
After the execution of the above statement, the current drive and directory will be “D:\temp”. If the operating system cannot switch to the directory, the function fails and Stroboscope issues an error.

In order to load a function as a statement, the following conditions must be met:

- The function must be exported from the DLL and the name used by the DLL to export the function must be known.
- The function must use the standard calling convention (__stdcall).
- The function must return a double precision (64-bit) floating point number and take from zero to five double precision floating point numbers as arguments.
- The number of arguments that the function takes must be known.

The FUNCTION statement is used to register a custom function with Stroboscope:

Syntax: FUNCTION NameInStrobo DllName NameInDll nArguments [CONSTANT];
Example: FUNCTION PresentValue Discount.dll PresentValue 3 CONSTANT;

*NameInStrobo* is the name of the function as it will be used in the Stroboscope model. *DllName* is the name of the Add-On DLL. *NameInDLL* is the name for the function as defined in the DLL. *nArguments* is the number of arguments that must be passed to the function. The optional keyword CONSTANT can be used to indicate that the function always returns the same value when passed the same arguments. CONSTANT should be used for most functions that are not stochastic. Functions that sample from probability distributions or that can return different values when fed with the same arguments should not use CONSTANT. The example registers the *PresentValue* function which is defined in the “Discount.dll” DLL. The example names the function with the same name used by the writer of the Add-On. The function takes three arguments and will return the same value if the same three arguments are passed to it. The writer of the Add-On may have documented the Add-On as follows:
PresentValue[FuturePayment, DiscountRate, NumPeriods]

Returns the present value of FuturePayment discounted at a rate of DiscountRate for NumPeriods periods.

The function can then be used as if it were a built-in function. The following example uses the function:

PRINT StdOutput
   "PV of $120 to 17.25 years from now, at 8.3%/yr is %.2f"
   PresentValue[120, 0.083, 17.25];

18.2 Developing Add-Ons for Stroboscope

Add-Ons for Stroboscope must be developed with a programming language capable of generating 32-bit Win32 DLLs. The discussion that follows assumes that the Add-On will be written in C or C++. Add-Ons can also be written in FORTRAN, Pascal, or any other language that can handle the calling conventions and data types described here (which may have different names in other languages).

The function prototypes and constant definitions for use by C and C++ Add-Ons are included in “Strobosc.h”. The “Strobosc.lib” import library can be used for linking.

The first of the following sub-sections provides a complete description of the Stroboscope Add-On Interface. The next subsection contains the complete source code for the CPM Add-On described earlier in this chapter.

18.2.1 Add-On Interface

When Stroboscope encounters a LOADADDON statement, it loads the dynamic link library specified as argument and searches for a function exported as “StroboAddOnInit”. If Stroboscope does not find StroboAddOnInit, it issues a compilation error indicating that the Add-On initialization function was not found.

StroboAddOnInit must follow the “__stdcall” calling convention, return a 32-bit integer, and take a NULL terminated constant ANSI string as argument. The Add-On can use any name for the StroboAddOnInit function, as long as it is exported as StroboAddOnInit. Thus, StroboAddOnInit can be defined as follows:
int __stdcall MyAddOnInitFunc(const char* szModelName)
{
    //Perform initialization tasks that can call back into Stroboscope
    //to Register statements, functions, or variables; or perform any
    //other task necessary to initialize the Add-On.

    //szModelName points to a buffer that contains the name of the model
    //being processed. (E.g., C:\Strobos\Models\HeavyEq\Scrapers.str"

    //The function should return 0 if the Add-On could not initialize
    //properly. Otherwise the Add-On should return non-zero.
    if (bFailed)
        return 0;
    return 1;
}

The function is exported as “StroboAddOnInit” by using the following entry in the
exports section of the module definition (*.def) file:

EXPORTS
    StroboAddOnInit=MyAddOnInitFunc @2

From within StroboAddOnInit, the Add-On can call any of the functions exported
by Stroboscope. The Add-On should use this opportunity to register the statements,
functions, and variables it exposes; as well as to perform any other initialization required
by the Add-On. Particularly, statements must be registered in the call to
StroboAddOnInit. Otherwise Stroboscope will not recognize them as statements as it
continues on its first pass through the model file.

When registering a statement, function, or variable, the Add-On supplies the
name of the statement, function or variable; the address of a function in the Add-On; and
other data. When Stroboscope executes a statement, calls a function, or requests the value
of a variable registered by the Add-On, it calls the function whose address was passed-in
with the registration.

18.2.1.1 Registering Statements

The statements exported by the Add-On should be coded as functions with the
following prototype:
int __stdcall Statement(const char* szStatementArguments);

szStatementArguments is a NULL terminated constant ANSI string that contains the arguments for the statement. The string does not include the keyword used to invoke the statement nor the ending semicolon. The function should return non-zero if the execution of the statement is successful and zero otherwise. If the function returns zero, Stroboscope issues a compilation error and stops processing the model.

Strobosc.h defines type FPExtStatement as a pointer to a statement function:

typedef int (__stdcall *FPExtStatement)(const char* szExpression);

Statements are registered by calling RegStatement, which is exported by Stroboscope and prototyped as follows:

void __stdcall RegStatement(char * szAlias, FPExtStatement pFunc);

szAlias is the name to be used in Stroboscope models to invoke the statement. pFunc is the address of the function that Stroboscope will call when it executes the statement.

An Add-On can register a statement with the same name as a built-in Stroboscope statement. The statement defined by the Add-On takes precedence when called by the model using the Add-On. The built-in statement takes precedence when called from within the Add-On.

Although RegStatement can be called any time, it only makes sense to do so from within StroboAddOnInit.

18.2.1.2 Registering Functions

The functions exported by the Add-On should be coded as functions with one of the following prototypes, depending on the number of arguments required by the function:

double __stdcall FunctionWithNoArgs();

double __stdcall FunctionWith1Arg(double dArg1);
double __stdcall FunctionWith2Args( double dArg1,
       double dArg2);

double __stdcall FunctionWith3Args( double dArg1,
       double dArg2,
       double dArg3);

double __stdcall FunctionWith4Args( double dArg1,
       double dArg2,
       double dArg3,
       double dArg4);

double __stdcall FunctionWith5Args( double dArg1,
       double dArg2,
       double dArg3,
       double dArg4,
       double dArg5);

Functions are registered by calling RegFunction, which is exported by
Stroboscope and prototyped as follows:

void __stdcall RegFunction( char* szAlias,
       void* pFunc,
       int nArguments,
       int bIsConstant);

szAlias is the name to be used in the Stroboscope model to invoke the function.
pFunc is the address of the function that Stroboscope will call when an expression in the
model references the function. nArguments is the number of arguments that the function
takes; it must be in the range 0 to 5. bIsConstant should be non-zero if the function
returns the same value when fed with the same arguments (functions that sample from
probability distributions, for example, are not constant and should set bIsConstant to
zero).

An Add-On cannot register a function with a name that matches a built-in
Stroboscope function or a function already registered by another Add-On. Functions can
be registered from within StroboAddOnInit or at any other time.

18.2.1.3 Registering Variables

The variables exported by the Add-On should be coded as functions with the
following prototype:
double __stdcall ExtVariable(unsigned long dwData1,
                        unsigned long dwData2);

\textit{dwData1} and \textit{dwData2} were values supplied by the Add-On to Stroboscope when the variable was registered. This allows a single Add-On function to serve several variables. The function should return the current value of the variable.

Strobosc.h defines type FPExtVariable as a pointer to a variable function:

\begin{verbatim}
typedef double (__stdcall *FPExtVariable)(unsigned long dwData1,
                                          unsigned long dwData2);
\end{verbatim}

Variables are registered by calling \texttt{RegVariable}, which is exported by Stroboscope and prototyped as follows:

\begin{verbatim}
int __stdcall RegVariable( const char * szAlias,
                   FPExtVariable pFunc,
                   unsigned long dwData1,
                   unsigned long dwData2,
                   int bIsConst);
\end{verbatim}

\textit{szAlias} is the name to be used in Stroboscope models to access the variable. \textit{pFunc} is the address of the function that Stroboscope will call to retrieve the value of the variable. \textit{dwData1} and \textit{dwData2} are values that Stroboscope will pass back to \textit{pFunc} when it needs to retrieve the value of a variable. \textit{bIsConst} should be non-zero when the variable has a constant value. If the value of the variable can change, \textit{bIsConst} should be zero.

An Add-On can register a variable with the same name as a built-in Stroboscope variable. The variable defined by the Add-On takes precedence and hides the built-in variable in expressions that are subsequently compiled. Previously compiled expressions continue to reference the built-in variable. An Add-On can register a variable at any time.

\textbf{18.2.1.4 Other Functions Exported by Stroboscope}

The functions listed below can be called from within the body of a custom statement or function. They are all prototyped in “Strobosc.h”.
18.2.1.4.1 EvaluateExpression

double __stdcall EvaluateExpression(const char* szExpression);

EvaluateExpression returns the result of evaluating *szExpression*. *szExpression* can reference built-in Stroboscope functions and variables, as well as functions and variables registered by Add-Ons.

18.2.1.4.2 EvaluateParsedExpression

double __stdcall EvaluateParsedExpression(long dwExpHandle);

EvaluateParsedExpression returns the current value of an expression previously compiled with the ParseExpression function (described below). *dwExpHandle* is the handle returned by a previous call to ParseExpression.

18.2.1.4.3 ExecuteStatement

int __stdcall ExecuteStatement(const char* szFullStatement);

ExecuteStatement executes the statement and arguments in *szFullStatement*. *szFullStatement* should not include the ending semicolon. The function returns zero if it fails to execute the statement. If an Add-On registers a statement with a name that matches a built-in Stroboscope statement, this function will execute the built-in Stroboscope statement (this is not the case when a statement is executed from a model).

18.2.1.4.4 ExtractArgument

#ifdef __cplusplus
char* __stdcall ExtractArgument(char*& szRemainingArguments);
#else
char* __stdcall ExtractArgument(char** const szRemainingArguments);
#endif

ExtractArgument parses statement arguments in the same manner as Stroboscope. ExtractArgument returns a pointer to the first meaningful character in *szRemainingArguments* and writes a NULL character immediately after the end of the first argument. ExtractArgument then changes *szRemainingArguments* so that it points to the first meaningful character in the next argument. If there are no more arguments,
ExtractArgument changes \textit{szRemainingArguments} so that it points to the NULL character marking the end of the original string.

18.2.1.4.5 \textit{ParseExpression}

\begin{verbatim}
long __stdcall ParseExpression(const char* szExpression);
\end{verbatim}

\textit{ParseExpression} compiles \textit{szExpression} but does not evaluate it. \textit{szExpression} can reference built-in Stroboscope functions and variables as well as those registered by Add-Ons. The return value is a handle that must be saved for use with \textit{EvaluateParsedExpression}, \textit{PublishParsedExpression}, or \textit{ReleaseParsedExpression}. When the handle is no longer needed, it should be freed with a call to \textit{ReleaseParsedExpression}.

18.2.1.4.6 \textit{PrintToStdError}

\begin{verbatim}
void __stdcall PrintToStdError(const char* szText);
\end{verbatim}

\textit{PrintToStdError} prints the text specified by \textit{szText} to the standard error device. By convention, carriage returns should be placed at the beginning of \textit{szText} rather than at the end.

18.2.1.4.7 \textit{PrintToStdOutput}

\begin{verbatim}
void __stdcall PrintToStdOutput(const char* szText);
\end{verbatim}

\textit{PrintToStdOutput} prints the text specified by \textit{szText} to the standard output device.

18.2.1.4.8 \textit{PrintToStdTrace}

\begin{verbatim}
void __stdcall PrintToStdTrace(const char* szText);
\end{verbatim}

\textit{PrintToStdTrace} prints the text specified by \textit{szText} to the standard trace device.

18.2.1.4.9 \textit{PublishParsedExpression}

\begin{verbatim}
int __stdcall PublishParsedExpression(const char* szVarName, const char* szDescription, unsigned long dwExpHandle);
\end{verbatim}

\textit{PublishParsedExpression} exposes as a variable an expression previously compiled with \textit{ParseExpression}. \textit{szVarName} is the variable name to be used in the model to
evaluate the parsed expression. \textit{szDescription} is a description of the variable. \\
\textit{dwExpHandle} is the handle to the parsed expression obtained by calling \texttt{ParseExpression}.

\texttt{PublishParsedExpression} returns non-zero if the expression is published successfully. If \texttt{PublishParsedExpression} fails, \textit{dwExpHandle} is no longer valid. For this reason, \textit{PublishParsedExpression} should be called soon after \texttt{ParseExpression} (in time to call \texttt{ParseExpression} again).

An expression that has been published should never be freed with a call to \texttt{ReleaseParsedExpression}.

\subsection{18.2.1.4.10 \texttt{ReleaseParsedExpression}}

\begin{verbatim}
void __stdcall ReleaseParsedExpression(unsigned long dwExpHandle);
\end{verbatim}

\texttt{ReleaseParsedExpression} frees the resources associated with an expression previously compiled with \texttt{ParseExpression}. \textit{dwExpHandle} is the handle returned by a previous call to \texttt{ParseExpression}.

An expression that has been published with a call to \texttt{PublishParsedExpression} should never be freed with a call to \texttt{ReleaseParsedExpression}.

\subsection{18.2.1.4.11 \texttt{StrbMathError}}

\begin{verbatim}
void __stdcall StrbMathError(char* szFunction, int nErrorType);
\end{verbatim}

The \texttt{StrbMathError} function can be called from within a function to indicate an error. \textit{szFunction} is the name of the function in which the error occurred. A call to \texttt{StrbMathError} does not return control to the caller. Control is transferred directly to the Stroboscope engine, which issues a runtime error and displays a message that depends on the value of \textit{nErrorType}. The possible values for \textit{nErrorType} are defined in Table 15 below.
Table 15 - StrbMathError Error Constants

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR_UNDEFINED</td>
<td>The value returned by the function is currently undefined.</td>
</tr>
<tr>
<td>STR_DOMAIN</td>
<td>Argument to function is outside domain of function.</td>
</tr>
<tr>
<td>STR_OVERFLOW</td>
<td>Result is too large to be represented.</td>
</tr>
<tr>
<td>STR_PLOSS</td>
<td>Partial loss of significance occurred.</td>
</tr>
<tr>
<td>STR_SING</td>
<td>Argument singularity: argument to function has illegal value. (For example, value 0 is passed to function that requires non-zero value.)</td>
</tr>
<tr>
<td>STR_TLOSS</td>
<td>Total loss of significance occurred.</td>
</tr>
<tr>
<td>STR_UNDERFLOW</td>
<td>Result is too small to be represented.</td>
</tr>
</tbody>
</table>

18.2.2 CPM Add-On for Stroboscope Source Code

The following is the source code for the CPM Add-On described earlier. It is written in C++ using the string and template collection classes of the Microsoft Foundation Classes version 3.2. This code is included to illustrate how to use the Stroboscope Add-On Interface to develop an Add-On. Although usable as is, the Add-On presented here is merely an example that could certainly be improved upon. The source files have been compiled with Microsoft Visual C++ version 2.2.

The Add-On is implemented in two files: CpmAddOn.def and CpmAddOn.cpp. CpmAddOn.def is very short:

```
LIBRARY CPMADDON

CODE      PRELOAD MOVEABLE DISCARDABLE
DATA      PRELOAD SINGLE

EXPORTS
  StroboAddOnInit @2
```

CpmAddOn.cpp is substantially larger. The comments should make the source code self-explanatory:
// Stroboscope CPM Sample Add-On
// Copyright (C) 1995 Julio C. Martinez,
// All rights reserved.

// This source code is only intended as a supplement to the
// Stroboscope Add-On Interface Reference.
// See the reference for detailed information.

#include <afxwin.h>
#include <afxcoll.h>
#include <afxtempl.h>
#include <math.h>
#include "strobosc.h"

//working buffers for string manipulation
char szScratch[MAX_STATEMENT_LENGTH];
char szScratch2[MAX_STATEMENT_LENGTH];

//function prototypes for the statements registered by the Add-On
int __stdcall CpmActivity(const char* szArguments);
int __stdcall CpmPrecedence(const char* szArguments);
int __stdcall CpmCompile(const char* szArguments);
int __stdcall DoCpm(const char* szArguments);
int __stdcall CpmReplicate(const char* szArguments);
int __stdcall CpmReport(const char* szArguments);

//function prototypes for the functions registered by the Add-On
double __stdcall BackPass();
double __stdcall NewRepl();

//global variables
BOOL glb_bCompiled=FALSE;
BOOL glb_bProjectEnded=FALSE;
double glb_dProjectDuration;
const unsigned long glb_dwSimTimeHandle=ParseExpression("SimTime");

//The Add-On initialization function - must be exported in DEF file
int __stdcall StroboAddOnInit(const char* szModelName)
{
    //show on the error device that Stroboscope has loaded this Add-on
    PrintToStdError("\nProbabilistic CPM Add-On for Stroboscope");

    //if "SimTime" was not parsed correctly there's not much to do
    if (!glb_dwSimTimeHandle)
        return FALSE;

    //Register with Stroboscope the statements exposed by the add-on
    RegStatement("CPMACTIVITY",CpmActivity);
    RegStatement("PRECEDENCE",CpmPrecedence);
    RegStatement("CPMCOMPILE",CpmCompile);
    RegStatement("DOCPM",DoCpm);
    RegStatement("CPMREPORT",CpmReport);
    RegStatement("CPMREPLICATE",CpmReplicate);

    //Register with Stroboscope these functions, they are not meant
    //for use by the users of the add-on. Therefore we give them
    //Stroboscope names that are hard to guess
    RegFunction("Prv_BackPass",BackPass,0, FALSE);
    RegFunction("Prv_NewRep",NewRepl,0, FALSE);
//Define a dummy SaveValue with the name of the add-on. This will
//serve as an action target and allow us to force the evaluation of
//certain functions when certain events happen
ExecuteStatement("SAVEVALUE CpmAddOn 0");

//The only resource type in this network, basically a signal
ExecuteStatement("GENTYPE Time");

//Queue and Combi combination to represent the Start Activity
ExecuteStatement("QUEUE Prv_StartQ Time");
ExecuteStatement("COMBI Prv_StartCmb");
ExecuteStatement("LINK Prv_StartLink Prv_StartQ Prv_StartCmb");

//Define an action that calls our NewRepl function and sets up the
//add-on for a new replication.
ExecuteStatement("ONENTRY Prv_StartQ ASSIGN CpmAddOn Prv_NewRep[]");

// Consolidator for End Activity, it finishes when all the
// Activities with no successors have finished.
ExecuteStatement("CONSOLIDATOR Prv_End");

//Define an action that triggers the backward pass calculations by
//forcing the evaluation of our BackPass function. This is the first
//action to be performed when Prv_End finishes.
ExecuteStatement("ONEND Prv_End ASSIGN CpmAddOn Prv_BackPass[]");

//A persistent Collector for the duration of the project. It
//samples every time Consolidator "Prv_End" finishes.
ExecuteStatement("COLLECTOR ProjectDur*");
ExecuteStatement("ONEND Prv_End COLLECT ProjectDur SimTime");

return TRUE;
}//end StroboAddOnInit

//function to retrieve the value of SimTime from Stroboscope
double SimTime(){return EvaluateParsedExpression(glb_dwSimTimeHandle);}

//class that represents a CPM Activity
class CCPMActivity
{
friend double __stdcall BackPass();
friend double __stdcall NewRepl();

private:
    CString m_szName;
    BOOL m_bStarted;
    double m_dEarlyStart;
    unsigned long m_dwDurExpHandle;
    double m_dDuration;
    double m_dLateStart;
    double m_dFreeFloat;
    int m_nBackPassOrdersReceived;
    BOOL m_bWellInitialized; //set to false if constructor fails

    CTypedPtrArray<CPtrArray,CCPMActivity*> m_thePredecessors;
    CTypedPtrArray<CPtrArray,CCPMActivity*> m_theSuccessors;
void BackwardPass();
void NewPrepStarts();

// function that registers methods as pre-defined variables
BOOL ExposeMethod(const char* szSelector, double(CCPMActivity::*)(})();

public:
// constructor does big job
CCPMActivity(const char* szName, const char* szDuration);

// must release handle to unpublished expressions
~CCPMActivity(){ReleaseParsedExpression(m_dwDurExpHandle);}

// member functions that will be exposed as pre-defined variables
double DetermineDuration();
double GetEarlyStart();
double GetDuration();
double GetEarlyFinish();
double GetFreeFloat();
double GetTotalFloat();
double GetLateStart();
double GetLateFinish();
double bIsCritical();
double bStarted(){return m_bStarted;}
double bFinished()
    {return m_bStarted && SimTime()>=m_dEarlyStart+m_dDuration;}
double bGoingOn()
    {return m_bStarted && SimTime()<m_dEarlyStart+m_dDuration;}

int SuccessorCount(){return m_theSuccessors.GetSize();}
int PredecessorCount(){return m_thePredecessors.GetSize();}

void NewPredecessor(CCPMActivity* pPredecessor)
{    m_thePredecessors.Add(pPredecessor);
}

void NewSuccessor(CCPMActivity* pSuccessor)
{    m_theSuccessors.Add(pSuccessor);
}

BOOL Initialized(){return m_bWellInitialized;}

// map to obtain any CPM Activity given its name
CTypedPtrMap<CMapStringToPtr, CString, CCPMActivity*>* theActivityMap;

// array of CPM Activities that have no successors
CTypedPtrArray<CPtrArray, CCPMActivity*>* theEndingActivities;

typedef double(CCPMActivity::*FPCpmMemberFunction)();

// helper class to ease the conversion of "FPCpmMemberFunctions" back and
// forth to "unsigned longs" by using a cast. For example if dw is an
// unsigned long and a function expects an FPCpmMemberFunction, we can
// pass "(CCpmActMemFunc)dw". On the other hand, if pMemFunc is a
// FPCpmMemberFunction and a function expects an unsigned long, we
// can pass "(CCpmActMemFunc)pMemFunc".
class CCpmActMemFunc
{
private:
    union
        { FPCpmMemberFunction AsMemFunc;
          unsigned long AsDword;
        };

public:
    CCpmActMemFunc(const FPCpmMemberFunction& pMemFunc)
        { AsMemFunc=pMemFunc; }
    CCpmActMemFunc(const unsigned long dw)
        { AsDword=dw; }
    operator FPCpmMemberFunction() const { return AsMemFunc; }
    operator unsigned long() const { return AsDword; }
};

//the actual function that is exported to Stroboscope to provide access
//to the methods in CCPMActivity.
double __stdcall InvokeMethod(unsigned long dw1, unsigned long dw2)
{
    //the CPMActivity is disguised as the first info Dword
    CCPMActivity* pCpmActivity=(CCPMActivity*)dw1;
    //the particular method is disguised as the second info Dword
    FPCpmMemberFunction pMemFunc=(CCpmActMemFunc)dw2;
    //simple function call based on the two pointers
    return (pCpmActivity->*pMemFunc)();
}

//the helper function that exposes the methods in CCPMActivity by
//registering them as variables. The function used to determine the
//value of the variables is always "InvokeMethod". The additional data
//passed with the registration of the variable indicate the CPM Activity
//and particular method.
BOOL CCPMActivity::ExposeMethod(const char* szSelector,
    FPCpmMemberFunction pMemFunc)
{
    //build the name of the predefined variable by appending szSelector
    //to the name of the CPM Activity
    char szPredefVarName[64];
    sprintf(szPredefVarName, "%s.%s", (const char*)m_szName,szSelector);
    //register the variable with Stroboscope
    return RegVariable( szPredefVarName,
        InvokeMethod,
        (unsigned long)(void*)this,
        (CCpmActMemFunc)pMemFunc,
        FALSE);
}

CCPMActivity::CCPMActivity(const char* szName, const char* szDuration)
    : m_szName(szName), m_bWellInitialized(FALSE)
{
    char szScratch[MAX_STATEMENT_LENGTH];
theActivityMap.SetAt(szName,this);

// Each CPM Activity X is represented by a Consolidator In_X
// linked to a Normal X by a link IL_X. The release amount from
// the Consolidator is forced to be one (the consolidator will
// receive one from each predecessor).
sprintf(szScratch,"CONSOLIDATOR In_%s",szName);
if (!ExecuteStatement(szScratch)) return;

sprintf(szScratch,"NORMAL %s",szName);
if (!ExecuteStatement(szScratch)) return;

sprintf(szScratch,"LINK IL_%s In_%s %s Time",szName,szName,szName);
if (!ExecuteStatement(szScratch)) return;

sprintf(szScratch,"RELEASEAMT IL_%s 1", szName);
if (!ExecuteStatement(szScratch)) return;

// Parse and save the expression for the duration of the CPM Activity
m_dwDurExpHandle=ParseExpression(szDuration);
if (!m_dwDurExpHandle) return;

// Expose the expression for the duration of the CPM Activity as a
// variable. This variable is not for use outside of the Add-On and
// is therefore prefixed with something not likely to be guessed. See
// function DetermineDuration, it does a little more than evaluate
// the duration expression.
if (!ExposéMethod("Prv_Duration", DetermineDuration)) return;

// Set the Duration of the Normal that is encapsulated by this
// CPM Activity to the variable exposed above.
sprintf(szScratch,"DURATION %s %s.Priv_Duration",szName,szName);
if (!ExecuteStatement(szScratch)) return;

// Expose the different methods which are for use by models based
// on this Add-On. Method GetDuration is exposed as variables
// of form CPMActivity.Duration, these variables hide the standard
// Activity.Duration (which is originally an instance variable).
if (!ExposéMethod("Started",bStarted)) return;
if (!ExposéMethod("EarlyStart",GetEarlyStart)) return;
if (!ExposéMethod("Duration",GetDuration)) return;
if (!ExposéMethod("GoingOn",bGoingOn)) return;
if (!ExposéMethod("EarlyFinish",GetEarlyFinish)) return;
if (!ExposéMethod("Finished",bFinished)) return;
if (!ExposéMethod("FreeFloat",GetFreeFloat)) return;
if (!ExposéMethod("TotalFloat",GetTotalFloat)) return;
if (!ExposéMethod("LateStart",GetLateStart)) return;
if (!ExposéMethod("LateFinish",GetLateFinish)) return;
if (!ExposéMethod("Critical",bIsCritical)) return;

// Persistent Collectors to keep the statistics of interest
// for each of the CPM Activities
sprintf(szScratch,"COLLECTOR ES_%s\",szName);
if (!ExecuteStatement(szScratch)) return;

sprintf(szScratch,"COLLECTOR LS_%s\",szName);
if (!ExecuteStatement(szScratch)) return;

sprintf(szScratch,"COLLECTOR EF_%s\",szName);
if (!ExecuteStatement(szScratch)) return;
sprintf(szScratch,"COLLECTOR LF_%s",szName);
if (!ExecuteStatement(szScratch)) return;

sprintf(szScratch,"COLLECTOR Dur_%s",szName);
if (!ExecuteStatement(szScratch)) return;

sprintf(szScratch,"COLLECTOR CR_%s",szName);
if (!ExecuteStatement(szScratch)) return;

sprintf(szScratch,"COLLECTOR TF_%s",szName);
if (!ExecuteStatement(szScratch)) return;

sprintf(szScratch,"COLLECTOR FF_%s",szName);
if (!ExecuteStatement(szScratch)) return;

//Collect the statistics related to this CPM Activity by defining
//actions that are executed when the backward pass is done. (The
//very first action attached to "ONEND Prv_End" does the backward
//pass).

sprintf(szScratch,"ONEND Prv_End COLLECT ES_%s %s.EarlyStart",
    szName,szName);
if (!ExecuteStatement(szScratch)) return;

sprintf(szScratch,"ONEND Prv_End COLLECT LS_%s %s.LateStart",
    szName,szName);
if (!ExecuteStatement(szScratch)) return;

sprintf(szScratch,"ONEND Prv_End COLLECT EF_%s %s.EarlyFinish",
    szName,szName);
if (!ExecuteStatement(szScratch)) return;

sprintf(szScratch,"ONEND Prv_End COLLECT LF_%s %s.LateFinish",
    szName,szName);
if (!ExecuteStatement(szScratch)) return;

sprintf(szScratch,"ONEND Prv_End COLLECT Dur_%s %s.Duration",
    szName,szName);
if (!ExecuteStatement(szScratch)) return;

sprintf(szScratch,"ONEND Prv_End COLLECT CR_%s %s.Critical",
    szName,szName);
if (!ExecuteStatement(szScratch)) return;

sprintf(szScratch,"ONEND Prv_End COLLECT TF_%s %s.TotalFloat",
    szName,szName);
if (!ExecuteStatement(szScratch)) return;

sprintf(szScratch,"ONEND Prv_End COLLECT FF_%s %s.FreeFloat",
    szName,szName);
if (!ExecuteStatement(szScratch)) return;

//Indicate that all went well
m_bWellInitialized=TRUE;

}//end CCPMAcCtivity::CCPMActivity
This is the method exposed as "CPMActivityName.Prv_Duration", and which is used to determine the duration of the Normal encapsulated by the CPM Activity. Before evaluating the expression supplied by the model, it uses the opportunity to record that the CPM Activity started and its early start date and duration.

```cpp
double CCPMActivity::DetermineDuration()
{
    m_bStarted=TRUE;
    m_dEarlyStart=SimTime();
    m_dDuration=EvaluateParsedExpression(m_dwDurExpHandle);
    if (m_dDuration<0)
        m_dDuration=0;
    return m_dDuration;
}
```

Method exposed as "CPMActivityName.EarlyStart"

```cpp
double CCPMActivity::GetEarlyStart()
{
    if(!m_bStarted)
    {
        PrintToStdError("\nAccessing early start of a CPM Activity "
                        "that has not started");
        StrbMathError(m_szName+.EarlyStart",STR_UNDEFINED);
    }
    return m_dEarlyStart;
}
```

Method exposed as "CPMActivityName.Duration"

```cpp
double CCPMActivity::GetDuration()
{
    if(!m_bStarted)
    {
        PrintToStdError("\nAccessing the duration of a CPM Activity "
                        "that has not started");
        StrbMathError(m_szName+.Duration",STR_UNDEFINED);
    }
    return m_dDuration;
}
```

Method exposed as "CPMActivityName.EarlyFinish"

```cpp
double CCPMActivity::GetEarlyFinish()
{
    if(!m_bStarted)
    {
        PrintToStdError("\nAccessing the early finish of a CPM "
                        "Activity that has not started");
        StrbMathError(m_szName+.EarlyFinish",STR_UNDEFINED);
    }
    return m_dEarlyStart+m_dDuration;
}
```

Method exposed as "CPMActivityName.FreeFloat"

```cpp
double CCPMActivity::GetFreeFloat()
{
    if(!glb_bProjectEnded)
    {
        PrintToStdError("\nAccessing the free float of a CPM Activity "
                        "before the forward pass is complete");
        StrbMathError(m_szName+.FreeFloat",STR_UNDEFINED);
    }
    return m_dFreeFloat;
}
//method exposed as "CPMActivityName.TotalFloat"
double CCPMActivity::GetTotalFloat()
{
    if(!glb_bProjectEnded)
    {
        PrintToStdError("\nAccessing total float of a CPM Activity "
                        "before the forward pass is complete");
        StrbMathError(m_szName+.TotalFloat",STR_UNDEFINED);
    }

    return m_dLateStart-m_dEarlyStart;
}

//method exposed as "CPMActivityName.LateStart"
double CCPMActivity::GetLateStart()
{
    if(!glb_bProjectEnded)
    {
        PrintToStdError("\nAccessing the late start of a CPM Activity "
                        "before the forward pass is complete");
        StrbMathError(m_szName+.LateStart",STR_UNDEFINED);
    }

    return m_dLateStart;
}

//method exposed as "CPMActivityName.LateFinish"
double CCPMActivity::GetLateFinish()
{
    if(!glb_bProjectEnded)
    {
        PrintToStdError("\nAccessing late finish of a CPM Activity "
                        "before the forward pass is complete");
        StrbMathError(m_szName+.LateFinish",STR_UNDEFINED);
    }

    return m_dLateStart+m_dDuration;
}

//method exposed as "CPMActivityName.Critical"
double CCPMActivity::bIsCritical()
{
    if(!glb_bProjectEnded)
    {
        PrintToStdError("\nAccessing criticality of a CPM Activity "
                        "before the forward pass is complete");
        StrbMathError(m_szName+.Critical",STR_UNDEFINED);
    }

    return fabs(m_dLateStart-m_dEarlyStart)<1e-5;
}

//performs the backward pass calculations for this CPM Activity
void CCPMActivity::BackwardPass()
{
    //react only after all successors complete their backward pass calc
    if(++m_nBackPassOrdersReceived<SuccessorCount())
        return;

    //start out with the highest values for late start and free float
    m_dLateStart=glb_dProjectDuration-m_dDuration;
    m_dFreeFloat=glb_dProjectDuration;

    //ending activities determine their free float differently
    if (!SuccessorCount())
        m_dFreeFloat=min(m_dFreeFloat,
                        glb_dProjectDuration-(m_dEarlyStart+m_dDuration));
for (int i=0; i<SuccessorCount(); i++)
{  
m_dLateStart=min(m_dLateStart,
        m_theSuccessors[i]->m_dLateStart-m_dDuration);

    m_dFreeFloat=min(m_dFreeFloat,
        m_theSuccessors[i]->m_dEarlyStart-m_dEarlyStart-m_dDuration);
}

//signal predecessors to do their backward pass calculations
for (i=0;i<PredecessorCount(); i++)
    m_thePredecessors[i]->BackwardPass();

//sets up data in the Cpm Activity for a new replication
void CCPMActivity::NewReplStarts()
{
    m_bStarted=FALSE;
    m_nBackPassOrdersReceived=0;
}

//Called on first action attached to Consolidator Prv_End's OnEnd event.
//It begins the backward pass.
double __stdcall BackPass()
{
    //update global variables
    glb_dProjectDuration=SimTime();
    glb_bProjectEnded=TRUE;

    //instruct all CPM Activities with no successors to perform their
    //backward pass calculations
    for (int i=0; i<theEndingActivities.GetSize(); i++)
        theEndingActivities[i]->BackwardPass();

    return 0;
}

//sets up the model for a new replication
double __stdcall NewRepl()
{
    //update global variables
    glb_bProjectEnded=FALSE;

    //iterate through all the CPM Activities and set them up for a
    //new replication
    CString szActivity;
    CCPMActivity* pCpmActivity;
    POSITION pos=theActivityMap.GetStartPosition();
    while (pos)
    {  
        theActivityMap.GetNextAssoc(pos,szActivity,pCpmActivity);
        pCpmActivity->NewReplStarts();
    }

    return 0;
}
//the function registered with Stroboscope as CPMACTIVITY
int __stdcall CpmActivity(const char* szArguments)
{
    //cannot define CPM Activities after network has been "compiled"
    if (glb_bCompiled)
    {
        PrintToStdError("\nCannot define CPM Activity after 
        "the network has been compiled");
        return FALSE;
    }

    strncpy(szScratch,szArguments,MAX_STATEMENT_LENGTH);
    char* szWork=szScratch;
    if (!strlen(szWork))
    {
        PrintToStdError("\nMissing CPM Activity Name");
        return FALSE;
    }

    char* szActivityName=ExtractArgument(szWork);
    if (!strlen(szWork))
    {
        PrintToStdError("\nMissing duration expression");
        return FALSE;
    }

    char* szDuration=ExtractArgument(szWork);
    if (strlen(szWork))
    {
        PrintToStdError("\nToo many arguments");
        return FALSE;
    }

    //the CCPMACTIVITY constructor does the real job
    CCPMACTIVITY* pTheActivity=
        new CCPMACTIVITY(szActivityName,szDuration);

    //if somehow the CCPMACTIVITY constructor failed, return FALSE
    return pTheActivity->Initialized();
}

//the function registered with Stroboscope as CPMPRECEDENCE
int __stdcall CpmPrecedence(const char* szArguments)
{
    //cannot define precedences after network has been "compiled"
    if (glb_bCompiled)
    {
        PrintToStdError("\nCannot define CPM Precedence relationship 
        "after the network has been compiled");
        return FALSE;
    }

    strncpy(szScratch,szArguments,MAX_STATEMENT_LENGTH);
    char* szWork=szScratch;
    if (!strlen(szWork))
    {
        PrintToStdError("\nMissing Predecessor");
        return FALSE;
    }

    CString szPredecessor=ExtractArgument(szWork);
    CCPMACTIVITY* pPredecessor;
    if (!theActivityMap.Lookup(szPredecessor,pPredecessor))
    {
        PrintToStdError("\nPredecessor not defined");
        return FALSE;
    }
if (!strlen(szWork))
{ PrintToStdError("\nMissing Successor");
  return FALSE;
}

CString szSuccessor=ExtractArgument(szWork);
CCPMActivity* pSuccessor;
if (!theActivityMap.Lookup(szSuccessor,pSuccessor))
{ PrintToStdError("\nSuccessor not defined");
  return FALSE;
}

if (strlen(szWork))
{ PrintToStdError("\nToo many arguments");
  return FALSE;
}

//tell each of the CPM Activities that they're involved in a new
//precedence relationship
pPredecessor->NewSuccessor(pSuccessor);
pSuccessor->NewPredecessor(pPredecessor);

//define the link that connects the predecessor CPM Activity to the
//successor CPM Activity.
sprintf(szScratch2,"LINK FS_%s_%s %s In_%s Time",
   (const char*) szPredecessor,(const char*) szSuccessor,
   (const char*) szPredecessor,(const char*) szSuccessor);
if (!ExecuteStatement(szScratch2)) return FALSE;
return TRUE;
}//end CpmPrecedence

//the function registered with Stroboscope as CPMCOMPILE
int __stdcall CpmCompile(const char* szArguments)
{
  //no need to compile if already done
  if (glb_bCompiled) return TRUE;

  if (strlen(szArguments))
  { PrintToStdError("\nCPMCOMPILE takes no arguments");
    return FALSE;
  }

  //iterate through all CPM Activities and do things that depend on
  //the number of predecessors and successors
  POSITION pos=theActivityMap.GetStartPosition();
  CString szActivity;
  CCPMActivity* pCpmActivity;
  while (pos)
  {
    theActivityMap.GetNextAssoc(pos,szActivity,pCpmActivity);

    //On the forward pass, CPM Activity X starts when
    //Consolidator In_X ends. In_X ends when it has
    //received one resource from each predecessor Activity
    //(i.e., all predecessors have finished).
    sprintf(szScratch2,
      "CONSOLIDATEWHEN In_%s In_%s.Time.Count>=%i",
      (const char*)szActivity,(const char*)szActivity,
      max(1,pCpmActivity->PredecessorCount()));
    if (!ExecuteStatement(szScratch2)) return FALSE;
//special treatment for beginning CPM Activities
if(!pCpmActivity->PredecessorCount())
{
    // CPM Activities that have no predecessors must be linked
    // from the starting Activity, Prv_StartCmb.
    sprintf(szScratch2,"LINK FS_%s Prv_StartCmb In_%s Time",
            (const char*)szActivity,(const char*)szActivity);
    if (!ExecuteStatement(szScratch2)) return FALSE;
}

//special treatment for ending CPM Activities
if(!pCpmActivity->SuccessorCount())
{
    //keep track of these guys
    theEndingActivities.Add(pCpmActivity);

    //Define a link that connects to the End Activity
    sprintf(szScratch2,"LINK FS_%s_End %s Prv_End Time",
            (const char*)szActivity,(const char*)szActivity);
    if (!ExecuteStatement(szScratch2)) return FALSE;
}

//Allow the Prv_End Consolidator to Consolidate when all the ending
//CPM Activities have finished.
sprintf(szScratch2,"CONSOLIDATEWHEN Prv_End Prv_End.Time.Count>=%i",
        theEndingActivities.GetSize());
if (!ExecuteStatement(szScratch2)) return FALSE;

//Model has been succesfully compiled
return glb_bCompiled=TRUE;

//the function registered with Stroboscope as DOCPM
int __stdcall DoCpm(const char* szArguments)
{
    if (strlen(szArguments))
    {
        PrintToStdError("\nDOCPM takes no arguments");
        return FALSE;
    }

    //if the model has not been compiled, compile it.
    if (!glb_bCompiled) CpmCompile("\n");

    //Simply initialize and simulate.
    if (!ExecuteStatement("INIT Prv_StartQ 1")) return FALSE;
    if (!ExecuteStatement("SIMULATE")) return FALSE;

    return TRUE;
}

//the function registered with Stroboscope as CPMREPLICATE
int __stdcall CpmReplicate(const char* szArguments)
{
    strncpy(szScratch,szArguments,MAX_STATEMENT_LENGTH);
    char* szWork=szScratch;
if (!strlen(szWork))
{ PrintToStdError("\nMissing number of replications to do");
 return FALSE;
}

CString szReplCount=ExtractArgument(szWork);
int nReplications=EvaluateExpression(szReplCount);

if (strlen(szWork))
{ PrintToStdError("\nToo many arguments");
 return FALSE;
}

//Cpms are very short and flushing run info after every run wastes
//too many much time
if (!ExecuteStatement("SILENTREPLICATE")) return FALSE;

//replicate the indicated number of times
for (;nReplications; nReplications--)
{ if (!ExecuteStatement("CLEAR")) return FALSE;
 if (!DoCpm("")) return FALSE;
}

//print a report showing the results
return CpmReport("");
}//end CpmReplicate

//the function registered with Stroboscope as CPMREPORT
int __stdcall CpmReport(const char* szArguments)
{
 int nReplications=EvaluateExpression("ProjectDur.nSamples");

 if (!nReplications)
 { PrintToStdOutput("No replications run, nothing to report");
  return TRUE;
 }

 sprintf(szScratch2,"Number of replications performed : %i\n",
 nReplications);
 PrintToStdOutput(szScratch2);

 sprintf(szScratch2,"Average Project Duration         : %.2f\n",
 EvaluateExpression("ProjectDur.AveVal"));
 PrintToStdOutput(szScratch2);

 //if sufficient replications print Confidence Interval Info
 if (nReplications>=2)
 { sprintf(szScratch2,
 "90%% CI on project duration       : [%2f,%.2f]\n",
 EvaluateExpression("ProjectDur.AveVal-"
 "Confidence[ProjectDur.SDVal,0.90,"nSamples"]"),
 EvaluateExpression("ProjectDur.AveVal+
 "Confidence[ProjectDur.SDVal,0.90,"nSamples"]"));
 PrintToStdOutput(szScratch2);
 }

 PrintToStdOutput("\n\n");
//header of output table
PrintToStdOutput("CPM Activity Time ESD LSD "
"EFD LFD FF TF %Critic\n");
PrintToStdOutput("======================================="
"================================\n");

//iterate through all CPM Activities and print a row for each
POSITION pos=theActivityMap.GetStartPosition();
CString szActivity;
CCPMActivity* pCpmActivity;
while (pos)
{
    theActivityMap.GetNextAssoc(pos,szActivity,pCpmActivity);
    CString szVariableEnding = szActivity+.AveVal;
    sprintf(szScratch2,
            "%-14s %6.2f %6.2f %6.2f %6.2f %6.2f %6.2f %6.2f %6.2f%\n",
            (const char*)szActivity,
            EvaluateExpression(CString("Dur_")+szVariableEnding),
            EvaluateExpression(CString("ES_")+szVariableEnding),
            EvaluateExpression(CString("LS_")+szVariableEnding),
            EvaluateExpression(CString("EF_")+szVariableEnding),
            EvaluateExpression(CString("LF_")+szVariableEnding),
            EvaluateExpression(CString("FF_")+szVariableEnding),
            EvaluateExpression(CString("TF_")+szVariableEnding),
            100.0*EvaluateExpression(CString("CR_")+szVariableEnding));
    PrintToStdOutput(szScratch2);
}
return TRUE;

18.3 Recap

Add-Ons can be used to aid in the development of very complex models or as general extensions to the Stroboscope language. An Add-On can define additional statements such as COMBI, functions such as Pertpg, and Add-On (system) maintained variables such as Collector.AveVal.

A tool capable of generating Win32 DLL’s and the “Strobosc.h” and “Strobosc.lib” files (both of which are provided with Stroboscope) are required for the development of an Add-On. This chapter explained how to develop an Add-On and presented a complete example that can be used to analyze probabilistic CPM networks.
Chapter 19
Conclusion

Academics and practitioners have recognized the need to use computer simulation to plan and analyze construction processes for over 25 years. As a consequence, there has been substantial research activity related to the simulation of construction processes and projects.

Processes in most industries can be easily represented from the point of view of moving entities that compete for static servers. For this reason, most commercial general purpose simulation systems adopt the Process Interaction (PI) and Event Scheduling (ES) strategies. Construction processes are different from those encountered in other disciplines. They tend to be complex dynamic processes that require the collaboration of many distinct resources to perform activity-oriented tasks. The Activity Scanning (AS) strategy is thus the most appropriate for the modeling of construction processes. Due to the unavailability of suitable AS simulation tools, researchers have focused on developing and enhancing AS simulation systems specific to construction.

The oldest and most widely used construction process simulation system is CYCLONE (Halpin 1976). Over the years, many researchers have enhanced CYCLONE’s capabilities in different ways. These enhancements include extensions that enable CYCLONE to handle learning curves; keep track of cost; and model transient effects.

Other researchers have developed entirely new systems that build upon the basic CYCLONE concepts, notably RESQUE (Chang 1986), COOPS (Liu 1991), and CIPROS
(Odeh 1992). In addition, other systems have been developed for the modeling of construction at the project level.

In order to model construction processes realistically, a simulation tool must make dynamic use of the state of the simulation and the properties of resources in numerous ways that cannot be foreseen at the time the simulation system was designed. In order to achieve this, the simulation system must be programmable. This is the difference between a simulation language and a simulator. Previously existing simulation tools for construction are simulators and do not meet these requirements.

Stroboscope represents a radical departure from other construction simulation systems. It is a **programming language** that represents resources as objects that have assignable, persistent, and dynamic properties; and that can actively and dynamically take into consideration the state of the simulated process.

Another major difference between Stroboscope and other construction simulation tools is its open design. The input and output to a Stroboscope model are determined and designed by the creator of the model at two levels. The first level is using Stroboscope’s built-in programmability. The second level is by extending Stroboscope through Dynamic Link Libraries created with high level compiled languages such as C, C++, Pascal, and FORTRAN. Previous construction simulation tools employed closed designs that limit the input and output to what the system provides.

Construction modeling tasks that required work at the level of a doctoral dissertation, and merited several journal articles, can be handled as trivial examples on how to use one of the many modeling capabilities available in Stroboscope. Many such examples have been presented throughout this manuscript. This is not intended to diminish the contribution of the associated work, but rather to provide evidence supporting the system’s robust and flexible design. A more detailed comparison of Stroboscope to several process and project simulation tools appears in Appendix D.
19.1 Fundamental Capabilities Provided by Stroboscope to Support Construction Simulation

Stroboscope provides essential capabilities that enable it to model construction processes at the level of detail necessary to make decisions. These capabilities are outlined in the subsections that follow in terms of what is necessary to support the design of construction processes.

19.1.1 Access to the State of the Simulation

The dynamic nature of construction operations requires that the behavior of a system be defined in terms of the state of the simulation. As the various activities in a process are performed, information that influences the outcome of subsequent activities becomes available. For example, the hauling distance, and therefore the hauling time, in an earth-moving operation for the construction of a road may vary as the road is constructed. The state of the system also affects the various decisions that can be made regarding the remainder of the process. For example, in an earth-moving project for the construction of a dam, the destination of a truck (loaded with dirt) depends on where dirt was dumped in the past as well as on the length of the existing truck queues at all possible dump locations.

19.1.2 Resource Characterization

The properties of resources influence the outcome of activities and the decisions that need to made by the supervisor of an operation. The different types of resources that collaborate must be appropriately matched based on their properties. For example, it may be more efficient to use a big tractor to push-load a big scraper, and a small tractor to push-load a small scraper. Some resource combinations may or may not be suitable to undertake a task. For example, the power and boom length of a crane determine the loads it is capable of carrying. In addition, the outcome of a particular task may be highly
sensitive to the properties of the equipment used to perform the task. For example, a
large tractor may be able to push-load a scraper much faster than a small tractor.

A simulation tool must not only be able to recognize the properties of resources,
but must also characterize resources in persistent and dynamic ways on an individual
basis. For example, in order to determine when a particular piece of equipment needs to
re-fuel, the piece of equipment must carry information that indicates the amount of fuel
currently in its tank. This information must be updated as the equipment performs tasks
that consume fuel.

19.1.3 Compound Resources

Resources in construction are often combined and for some time thereafter act as
a single entity. Later, some other resources may join in, or some of the original resources
may detach from the group. A construction simulation tool must be able to model such a
group as a compound resource and to characterize the compound resource depending on
the properties of its components and the experiences of the resource. For example, a
simulation tool must be able to combine a flatbed and the several steel shapes it may
carry into a single resource. The resulting resource may be characterized according to its
gross weight, which depends on the weight of the flatbed and the weight of the steel
shapes it carries. As steel shapes are unloaded or additional shapes are loaded, it should
be possible to update the gross weight of the loaded flatbed.

19.1.4 Bulk Resources

In construction processes, bulk resources are frequently consumed, used, and
produced in stochastic amounts. A construction simulation system must be able to
represent these resources effectively. Typical examples are the amount of rock fragments
that result from a dynamite blast and the amount of soil scraped by a push-loaded
scraper.
19.1.5 Programmability - Simulation Languages Vs. Simulators

The properties of resources and the state of the system are simultaneous factors that affect operations in a number of ways that cannot be determined a priori by the designer of a simulation system. As a consequence, the interaction between the properties of resources and the state of the simulation must be determined specifically for each simulation model (not as part of the design of the simulation system). The only way to achieve this is by making the simulation system programmable.

Examples of how state and resource properties are interrelated abound. In a dirt-hauling operation it may be important to obtain the ratio of the capacity of a hauler to the bucket size of a loader to figure out how many loads are required to fill the hauler. It may be necessary to perform some computations based on the density and amount of soil, hauler horsepower, and terrain slope to estimate hauling speed. To decide whether a hauler is appropriate for a task, it may be necessary to look at its height, at its capacity, and at its length and compare them to the maximum height of an underpass, maximum payload of a road, and the tightest corner that needs to be turned. To determine the amount of plastic concrete that results from mixing certain amounts of cement, aggregates and water, computations that go beyond simple addition must be performed. The list of possible interrelationships that can be encountered can go on endlessly.

An alternative to programmability that was considered in Strobosope’s design, but that was promptly ruled out, was to use “templates”. Templates are used to specify interrelationships that fall into a predetermined pattern. For example, a template may allow us to compare the property of a resource to a number and give us a choice of several relational operators for the comparison. Or it may be enhanced by allowing the property to be multiplied by a factor before the comparison. Essentially, a template is a formula of pre-defined form, in which the user can choose the operands and operators, sometimes supplying a value and other times choosing from a list of available alternatives. A system can acquire some flexibility through templates and a small group of them can cover many situations.

No matter how many templates a system provides, however, the diverse and complex nature of construction processes will always require a function of a form not
foreseen by the system designer (or requiring a choice of operands or operators not provided in the template).

If a system designer decides to provide templates to cover what she considers to be a substantial percentage of all possible situations, the number of templates will be very large. In spite of this, when a truly innovative process requires study through simulation, the probability that a template unforeseen by the system designer is required is considerable (no matter how exhaustive the provided templates are).

Templates also have implications regarding system complexity. As the number of templates increases, the burden placed on the user also increases. For each template, a user will need to know not only of the availability of the template, but also about its details. This results in a paradox. That is, a simulation system that lacks a feature bound to be needed at a crucial time, but yet so complex that very few will be able to exploit its full potential.

A system that moves away from templates towards programmability provides a considerable increase in power and flexibility, and at the same time reduces system complexity. Programmability allows for complex interrelationships between variables that have forms that are unpredictable until the need for them arises.

The design philosophy of a programmable system is different from that of a template based system. The issue is no longer to try to predict the possible interrelationships between variables (this can be taken care of at the time the problem arises). Instead, the problem is to determine what variables to make accessible, how to make them accessible, and what operators and functions to provide. When users need to express complex variable relationships, they only need to appropriately combine the variables, operators, and functions to which they have access. This has the advantage that the form of the function need not be known until the need to use it arises.

## 19.2 Contributions

Stroboscope is a radical departure from previous construction simulation systems in that it was designed as a simulation programming language that provides seamless and dynamic access to the state of the simulation and the properties of resources. It is capable
of modeling the highly complex and dynamic processes encountered in construction with unprecedented ease.

The following list summarizes some of Stroboscope’s contributions to the state-of-the-art.

- A framework that provides dynamic and comprehensive access to the state of the simulation through pre-defined system-maintained variables.

- A framework that provides dynamic access to the properties of resources at the individual and set level through pre-defined system-maintained variables.

- The categorization and implementation of resource properties into the following classifications:
  - Essential properties such as unique identifier, time of creation, and time of entry to recent nodes that are defined and updated by the system automatically.
  - Static properties that are common to a group of resources. Such properties are generally found in tables and include such things as the size of a loader’s bucket and the power of its engine (these are common throughout all loaders of the same model).
  - Properties that are unique to each individual resource and that can be strategically updated according to each resource’s experience such as the amount of fuel currently in a truck’s fuel tank.
  - Properties defined in terms of other properties and the state of the simulation, and which are kept up to date by the system, such as the number of hours that a piece of equipment can work without exhausting its fuel reserves.

- The behavioral analysis and abstraction of conditional Activities (Combis) into methods (attributes) that allow them to:
  - Attain a dynamically determined priority based on the state of the simulation and the properties of resources (Priority).
• Allow or disallow their activation based on arbitrarily complex rules not related to resource availability (Semaphore).

• Allow their activation only when very precisely and dynamically established sets of resources are available (Enough).

• Faithfully enforce resource selection schemes that resemble the process by which resources are selected in the field (DrawOrder, DrawWhere, DrawUntil).

• The design of an auxiliary node release graph scheme that can handle complex resource routing, as well as assembly and disassembly in a straightforward and logical manner. A properly configured release graph can accomplish the following tasks:
  
  • It can route a resource probabilistically or by decision based on the properties of the resource, the properties of other resources, and the state of the simulation.

  • It can combine several resources into a compound resource that acts as a single entity. The compound resource itself can be combined with other resources to produce yet another compound resource. Thus, compound resources can form containment hierarchies of unlimited depth.

  • It can remove components from within a compound resource without otherwise altering the compound resource.

  • It can add new components to a compound resource without otherwise altering the compound resource.

  • It can characterize a compound resource according to its components and to the state of the simulation.

• The identification of precisely defined strategic points in the logic of a simulation where:

  • Values can be assigned to variables and arrays.
• Statistics can be collected.

• Formatted output can be printed.

• An Add-On Interface Specification that allows Stroboscope to be extended seamlessly using high-level compiled languages and without the need to link statically with the Stroboscope engine.

• A Three-Phase Activity Scanning executive that prevents zero-duration Activities from introducing undesirable side-effects in the simulation logic.

In addition, Stroboscope has the following desirable capabilities and features that enhance its suitability to model construction operations:

• It can produce highly customized output that can pinpoint the important results so that they are not lost in the otherwise vast amounts of data collected and reported by the system. For example, it can present the results of a simulation in terms of total or unit cost statistics.

• A language based on a syntax that is easy to remember, read, and maintain.

• An Integrated Development Environment that allows simulation models to be edited, run, and debugged easily.

• A Graphical User Interface that can be used to create simulation networks using drag and drop drawing. The GUI can run models directly and can also generate the Stroboscope source code for simulation models.

• Statistics collection objects that include:
  • Standard Collectors
  • Binned Collectors
  • Moving Average Collectors
  • Weighted Collectors
  • Time-Weighted Collectors
• One and two-dimensional arrays.

• A powerful statement-level flow control language that includes if-elseif-else-endif blocks as well as while-wend blocks that can be nested within each other.

• A statement preprocessing operator that can be used to generate Stroboscope language statements automatically.

• Sophisticated random number stream management that allows the implementation of variance reduction techniques that include, but is not limited to, antithetic variates and common random numbers.

• The ability to call functions defined in external Dynamic Link Libraries (DLLs) without re-linking.

The contribution of this research goes beyond those listed above. The paramount contribution is the design of the language; the manner in which its capabilities can model truly complex construction processes with unprecedented ease.

Perhaps the strongest evidence of the language’s suitability is that it is already used to teach construction simulation courses in the University of Michigan as well as in other universities in the United States and Europe. It is expected that in time this will extend to the construction industry.

19.3 Validation and Implementation

In some areas of research the implementation of an idea is separate from the idea itself. For example, a new method for the solution of a linear programming problem could be sound, precise and effective. It could be validated and verified on paper without the need for a corresponding computer program, or it could be validated and verified with a poorly written computer program that is only capable of solving a few small problems.

In simulation, as well as in some other areas of computer-oriented research, this is not the case. Simulation modeling is almost exclusively a computer-based activity. A
simulation tool that is not well implemented cannot be used to validate and verify the ideas behind it.

In simulation, it is very easy to believe that the conceptual model of a process is correct when in reality it is not. When the model is implemented and carefully examined, incorrect concepts show up and make us aware of our misconceptions. This phenomenon carries on to the simulation tool. The concepts behind the tool need to be exercised with process models that are validated and verified, and that in turn allow the validation and verification of the modeling tool itself.

Poor conceptual designs are very difficult or impossible to implement well. In contrast, good designs of even complex systems are relatively easy to implement. Stroboscope’s implementation is powerful, accurate, fast, and simple to use. It is proof of the soundness of its conceptual design.

The Stroboscope language as described in this manuscript is implemented as a simulation engine (Strobosc.dll) that serves several front ends: a Graphical User Interface hosted on a popular drawing package (StrGUI32.vsl, StrobGUI.vsl, StrbTempl.vst, Strobos.vss), an Integrated Development Environment (StrbWin.exe), a dialog based driver (MacStrbW.exe), a windows-based command line tool (StrbMini.exe), and a console-based command line tool (Strobos.exe).

The simulation engine, driven by the various front ends, has been used to model hundreds of nontrivial processes in construction and other domains. The Stroboscope models for these processes have been carefully validated and verified, and used to validate and verify Stroboscope’s conceptual design.

19.4 Capabilities, Limitations and Directions for Future Research

Stroboscope uses dynamic memory to store all the elements involved in a simulation model. As a result, the number of nodes, number of resource types, number of properties, number of programming objects, and model size are only limited by available
memory. The only built-in limitations are the length of statements (4096 characters maximum), and the number of OutFiles (256 maximum per model).

Stroboscope can be enhanced at both the conceptual design and implementation level. In terms of design, there are three issues that need to be researched:

- How to model the preemption of activities so that active resources can be claimed by other tasks with higher priority. Modeling preemption requires the consideration of the following issues:
  - Which of the several possible simultaneous instances of an Activity should be preempted?
  - Once the specific instance that will be preempted is identified, what will happen with other resources held by the instance?
  - Once the preemptor is done with the resources, how does the preempted instance continue?
  - How does the preempted instance adjust its duration?

- How to model continuous activities that consume and produce resources at rates determined by the state of the simulation and the properties of resources.

- How to modify the design of the engine to allow the representation of resource types hierarchically, so that derived resources can be treated as if they were objects of a superclass.

In terms of implementation, Stroboscope could be enhanced to:

- Include variables of intrinsic types other than double-precision floating point numbers, such as integers and strings.

- Include variables to represent modeling elements so that the modeling elements can be referenced indirectly and so that they can be placed in arrays.

- Allow the definition of functions, within a model file, that take arguments and have local variables and flow control.
Appendices
Appendix A

Stroboscope Reference

This appendix lists the functions, statements, pre-defined variables, and action-targets available in the Stroboscope language. The lists use abbreviations that represent user-defined identifiers. Most abbreviations are self-explanatory. Some abbreviations may need explanation, these are shown in Table 16 below:

Table 16 - Abbreviations for User-Defined Identifiers

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity</td>
<td>A Combi, a Consolidator, or a Normal</td>
</tr>
<tr>
<td>CursorSupporter</td>
<td>An Assembler, Characterized Dynafork, Characterized Link, Characterized Queue, DisAssembler, or Filter</td>
</tr>
<tr>
<td>Customer</td>
<td>A SubType or a Compound Resource Type</td>
</tr>
<tr>
<td>HeteroHolder</td>
<td>A MultiReceiver or a DisAssembler</td>
</tr>
<tr>
<td>Hotel</td>
<td>An Activity, a Characterized Queue, or a Compound Resource Type</td>
</tr>
<tr>
<td>MultiReceiver</td>
<td>An Activity or an Assembler</td>
</tr>
</tbody>
</table>

Operators

Stroboscope expressions and operators are similar to those of the C programming language. Table 17 summarizes the available operators and their precedence. Operators with higher precedence are resolved first. If two operators have the same precedence, they are evaluated from left to right.

Table 17 - Stroboscope Operators and Their Precedence

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&lt; &gt;$</td>
<td>Preprocessor replacement</td>
<td>1 (highest)</td>
</tr>
<tr>
<td>( )</td>
<td>Parentheses</td>
<td>2</td>
</tr>
</tbody>
</table>
### Operator Precedence

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ]</td>
<td>Function call</td>
<td>2</td>
</tr>
<tr>
<td>[ ]</td>
<td>Array subscripts</td>
<td>2</td>
</tr>
<tr>
<td>!</td>
<td>Logical NOT</td>
<td>3</td>
</tr>
<tr>
<td>-</td>
<td>Negation</td>
<td>3</td>
</tr>
<tr>
<td>^</td>
<td>Power</td>
<td>4</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>5</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>5</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>6</td>
</tr>
<tr>
<td>+</td>
<td>Addition</td>
<td>6</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
<td>7</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>7</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
<td>7</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>7</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal to</td>
<td>8</td>
</tr>
<tr>
<td>==</td>
<td>Equal to</td>
<td>8</td>
</tr>
<tr>
<td>&amp;</td>
<td>Logical AND</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Logical OR</td>
</tr>
<tr>
<td>? :</td>
<td>Conditional (IF)</td>
<td>11 (lowest)</td>
</tr>
</tbody>
</table>

### Functions

Stroboscope programming functions are called with the name of the function followed by a list of arguments enclosed in square brackets. Arguments are separated by commas.

The functions available in Stroboscope are listed in Table 18 below.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abs[val]</td>
<td>Absolute value of val</td>
</tr>
<tr>
<td>Acos[val]</td>
<td>ArcCosine of val</td>
</tr>
<tr>
<td>Antithetics[]</td>
<td>Status of system antithetic sampling generation</td>
</tr>
<tr>
<td>ArrayName[index]</td>
<td>Value stored at position index in array ArrayName</td>
</tr>
<tr>
<td>Asin[val]</td>
<td>ArcSine of val</td>
</tr>
<tr>
<td>Atan[val]</td>
<td>ArcTangent of val</td>
</tr>
<tr>
<td>AtanXdivY[x,y]</td>
<td>ArcTangent of x/y</td>
</tr>
<tr>
<td>Beta[a,b]</td>
<td>Sample from unit Beta distribution</td>
</tr>
<tr>
<td>sBeta[a,b,s]</td>
<td>Sample from unit Beta distribution using stream s</td>
</tr>
<tr>
<td>Confidence[SD,lvl,nSamples]</td>
<td>Half width of a confidence interval</td>
</tr>
<tr>
<td>Cos[val]</td>
<td>Cosine of val</td>
</tr>
<tr>
<td>Cosh[val]</td>
<td>Hyperbolic cosine of val</td>
</tr>
<tr>
<td>Erlang[order,mean]</td>
<td>Sample from Erlang distribution</td>
</tr>
<tr>
<td>sErlang[order,mean,s]</td>
<td>Sample from Erlang distribution using stream s</td>
</tr>
<tr>
<td>Exp[val]</td>
<td>Exponential function of val</td>
</tr>
<tr>
<td>sExponential[mean,s]</td>
<td>Sample from Exponential dist. using stream s</td>
</tr>
<tr>
<td>Gamma[a,b]</td>
<td>Sample from Gamma distribution</td>
</tr>
<tr>
<td>sGamma[a,b,s]</td>
<td>Sample from Gamma distribution using stream s</td>
</tr>
<tr>
<td>Int[val]</td>
<td>Integral part of val</td>
</tr>
<tr>
<td>LastRnd[]</td>
<td>Last Random Number</td>
</tr>
<tr>
<td>sLastRnd[s]</td>
<td>Last Random Number retrieved from stream s</td>
</tr>
<tr>
<td>Ln[val]</td>
<td>Natural logarithm of val</td>
</tr>
<tr>
<td>Log[val]</td>
<td>Base_{10} logarithm of val</td>
</tr>
<tr>
<td>Matrix[row, column]</td>
<td>Value stored in row, column within array Matrix</td>
</tr>
<tr>
<td>Max[val1, val2]</td>
<td>Maximum of val1 and val2</td>
</tr>
<tr>
<td>Min[val1, val2]</td>
<td>Minimum of val1 and val2</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>------------------------------------------------------------------</td>
</tr>
<tr>
<td>Mod[(val,div)]</td>
<td>Remainder of (val / div)</td>
</tr>
<tr>
<td>Normal[(mean,stdev)]</td>
<td>Sample from normal distribution</td>
</tr>
<tr>
<td>sNormal[(mean,stdev,s)]</td>
<td>Sample from normal distribution using stream (s)</td>
</tr>
<tr>
<td>NormalInv[(mean,stdev,Cumulative)]</td>
<td>Inverse of the normal distribution</td>
</tr>
<tr>
<td>OffSetSeed[(seed,positions)]</td>
<td>The seed separated by (positions) from (seed)</td>
</tr>
<tr>
<td>Pert[(p0,mode,p100)]</td>
<td>Sample from PERT Beta distribution</td>
</tr>
<tr>
<td>sPert[(p0,mode,p100,s)]</td>
<td>Sample from PERT Beta dist. using stream (s)</td>
</tr>
<tr>
<td>Pertpg[(p5,mode,p95)]</td>
<td>Sample from Perry &amp; Grieg Beta distribution</td>
</tr>
<tr>
<td>sPertpg[(p5,mode,p95,s)]</td>
<td>Sample from Perry &amp; Grieg Beta using stream (s)</td>
</tr>
<tr>
<td>Rnd[]</td>
<td>Random Number</td>
</tr>
<tr>
<td>sRnd[(s)]</td>
<td>Random Number from stream (s)</td>
</tr>
<tr>
<td>Round[(expression,decimals)]</td>
<td>Round to decimal places (which can be negative)</td>
</tr>
<tr>
<td>ScaledBeta[(low,high,a,b)]</td>
<td>Sample from scaled Beta distribution</td>
</tr>
<tr>
<td>sScaledBeta[(low,high,a,b,s)]</td>
<td>Sample from scaled Beta dist. using stream (s)</td>
</tr>
<tr>
<td>sSeed[(s)]</td>
<td>The current seed for stream (s)</td>
</tr>
<tr>
<td>Sin[(val)]</td>
<td>Sine of (val)</td>
</tr>
<tr>
<td>Sinh[(val)]</td>
<td>Hyperbolic sine of (val)</td>
</tr>
<tr>
<td>Sqrt[(val)]</td>
<td>Square root of (val)</td>
</tr>
<tr>
<td>StdNormalInv[(Cumulative)]</td>
<td>Inverse of the standard normal distribution</td>
</tr>
<tr>
<td>Tan[(val)]</td>
<td>Tangent of (val)</td>
</tr>
<tr>
<td>Tanh[(val)]</td>
<td>Hyperbolic tangent of (val)</td>
</tr>
<tr>
<td>tInv[(alpha,DegFreedom)]</td>
<td>Inverse of the t Distribution</td>
</tr>
<tr>
<td>Triangular[(low,mode,high)]</td>
<td>Sample from Triangular distribution</td>
</tr>
<tr>
<td>sTriangular[(low,mode,high,s)]</td>
<td>Sample from Triangular dist. using stream (s)</td>
</tr>
<tr>
<td>Uniform[(low,high)]</td>
<td>Sample from Uniform distribution</td>
</tr>
<tr>
<td>sUniform[(low,high,s)]</td>
<td>Sample from Uniform distribution using stream (s)</td>
</tr>
</tbody>
</table>
System-Maintained Variables

Stroboscope allows expressions to access the state of the simulation and the properties of resources through predefined variables. Variables are classified according to the scope in which they can be used as global, instance, or cursor variables.

Stroboscope creates most of the predefined variables from user-defined modeling elements. The list that follows contains names in italics that must be replaced with the appropriate user-defined names.

The following predefined variables are available:

**Global variables:**

CurSeed
RelTime
SimTime

*Activity.* [AveDur | AveInter | CurInst | FirstStart | LastStart | MaxDur | MaxInter | MinDur | MinInter | RelTotInst | SDDur | SDInter | TotInst]

*HeteroHolder.* InContext

*Hotel.Customer.* [AveEnts | AvTotTm | AvVstTm | SDAvVst | MnVstTm | MxVstTm]

*CharQue.Filter.* Count

*CharQue.Filter.Property.* [AveVal | MaxVal | MinVal | SDVal | SumVal]

*CharQue.Property.* [AveVal | MaxVal | MinVal | SDVal | SumVal]

*CharQue.SubType.* Count

*CharType.* [AveLife | MaxLife | MinLife | SDLife]

*CharType.* [AvePp | CurPp | MaxPp | MinPp | SDPp | TotPp]

*Collector.* [AveVal | MaxVal | MinVal | nSamples | SDVal | SumVal]

*CursorSupporter.* HasCursor

*GenReleaseLink.* LastAmtReleased

*GenLinkOutOfQueue.* LastAmtDrawn

*GenQueue.* LastAmtReceived

*Queue.* [AveCount | AveWait | CurCount | MaxCount | MinCount | SDCount | TotCount]

*SubType.* [AveLife | MaxLife | MinLife | SDLife]
SubType.{AvePp | CurPp | MaxPp | MinPp | SDPp | TotPp}
SubType.SubTypeProperty
TmWgtCollector.[AveVal | MaxVal | MinVal | SDVal | TtlWgt]
WgtCollector.{AveVal | MaxVal | MinVal | SDVal | TtlWgt}

Instance variables:
Activity.{Duration | Instance}
HeteroHolder.Filter.Property.[AveVal | MaxVal | MinVal | SDVal | SumVal]
HeteroHolder.Filter.Count
HeteroHolder.CharType.Property.[AveVal | MaxVal | MinVal | SDVal | SumVal]
HeteroHolder.ResourceType.Count
HeteroHolder.SubType.Count
OutOfQueLink.{AveDrawDur | MaxDrawDur | MinDrawDur | nDraws | SDDrawDur | SumDrawDur}

Cursor variables:
[CursorSupporter.]BirthTime
[CursorSupporter.]ResNum
[CursorSupporter.]SaveProp
[CursorSupporter.]SubTypeProperty
[CursorSupporter.]TimeIn

Statements

Stroboscope programming statements can span several lines with arguments separated by white-space. Arguments that include white-space must be enclosed in single quotations; text arguments must be enclosed in double quotations. A statement ends with a semicolon.

Comments are placed in source files by making the first non-white-space character after a statement a “/”. The comment continues until the end of the line.

The statements defined by the Stroboscope language are shown in Table 19 below.
Table 19 - Stroboscope Statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANTIUHETICS;</td>
<td>[BooleanExpThatTurnsOnOrOffAntitheticSampling];</td>
</tr>
<tr>
<td>APPFILE</td>
<td>Alias DiskFileName;</td>
</tr>
<tr>
<td>ARRAY</td>
<td>MatrixName Rows Columns {{ InitValue InitValue ...}};</td>
</tr>
<tr>
<td>ARRAY</td>
<td>ArrayName Size {{ InitValue InitValue ...}};</td>
</tr>
<tr>
<td>ASMBASELINK</td>
<td>Link Predecessor Assembler;</td>
</tr>
<tr>
<td>ASSEMBLER</td>
<td>Assembler CompCharTypeAssembled;</td>
</tr>
<tr>
<td>ASSIGN</td>
<td>ActionTarget [TargetArgument] [...]</td>
</tr>
<tr>
<td>BEFOREEND</td>
<td>Activity ActionTarget TargetArguments</td>
</tr>
<tr>
<td>BINCOLLECTOR</td>
<td>BinnedCollector[*] NumberOfBins TopOfFirst BottomOfLast;</td>
</tr>
<tr>
<td>BINQUEUE</td>
<td>BinnedQueue ResourceType NumberOfBins TopOfFirst BotOfLast;</td>
</tr>
<tr>
<td>BREAK;</td>
<td></td>
</tr>
<tr>
<td>CHARTYPE</td>
<td>CharType [Property] [...]</td>
</tr>
<tr>
<td>CLEAR;</td>
<td></td>
</tr>
<tr>
<td>COLLECT</td>
<td>ActionTarget [TargetArgument] [...]</td>
</tr>
<tr>
<td>COLLECTOR</td>
<td>Collector[*];</td>
</tr>
<tr>
<td>COMBI</td>
<td>Combi;</td>
</tr>
<tr>
<td>COMPTYPE</td>
<td>CompCharType;</td>
</tr>
<tr>
<td>CONSOLIDATOR</td>
<td>Consolidator;</td>
</tr>
<tr>
<td>CONSOLIDATEWHEN</td>
<td>Consolidator BooleanExpression;</td>
</tr>
<tr>
<td>CONTINUE;</td>
<td></td>
</tr>
<tr>
<td>DEBUGOFF;</td>
<td></td>
</tr>
<tr>
<td>DEBUGON;</td>
<td></td>
</tr>
<tr>
<td>DISASMBASELINK</td>
<td>Link DisAssembler Successor;</td>
</tr>
<tr>
<td>DISASSEMBLER</td>
<td>Disassembler CompoundCharTypeDisassembled;</td>
</tr>
<tr>
<td>Statement</td>
<td>Arguments</td>
</tr>
<tr>
<td>-------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>DISCIPLINE</td>
<td>CharQueue CursoredExpression;</td>
</tr>
<tr>
<td>DISPLAY</td>
<td>[QuotedString</td>
</tr>
<tr>
<td>DRAWAMT</td>
<td>OutOfQueueGenLink Expression;</td>
</tr>
<tr>
<td>DRAWDUR</td>
<td>OutOfQueLink Expression;</td>
</tr>
<tr>
<td>DRAWUNTIL</td>
<td>OutOfQueLink Expression;</td>
</tr>
<tr>
<td>DRAWORDER</td>
<td>OutOfQueueCharLink CursoredExpression;</td>
</tr>
<tr>
<td>DRAWWHERE</td>
<td>OutOfQueueCharLink CursoredExpression;</td>
</tr>
<tr>
<td>DUALBASELINK</td>
<td>Link DisAssembler Assembler;</td>
</tr>
<tr>
<td>DURATION</td>
<td>Activity Expression;</td>
</tr>
<tr>
<td>DYNAFORK</td>
<td>Fork ResourceType [Stream];</td>
</tr>
<tr>
<td>ELSE</td>
<td></td>
</tr>
<tr>
<td>ELSEIF</td>
<td>IfExpression;</td>
</tr>
<tr>
<td>ENDIF</td>
<td></td>
</tr>
<tr>
<td>ENDMODEL</td>
<td></td>
</tr>
<tr>
<td>ENOUGH</td>
<td>OutOfQueueLink BooleanExpression;</td>
</tr>
<tr>
<td>FILTER</td>
<td>Filter CharType CursoredExpression;</td>
</tr>
<tr>
<td>FILTEREXP</td>
<td>Filter CursoredExpression;</td>
</tr>
<tr>
<td>FORK</td>
<td>Fork ResourceType [Stream];</td>
</tr>
<tr>
<td>FUNCTION</td>
<td>NameInStrobo DllName NameInDll nArguments [CONSTANT];</td>
</tr>
<tr>
<td>GENTYPE</td>
<td>GenType;</td>
</tr>
<tr>
<td>IF</td>
<td>IfExpression;</td>
</tr>
<tr>
<td>INIT</td>
<td>SimpleCharQueue PositiveIntExpression SubType;</td>
</tr>
<tr>
<td>INIT</td>
<td>CompoundCharQueue PositiveIntExpression;</td>
</tr>
<tr>
<td>INIT</td>
<td>GenQueue PositiveFloatExpression;</td>
</tr>
<tr>
<td>LINK</td>
<td>Link Predecessor Successor [ResourceType];</td>
</tr>
<tr>
<td>LOADADDON</td>
<td>DllName;</td>
</tr>
<tr>
<td>MVAVGCOLLECTOR</td>
<td>MvAvgCollector[*] MaxSamplesExpression;</td>
</tr>
<tr>
<td>Statement</td>
<td>Arguments</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>NOENOUGHS</td>
<td><em>Combi</em>;</td>
</tr>
<tr>
<td>NORMAL</td>
<td><em>Normal</em>;</td>
</tr>
<tr>
<td>ONASSEMBLY</td>
<td><em>Assembler</em> {ActionTarget</td>
</tr>
<tr>
<td>ONDISASSEMBLY</td>
<td><em>Disassembler</em> {ActionTarget</td>
</tr>
<tr>
<td>ONDRAW</td>
<td><em>OutOfQueueLink</em> {ActionTarget</td>
</tr>
<tr>
<td>ONENTRY</td>
<td><em>Queue</em> {ActionTarget</td>
</tr>
<tr>
<td>ONEND</td>
<td><em>Activity</em> ActionTarget TargetArguments;</td>
</tr>
<tr>
<td>ONFLOW</td>
<td><em>Link</em> ActionTarget TargetArguments;</td>
</tr>
<tr>
<td>ONRELEASE</td>
<td><em>ReleaseLink</em> {ActionTarget</td>
</tr>
<tr>
<td>ONSTART</td>
<td><em>Activity</em> ActionTarget [TargetArgument] [..];</td>
</tr>
<tr>
<td>OUTFILE</td>
<td><em>Alias</em> DiskFileName;</td>
</tr>
<tr>
<td>PRINT</td>
<td><em>ActionTarget</em> [TargetArgument] [..];</td>
</tr>
<tr>
<td>PRIORITY</td>
<td><em>Combi Expression</em>;</td>
</tr>
<tr>
<td>QUEUE</td>
<td><em>Queue</em> ResourceType;</td>
</tr>
<tr>
<td>RELEASEAMT</td>
<td><em>GenReleaseLink</em> Expression;</td>
</tr>
<tr>
<td>RELEASEUNTIL</td>
<td><em>CharReleaseLink</em> BooleanExpression;</td>
</tr>
<tr>
<td>RELEASEORDER</td>
<td><em>CharReleaseLink</em> CursoredExpression;</td>
</tr>
<tr>
<td>RELEASEWHERE</td>
<td><em>CharReleaseLink</em> CursoredExpression;</td>
</tr>
<tr>
<td>REPORT</td>
<td>[Outfile];</td>
</tr>
<tr>
<td>RESETSTATS;</td>
<td></td>
</tr>
<tr>
<td>REVORDER</td>
<td><em>CharLinkOutOfQueue</em></td>
</tr>
<tr>
<td>SAVEPROPS</td>
<td><em>CharType</em> Property [Property] [..];</td>
</tr>
<tr>
<td>SAVEVALUE</td>
<td><em>Variable[</em>] Expression*;</td>
</tr>
<tr>
<td>SEED</td>
<td><em>Expression</em>;</td>
</tr>
<tr>
<td>SEEDALL</td>
<td><em>Expression</em> [SeparationInHundredThousands];</td>
</tr>
<tr>
<td>Statement</td>
<td>Arguments</td>
</tr>
<tr>
<td>-----------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SEEDN</td>
<td>Stream Expression;</td>
</tr>
<tr>
<td>SEMAPHORE</td>
<td>Combi BooleanExpression;</td>
</tr>
<tr>
<td>SILENTREPLICATE</td>
<td>[BooleanExpThatTurnsOnOrOff SilentReplications];</td>
</tr>
<tr>
<td>SIMULATE</td>
<td></td>
</tr>
<tr>
<td>SIMULATEUNTIL</td>
<td>BooleanExpression;</td>
</tr>
<tr>
<td>STATEMENT</td>
<td>Alias DllName NameInDll;</td>
</tr>
<tr>
<td>STREAMS</td>
<td>NumberOfRandomNumberStreams;</td>
</tr>
<tr>
<td>STRENGTH</td>
<td>OutOfForkLink Expression;</td>
</tr>
<tr>
<td>SUBTYPE</td>
<td>CharType SubType [Expression] [...];</td>
</tr>
<tr>
<td>TMWGTCOLLECTOR</td>
<td>TmWgtSaveVal[*] Expression;</td>
</tr>
<tr>
<td>VARIABLE</td>
<td>Variable Expression;</td>
</tr>
<tr>
<td>VARPROP</td>
<td>CharType Property AnonymouslyCursoredExpression;</td>
</tr>
<tr>
<td>WEND</td>
<td></td>
</tr>
<tr>
<td>WGTCOLLECTOR</td>
<td>WgtCollector[*];</td>
</tr>
<tr>
<td>WHILE</td>
<td>WhileExpression;</td>
</tr>
</tbody>
</table>

### Action Targets

Stroboscope events operate on action targets. Some of these are pre-defined and others come from the names of user defined objects. The operation to be performed on a target in a particular event depends on the arguments passed to the target. All targets take arguments in the following form:

Event *Element Target* [PRECOND *BooleanExpression*] [TargetSpecificArgs ...];

The following action targets are available:

#### Single LValues:

Targets to which a single double precision floating point value can assigned or added to.

**Arguments:**
One single argument consisting of an expression, the result of which will be assigned or added to the target.

**Targets:**

- `CharQueue.CharType.SaveProp`
- `CharQueue.Filter.SaveProp`
- `Collector`
- `[CursorSupporter.]SaveProp`
- `HeteroHolder.Chartype.SaveProp`
- `HeteroHolder.Filter.SaveProp`
- `MvAvgCollector`
- `SaveValue`
- `TmWgtCollector`

**Weighted Collector LValues:**

Targets to which a single double precision floating point value can be applied with a weight determined by another double precision floating point value.

**Arguments:**

Two arguments consisting of expressions, the result of the first expression is added to the statistics kept by the target, with a weight determined by the result of the second expression.

**Targets:**

- `WgtCollector`
**Array LValues:**

Singly indexed set of double precision floating point values. A value can be assigned to a particular element given an index, or all the values in the set can be initialized.

**Arguments:**

When setting the value of a particular element: Two arguments consisting of expressions; the first expression gets truncated to an integer that indicates the index within the array; the result of the second expression will be assigned to the indexed element in the array.

When setting values to all the elements: A list of expressions separated by white-space enclosed in curly braces ‘{‘ and ‘}’. The result of each expression will be assigned to the corresponding element in the array.

**Targets:**

`ArrayName`

**Matrix LValues:**

Doubly indexed (row, column) set of double precision floating point values. A value can assigned to a particular element given a row and a column, or all the values in the set can be initialized.

**Arguments:**

When setting the value of a particular element: Three arguments consisting of expressions; the first expression gets truncated to an integer that indicates the row within the matrix; the second expression gets truncated to an integer that indicates the column within the matrix; the result of the third expression will be assigned to the indexed element in the matrix.

When setting values to all the elements: A list of expressions separated by white-space enclosed in curly braces ‘{‘ and ‘}’. The result of each expression will be assigned to the corresponding element in the matrix.

**Targets:**

`MatrixName`
GENERATE:

A predefined target that creates characterized resources.

Arguments:

An expression whose value gets truncated to an integer, followed by the name of a SubType or Compound characterized type. The value of the expression determines the number of resources of the specified SubType or Compound type to be created.

Restrictions:

The GENERATE target can only be used in the BEFOREDRAWS and BEFOREEND events. The created resources join the other resources held by the starting Combi or ending Activity.

Outfiles:

Targets that store formatted text output.

Arguments:

A Format String enclosed in double quotations is required as first argument. One additional argument consisting of an expression is required for each Format Specifier in the Format String. Format Specifiers are substituted with the result of the corresponding expression.

See page 431 for a detailed explanation of the Format String and Format Specifiers.

Targets:

AppFile
OutFile
StdError
StdOutput
StdTrace
**Format Strings & Format Specifiers**

The Format String argument consists of ordinary characters, escape sequences, and (if other arguments follow Format String) format specifications. The ordinary characters and escape sequences are copied to the OutFile in order of their appearance.

If arguments follow the format string, the format string must contain specifications that determine the output format for the arguments. Format specifications always begin with a percent sign (%) and are read left to right. When the first format specification (if any) is encountered, the value of the first argument after format is converted and output accordingly. The second format specification causes the second argument to be converted and output, and so on. There must be an equal number of extra arguments as there are format specifications.

**Format Specification Fields**

A format specification, which consists of optional and required fields, has the following form:

```
%[flags] [width] [.precision] type
```

Each field of the format specification is a single character or a number signifying a particular format option. The simplest format specification contains only the percent sign and a `type` character (for example, `%f`). The fields in a format specification are described in the following list:

- **type**: Required character that determines whether the associated argument is interpreted as a decimal point number, a number in scientific format, or a combination of the two.
- **flags**: Optional character or characters that control justification of output and printing of signs, blanks, and decimal points. More than one flag can appear in a format specification.
- **width**: Optional number that specifies minimum number of characters output.
- **precision**: Optional number that specifies the maximum number of characters printed for all or part of the output field.
If a percent sign is followed by a character that has no meaning as a format field, the character is copied to the OutFile. For example, to print a percent-sign character, use `%%`.

**Type Field Characters**

The *type* character is the only required format field; it appears after any optional format fields. The *type* character determines whether the associated argument is output in decimal point notation, scientific notation, or a suitable choice between the two, as indicated in Table 20 below.

**Table 20 - Type Field Characters**

<table>
<thead>
<tr>
<th>Character</th>
<th>Output format</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>Signed value having the form <code>[-]dddd.dddd</code>, where <code>dddd</code> is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number, and the number of digits after the decimal point depends on the requested precision.</td>
</tr>
<tr>
<td>e</td>
<td>Signed value having the form <code>[-]d.dddd e [sign]ddd</code>, where <code>d</code> is a single decimal digit, <code>dddd</code> is one or more decimal digits, <code>ddd</code> is exactly three decimal digits, and <code>sign</code> is <code>+</code> or <code>-</code>.</td>
</tr>
<tr>
<td>E</td>
<td>Identical to the <code>e</code> format, except that <code>E</code>, rather than <code>e</code>, introduces the exponent.</td>
</tr>
<tr>
<td>g</td>
<td>Signed value printed in <code>f</code> or <code>e</code> format, whichever is more compact for the given value and precision. The <code>e</code> format is used only when the exponent of the value is less than <code>-4</code> or greater than or equal to the precision argument. Trailing zeros are truncated, and the decimal point appears only if one or more digits follow it.</td>
</tr>
<tr>
<td>G</td>
<td>Identical to the <code>g</code> format, except that <code>E</code>, rather than <code>e</code>, introduces the exponent (where appropriate).</td>
</tr>
</tbody>
</table>
Flag Directives

The first optional field of the format specification is flag. A flag directive is a character that justifies output and prints signs, blanks, and decimal points. More than one flag directive may appear in a format specification, as shown in Table 21 below.

Table 21 - Flag Directives

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Left align the result within the given field width.</td>
<td>Right align.</td>
</tr>
<tr>
<td>+</td>
<td>Prefix the output value with a sign (+ or -).</td>
<td>Sign appears only for negative signed values (-).</td>
</tr>
<tr>
<td>0</td>
<td>If width is prefixed with 0, zeros are added until the minimum width is reached. If 0 and - appear, the 0 is ignored.</td>
<td></td>
</tr>
<tr>
<td>blank (' ')</td>
<td>Prefix the output value with a blank if the output value is signed and positive; the blank is ignored if both the blank and + flags appear.</td>
<td>No blank appears.</td>
</tr>
<tr>
<td>#</td>
<td>When used with the e, E, or f format, the # flag forces the output value to contain a decimal point in all cases.</td>
<td>Decimal point appears only if digits follow it.</td>
</tr>
<tr>
<td>#</td>
<td>When used with the g or G format, the # flag forces the output value to contain a decimal point in all cases and prevents the truncation of trailing zeros.</td>
<td>Decimal point appears only if digits follow it. Trailing zeros are truncated.</td>
</tr>
</tbody>
</table>

Width Specification

The second optional field of the format specification is the width specification. The width argument is a nonnegative decimal integer controlling the minimum number of characters printed. If the number of characters in the output value is less than the specified width, blanks are added to the left or the right of the values (depending on whether the - flag (for left alignment) is specified) until the minimum width is reached.
If $width$ is prefixed with 0, zeros are added until the minimum $width$ is reached (not useful for left-aligned numbers).

The $width$ specification never causes a value to be truncated. If the number of characters in the output value is greater than the specified $width$, or $width$ is not given, all characters of the value are printed (subject to the precision specification).

**Precision Specification**

The third optional field of the format specification is the precision specification. It specifies a nonnegative decimal integer, preceded by a period (.), which specifies the number of characters to be printed, the number of decimal places, or the number of significant digits. Unlike the $width$ specification, the precision specification can cause rounding of a floating-point value.

The interpretation of the precision value and the default when precision is omitted depend on the type, as shown in Table 22 below:

**Table 22 - Interpretation of the Precision Specification**

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>e, E</td>
<td>The precision specifies the number of digits to be printed after the decimal point. The last printed digit is rounded.</td>
<td>Default precision is 6; if precision is 0 or the period (.) appears without a number following it, no decimal point is printed.</td>
</tr>
<tr>
<td>f</td>
<td>The precision value specifies the number of digits after the decimal point. If a decimal point appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.</td>
<td>Default precision is 6; if precision is 0, or if the period (.) appears without a number following it, no decimal point is printed.</td>
</tr>
<tr>
<td>g, G</td>
<td>The precision specifies the maximum number of significant digits printed.</td>
<td>Six significant digits are printed, with any trailing zeros truncated.</td>
</tr>
</tbody>
</table>
Escape Characters

Format String may include special sequences of characters, called *escape sequences*, to represent special symbols. The *escape sequences* available in Stroboscope are listed in Table 23 below.

*Table 23 - Escape Sequences*

<table>
<thead>
<tr>
<th>Escape</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>writes a line-feed carriage-return pair to the OutFile</td>
</tr>
<tr>
<td>\t</td>
<td>writes a tab to the OutFile</td>
</tr>
<tr>
<td>\ddd</td>
<td>writes the character whose ASCII code is <em>ddd</em> to the OutFile. <em>ddd</em> must be an integer between 0 and 255.</td>
</tr>
<tr>
<td>\ \</td>
<td>writes a single “\” to the OutFile</td>
</tr>
</tbody>
</table>
Appendix B
Flow Charts

The flow charts that follow illustrate how the Stroboscope simulation engine processes models. These flow charts are not software development aids, but rather provide information that is of interest to the creator of simulation models.
Start

Set SimTime=0. Turn on MCS flag for all Combis. Sort all groups of links in the order they were defined

Evaluate the Terminating Condition: Is it True?

Yes → Simulation Stops: Terminating Condition Reached

No → Sort all Combis according to the result of evaluating their Priority. Ties resolved by order of Combi definition

Combi Instantiation Phase

Evaluate the Terminating Condition: Is it True?

Yes →

No → Advance Clock

Flowchart 1 - Simulation Loop
Flowchart 2 - Combi Instantiation Phase
Flowchart 3 - Combi Instantiation Attempt
Start

Is the Predecessor Queue Empty? (Yes/No)

If Yes:
- Evaluate DrawAmt, remove from the predecessor Queue the smallest of the result or the Content of the Queue
- Perform ONDRAW actions for the link

If No:
- Evaluate the DrawUntil for the link: Is it TRUE? (Yes/No)
  - If Yes: Flowchart 4 - Generic Resource Drawing Process
  - If No:
    - Evaluate the DrawDuration for this link and collect statistics on it
    - Add to the Successor Combi the amount of resource removed from the Predecessor Queue

Flowchart 4 - Generic Resource Drawing Process
Flowchart 5 - Characterized Draw and Release Process
Start

Evaluate the Duration and compute the EndTime of the Activity

Update Duration statistics, Interval statistics, and Instance Counts for the Activity

Package all the resources received by the activity, along with instance number, duration, start and end times into an instance of this activity

Insert event for termination of this instance in the Future Event List after all other instances whose End times are earlier or equal to this instance's End time

Perform the ONSTART actions for this activity

Activity Instantiation Complete

Flowchart 6 - Activity Instantiation
Flowchart 7 - Advance Clock
Start

Put the Activity "In Context"

Perform the BEFOREEND actions for this activity.

Decrement the count of current Instances of the terminating activity

Send a "Get in Context" message to the successor node of link i

i=0

i = i+1

Is i = to the number of Links out of the activity ?

Yes

Update Time-Spent statistics for all resource types still in the instance

Send a "Stop Receiving" message to the successor node of link i

i=0

i = i+1

Is i = to the number of Links out of the activity ?

Yes

Perform the ONEND actions for this activity

Send a "Get Out of Context" message to the successor node of link i

i=0

i = i+1

Is i = to the number of Links out of the activity ?

Yes

Put the Activity Out of Context

Destroy all resources still held by the activity

Activity Termination Complete

Perform the Release Process for Link i

Send a "Get in Context" message to the successor node of link i

i=0

i = i+1

Is i = to the number of Links out of the activity ?

Yes

Put the Activity Out of Context

Activity Termination Complete

Flowchart 8 - Activity Termination
Flowchart 9 - Generic Release Process

- Start
  - Evaluate the ReleaseAmt for this link
  - Perform ONRELEASE actions for the link
  - Create ReleaseAmt of resource of the type that flows through the link
  - Send receive message to the successor node, passing created resource
- Release Complete
Flowchart 10 - Queue Reactions to Messages
Is $i = 0$ the number of old resources in the Queue?

- **Yes**
  - $i = i + 1$
  - **Is $i = 0$ the number of old resources in the Queue?**
  - **Yes**
    - **Insert the entering resource in position $i$ (pushing all others, if any, back)**
  - **No**
    - **Cursor old resource $i$**
    - **Evaluate the Discipline and assign the result to OldTag (old resource $i$ is cursored at this point)**
    - **Is NewTag $\geq$ OldTag?**
      - **No**
      - **Cursor old resource $i$**
      - **Evaluate the Discipline and assign the result to OldTag (old resource $i$ is cursored at this point)**
      - **Is NewTag $\geq$ OldTag?**
        - **No**
        - **Cursor the recently inserted resource**
        - **Insert the entering resource in position $i$ (pushing all others, if any, back)**
        - **Start Resource Insertion Complete**

*Flowchart 11 - Characterized Resource Insertion in Queue*
Flowchart 12 - Normal Activity Reactions to Messages
Reaction to "Get In Context"

Start

\[ \text{MsgCount} = \text{MsgCount} + 1 \]

Is \( \text{MsgCount} = 1 \) ?

No

End Processing "Get In Context"

Yes

Evaluate the \textit{Strength} for all links out of the Fork and randomly select one of them with probability proportional to its \textit{Strength}. This link becomes the active link

Send a "Get In Context" message to the successor connected to the active link

Reaction to "Receive"

Start

Send through the active link the resource received

End Processing "Receive"

Reaction to "Stop Receiving"

\[ \text{MsgCount} = \text{MsgCount} - 1 \]

Is \( \text{MsgCount} = 0 \) ?

No

End Processing "Stop Receiving"

Yes

Relay the "Stop Receiving" message to the successor connected by the active link

Reaction to "Get Out of Context"

Start

Relay the "Get out of Context" message to the successor connected by the active link

End Processing "Get Out of Context"

Flowchart 13 - Fork Reactions to Messages
Generic Resource Send

Start → Perform the ONFLOW actions for this link → Pass the resource to the Successor connected to this link → End Processing "Receive"

Characterized Resource Send

Start → Cursor the resource received → Perform the ONFLOW actions for this link → Pass the resource to the Successor connected to this link → UnCursor the Resource Received → End Processing "Receive"

Flowchart 14 - Sending Resources Through Links
Flowchart 15 - Dynafork and Disassembler Reactions to Get In and Out of Context and to Stop Receiving
Reaction to "Receive" for Generic Resources

Start

Evaluate the Strength for all links out of the Fork and randomly select one of them with probability proportional to its Strength. This link becomes the active link.

Send through the active link the resource received

End Processing "Receive"

Reaction to "Receive" for Characterized Resources

Start

Cursor the received resource

Evaluate the Strength for all links out of the Fork and randomly select one of them with probability proportional to its Strength. This link becomes the active link.

Send through the active link the resource received

Uncursor the resource

End Processing "Receive"

Flowchart 16 - Dynafork Reactions to Receive
Start

Cursor the Compound Resource Received. This also puts the DisAssembler “In Context”

Detach the received compound resource from its components (the latter will be released through the release links)

i=0

Is i = to the number of Release Links out of the DisAsm?

Yes

Perform all ONDISASSEMBLY actions

UnCursor the compound resource received. This also puts the DisAssembler “Out of Context”

No

Perform Release Process for Link i

i = i + 1

Send the compound resource through the DisAssembly Base Link

Re-attach any unreleased components to the received compound resource

End Processing “Receive”

A DisAssembly base link is NOT a Release Link, so it does not go in this loop

Does the DisAsm have a DisAssembly Base Link?

Yes

Destroy any unreleased components and the received compound resource

No
Flowchart 18 - Assembler Reactions to Get In Context and Receive

Reaction to "Get In Context"

Start

MsgCount = MsgCount + 1

Is MsgCount = 1?

End Processing "Get In Context"

Yes

Turn On the InContext Flag

Send a "Get In Context" message to the successor

No

Reaction to "Receive"

Start

Did the resource come through an Assembly Base Link?

End Processing "Receive"

No

Include the resource received in the set of resources that will be used to create the compound resource

Yes

No

Does the Assembler already have an Assembly Base?

Get the components of the received compound resource and put them in front of the other resources received

Yes

Stop Simulation: Runtime Error

No

Cursor the compound resource received

Yes

Copyright © 1996 Julio C. Martínez
Flowchart 19 - Assembler Reactions to Stop Receiving and Get Out of Context
Flowchart 20 - Consolidator Reactions to Messages
Appendix C
Terminology

This appendix lists terms commonly used throughout the manuscript and their meaning.

<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACD</td>
<td>Activity Cycle Diagram.</td>
</tr>
<tr>
<td>Action</td>
<td>The application of action arguments to an action target.</td>
</tr>
<tr>
<td>Action Arguments</td>
<td>The arguments being applied to an action target in an action.</td>
</tr>
<tr>
<td>Action Control Statement</td>
<td>Control statements whose purpose is to perform an action as soon as they are encountered during the processing of a simulation input file.</td>
</tr>
<tr>
<td>Action Definition</td>
<td>A statement that defines an action that will be performed at the occurrence of action events.</td>
</tr>
<tr>
<td>Event</td>
<td>Specific points within the logic of a simulation where actions can be performed.</td>
</tr>
<tr>
<td>Action Scanning</td>
<td>Activity Scanning.</td>
</tr>
<tr>
<td>Action Target</td>
<td>The object to which action arguments are applied in an action.</td>
</tr>
<tr>
<td>Action Verb</td>
<td>The ASSIGN, COLLECT, or PRINT keywords.</td>
</tr>
<tr>
<td>CAP</td>
<td>Clock Advance Phase. Changing the value of SimTime to the earliest of the events listed in the FEL and proceeding to process the FEL.</td>
</tr>
<tr>
<td>CIP</td>
<td>Combi Instantiation Phase. Attempting to instantiate all the Combis in a model until none can start or until the instantiation of a zero-duration Combi inserts a current event in the FEL.</td>
</tr>
<tr>
<td>CIPROS</td>
<td>Knowledge-Based Construction Integrated Project and</td>
</tr>
<tr>
<td>Term</td>
<td>Meaning</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Process Planning Simulation System.</td>
<td></td>
</tr>
<tr>
<td>Control Statement</td>
<td>A statement that has an immediate effect in the simulation model.</td>
</tr>
<tr>
<td>COOPS</td>
<td>Construction Object-Oriented Process Simulation.</td>
</tr>
<tr>
<td>CPM</td>
<td>Critical Path Method.</td>
</tr>
<tr>
<td>CYCLONE</td>
<td>Cyclic Operations Network.</td>
</tr>
<tr>
<td>Element Attribute Statement</td>
<td>A statement that changes the attribute of a modeling element.</td>
</tr>
<tr>
<td>Element Definition Statement</td>
<td>A statement that defines a modeling element such as a node, link, or statistical or programming object.</td>
</tr>
<tr>
<td>ES</td>
<td>Event Scheduling.</td>
</tr>
<tr>
<td>False Semaphore</td>
<td>When the simulation stops because the FEL is empty and no Combi can start. It is different from “Lack of Resources” in that a false Semaphore is stopping Stroboscope from attempting the instantiation of at least one Combi which could otherwise start.</td>
</tr>
<tr>
<td>FEL</td>
<td>Future Events List. A list of all Activity instances that have not been terminated. The list is sorted by Activity Instance end-time.</td>
</tr>
<tr>
<td>GSP</td>
<td>The General Simulation Program.</td>
</tr>
<tr>
<td>Processing the FEL</td>
<td>The process of removing current events from the FEL by terminating all Activity instances whose End-Time matches SimTime.</td>
</tr>
<tr>
<td>HOCUS</td>
<td>Hand Or Computer Universal Simulator</td>
</tr>
<tr>
<td>Lack of Resources</td>
<td>When the simulation stops because no Combi can start and the FEL is empty.</td>
</tr>
<tr>
<td>MCS</td>
<td>Maybe Can Start. A per Combi flag used to optimize the CIP.</td>
</tr>
<tr>
<td>PDL</td>
<td>The RESQUE Process Description Language.</td>
</tr>
<tr>
<td>Term</td>
<td>Meaning</td>
</tr>
<tr>
<td>------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>PERT</td>
<td>Project Evaluation and Review Technique.</td>
</tr>
<tr>
<td>PI</td>
<td>Process Interaction.</td>
</tr>
</tbody>
</table>
Appendix D
Comparison to Other Construction Simulation Tools

This appendix compares Stroboscope to simulation systems specifically designed for the modeling of construction processes. This appendix also shows how Stroboscope can address selected project level problems, and in those terms compares it to tools specifically designed for those problems.

Process Level Tools

All construction process simulation tools are based on Activity Cycle Diagrams (ACDs) and on the Activity Scanning (AS) simulation strategy. As a consequence they are similar at an abstract level.

Stroboscope differs from all other tools in that it is a simulation programming language and not a simulator. The following definitions and observations about simulation languages and simulators, from (Law and Kelton 1992), are applicable to construction systems and worth citing:

“A simulation language is a computer package that is general in nature but may have special features for certain types of applications.” …
“A model is developed in a simulation language by writing a program using the language’s modeling constructs. The major strength of most languages is their ability to model almost any kind of system, regardless of the system’s operating procedures or control logic.”

“A simulator is a computer package that allows one to simulate a system contained in a specific class of systems with little or no programming.” … “The major drawback of many simulators is that they are limited to modeling only those system configurations allowed by their standard features.”

In addition, construction process simulation systems differ in their underlying philosophy, modeling power, and ease of use.
Limitations Exhibited by All Other Construction Process Simulation Tools

The following issues are very common in construction operations and cannot be modeled, even with infinitely complex networks, by CYCLONE, RESQUE, COOPS or CIPROS. These issues can be modeled very easily by Stroboscope.

1. Uncertainty in the amount of resources consumed and produced. For example, the amount of earth scraped by a scraper is uncertain and must be accounted for accurately as it moves through a network. The amount scraped must be hauled and eventually dumped. Other examples include the amount of rock fragments that result from a dynamite blast and the amount of muck that must be removed in the construction of a tunnel.

2. Processes containing operations with non-stationary duration. For example, the hauling time in the construction of a road depends, among other factors, on haul distance. Since haul distance varies continuously, the parameters for the time distribution are a function of how much earth has been moved. Another example is an operation that is sensitive to the number of times it has been performed in the same process (learning curves).

3. Processes that depend on properties of non-homogenous sets of similar resources. For example, the hauling time for a flatbed containing several different types of steel shapes depends on the weight of the flatbed and on weight of the steel shapes. In order to determine the gross weight, the weight of the steel shapes must be summed up. This cannot be done when only the attributes of the “set header” are accessible (RESQUE and CIPROS).

4. Processes containing operations that are not activated unless complex resource requirements are met. (I.e, that depend on multiple properties of different types of resources in different locations). For example, loading a flatbed with steel shapes only if any of the flatbeds available (characterized by length and payload) can be loaded with steel shapes (must be shorter than the flatbed) to at least 80% of payload, by any of the cranes available (the crane must be able to lift the steel shape).
The following capabilities, not available in either CYCLONE, RESQUE, COOPS, or CIPROS, are either required, or facilitate the solution of a wide variety of problems.

1. The ability to assign values to the properties of resources at simulation runtime.

2. The ability to work with, and produce, derived quantities such as cost.

3. The ability to collect statistics on any aspect of the simulated process.

4. The ability to access resource properties aggregated over several resources.

5. The ability to produce formatted output while the simulation is running for use in animation and graphing programs.

6. The ability to perform multiple replications according to a wide variety of methods.

7. The ability to implement variance reduction techniques such as common random numbers and antithetic variates.

8. The ability to call functions written in conventional programming languages such as C, C++, Pascal, and FORTRAN.

The following sections describe and compare each of the existent construction simulation systems to Stroboscope.

**CYCLONE**

Cyclic Operations Network (CYCLONE) (Halpin & Woodhead 1976) is described in some detail in section 2.1.3.3.

A CYCLONE model is represented entirely on a network; there are no model details that do not appear in the graphic representation. The network on Figure 49 shows a CYCLONE model that includes all possible modeling elements along with all the possible parameters for the elements. There is nothing about the model that is not shown, and nothing more that can be specified (other than altering the network itself). All the
modeling element parameters are numerical (i.e., no variables, operators or function calls), except for the names of the probability distributions used for Activity durations.

CYCLONE resources are similar to Stroboscope generic resources, except that they have no type, are limited to integer amounts, and always flow one by one (there is no DrawAmount, DrawUntil, or ReleaseAmount). CYCLONE Queues have a parameter called “GENerate” that multiplies incoming resources by the specified integer value. CYCLONE Consolidators finish when their content reaches the value specified by the “CONsolidate” parameter.

“GENerates” are used in conjunction with “CONsolidates” to multiply or divide the number of resources in certain parts of a path; this is used to model fundamental issues such as resource unit matching. The “GEN 4” in Queue 70 (Figure 49), for example, multiplies each entering resource by four. The resource that enters can be interpreted as a “hauler”, and the resources that result after the multiplication can be interpreted as four “quarter haulers”. Each time Combi 10 starts, it obtains one “quarter hauler” and one “loader” to represent the time it takes to load one fourth of a hauler (one loader scoop). The “CON 4” in Consolidator 20 has the effect of dividing each entering resource by four. Every time four “quarter haulers” enter, they are converted into one “hauler” that immediately enters Normal 30 to represent the hauler traveling loaded.
Stroboscope Queues do not need a “GENerate” attribute. Resources travel in their real amounts, which can be very precisely controlled via link attributes such as DrawUntil, DrawAmount, and ReleaseAmount. Since soil is not modeled explicitly in Figure 49, the ultimate purpose of the “GENerate” and “CONsolidate” is to determine the time it takes to load a hauler by sampling and adding up the duration of “loading a scoop” four times. In Stroboscope, this can be modeled by simply specifying \( \text{Load’s duration} \) with an expression such as ‘\( \text{Pert}[1.5,2,2.1]+\text{Pert}[1.5,2,2.1]+\text{Pert}[1.5,2,2.1]+\text{Pert}[1.5,2,2.1] \)’. (More generally, it is possible to load a hauler until it is “full”, using random scoop sizes and durations.)

CYCLONE simulations stop when the first of two possible conditions are reached. The first is when the simulation clock reaches the simulation time limit. The second is when a special node marked as the “Counter” has been activated a specified number of times (500 times for node 60 in Figure 49). Stroboscope does not need a “Counter” node since the condition for a Stroboscope simulation to stop can be defined by any arbitrary expression.

Aside from the “GENerate” option at Queues and the “Counter” node, a Stroboscope network is a superset of a CYCLONE network. Due to Stroboscope’s expressive power, however, a Stroboscope network is usually substantially smaller and simpler than the equivalent CYCLONE network. This is especially true when similar resources with different characteristics are used for the same operations (e.g., loading with loaders and haulers of different models), and when Activity startup conditions follow complex logic.

CYCLONE resources have no type or properties, and cannot be distinguished between each other. As a consequence, similar resources must follow different paths. Figure 50 shows the CYCLONE network necessary to model an earth-moving operation similar to that of Figure 49, with the soil modeled explicitly and two different hauler sizes.

The top part of the network in Figure 50 is for 3 CY haulers while the bottom part is for 5 CY haulers. All loaders are the same size (1 CY). The loader size is convenient because it is the largest common factor of 3 and 5, and because it allows the soil to be expressed in its actual units.
The CYCLONE network is set to give priority to the big haulers because “get 5 CY hauler” is numbered lower than “get 3 CY hauler”. It is not possible to model FIFO service order using CYCLONE. One type of hauler must have priority over the other (this priority cannot change throughout the simulation).

The complexity and size of CYCLONE networks for cases with several types of haulers and several types of loaders increases exponentially. Thus, realistic models of construction operations can be so large and complex that they become unmanageable.

The Stroboscope network required to model an earth-moving operation with any number of different hauler and loader types is no different than the network required to model an operation with one type of hauler and one type of loader. This is because Stroboscope’s characterized resources have properties that can be used to establish

Figure 50 - CYCLONE Earth-moving Operation With 2 Types of Haulers
complex resource selection and utilization strategies, and because operations can be sensitive to these properties.

CYCLONE networks must model complex logic by pure “network force”. Figure 51 shows a simple network fragment where two Activities are prevented from starting during non-working hours.

![Figure 51 - CYCLONE Network Fragment With Clock](image)

Only the top two rows of nodes in Figure 51 are relevant to the operation being modeled. The rest of the model is composed of a simplified mechanism that essentially takes hold of the resources required to perform Activities 3 and 4 during non-work hours, and thus prevents them from starting. (In an actual network the “X”, “Y”, and “X+Y” must be substituted with actual numbers.) The CYCLONE networks required to model cases where more than a few Activities must be prevented from starting during non-work
hours are extremely large and complex. The situation is worse when different Activities have different work periods.

The clock mechanism described above is an extremely simple case. It is common for construction operations to be subject to very complex startup conditions in addition to those due to the availability of resources. Stroboscope’s Semaphores and the ability to access the state of the simulation make the modeling of such situations extremely simple. A Stroboscope model equivalent to that of Figure 51 would not contain the sections labeled “Actual Break” and “Commands Wait For Activities to Finish”. Only the section labeled “Alarm Clock” and the top two rows of nodes would be needed. Any Combi would be prevented from starting during non-work hours by simply setting its Semaphore to the expression ‘ResourcesActive.CurInst’.

RESQUE

RESource based QUEuing network simulation (RESQUE) (Chang 1986) is described in some detail in section 2.1.3.4.

RESQUE is significantly more powerful than CYCLONE because it can recognize the state of the simulation and that similar resources can be different. It is similar to Stroboscope in that the entire model is not represented exclusively by a network. The network is only a high level representation of the actual process. In RESQUE, the details of the process are defined via its own Process Description Language (PDL).

Stroboscope and RESQUE are superficially similar in that they address most of the same issues. They both recognize that resources of the same type may have different properties, that the state of the simulation is an important factor in determining whether an Activity should or should not be performed, and that resources are often grouped and for some time thereafter act as a single resource.

Despite the remarkable number of superficial similarities, Stroboscope and RESQUE are very different in terms of design, capabilities, and feel. The following essential differences are the root of numerous other differences:
• RESQUE resources are discrete entities that differ only in their “type” (analogous to a Stroboscope characterized resource type) and their “attribute” (a single integer identifier analogous to a Stroboscope SubType). This is in contrast to Stroboscope, where resources can be bulk (i.e., can exist in fractional amounts) or discrete. Stroboscope’s discrete resources can be characterized with an unlimited number of properties of four kinds: system-defined and maintained properties, properties common among all resources belonging to the same SubType, properties where information can be stored or retrieved, and properties expressed as functions of other properties and the state of the simulation.

• RESQUE’s access to the state of the simulation is limited to the current and total count of resources at Queues and to the total number of Activity instances. These values can only be compared to a fixed number, and used as a pre-condition for the activation of a Combi (e.g., start Combi 16 only if Combi 28 has been activated less than four times). This is in contrast to Stroboscope, where numerous aspects of the state of the simulation are available for use in any expression.

• In RESQUE, the resource assembly/disassembly mechanism is implemented as part of the Activity functionality, where some resources can be held by a node for several cycles (activations) before they are assembled. As a consequence, RESQUE Activities that assemble/disassemble are limited to one simultaneous instance. In addition, RESQUE models require assembly/disassembly to represent even very simple operations, such as loading a hauler with several scoops of soil. This is in contrast to Stroboscope, where resource assembly and disassembly are handled by separate nodes in a graphically intuitive manner.

• The only non-numeric identifiers in the RESQUE PDL are the names of the resource types and the keywords for the several PDL statements. As a consequence, a RESQUE PDL file is very difficult to follow, and contains a large sequence of numbers separated by commas. This is in contrast to a Stroboscope file, which consists mainly of user-defined identifiers, and meaningful statement and function names.
Unlike CYCLONE, it is not necessary to complicate a RESQUE network to model similar resources of different types. The differences between the resources can be handled with RESQUE’s PDL. The appropriate duration for loading a hauler, for example, is a function of the hauler and the loader. Figure 52 shows a RESQUE network fragment from (Chang 1986) that illustrates how the duration of the loading Activity is specified in RESQUE PDL.

The “RESLIST” section indicates that Queue number 2 is initialized with 1 loader with attribute value of 1 and one loader with attribute value of 2. It also indicates that Queue number 4 is initialized with 2 haulers with attribute value of 30 and 2 haulers with attribute value of 50.

Assuming that an instance of Combi number 6 starts with one hauler with attribute value of 50 and one loader with attribute value of 2, RESQUE would determine its duration as follows.
The DUR statement for Combi number 6 is defined with -10 and thus indicates that entry number 10 in the “RDVLIST” should be used. Entry 10 in the RDVLIST indicates that if the hauler received from Queue number 4 has an attribute value of 30, then entry number 12 in RDVLIST should be used. Since this is not the case, the next pair of values is checked which indicate that if the hauler has an attribute value of 50, entry number 14 should be used. Entry number 14 says that if the loader received from Queue number 2 has an attribute value of 1, the duration should be determined from entry number 6 in the “DURLIST” section. Since this is not the case, the next pair of values is checked which indicate that if the loader has an attribute value of 2, the duration should be determined from entry number 8 in DURLIST. Entry number 8 in DURLIST says that the duration is uniformly distributed (2) between 21 and 26 time units.

In RESQUE there must be one entry in the DURLIST section for each possible combination of attribute values of the resources involved in an operation. The attribute values themselves cannot be used as parameters to a probability distribution. In addition, there must be a series of chained conditionals in the RDVLIST section that leads to the appropriate entry in DURLIST. The number of entries in both sections grows exponentially as more types of resources with a wider variety of attributes are used.

The example above assumes that a probability distribution is available for each possible combination of resources. If the data available represents the duration of a scoop, the method described above does not work. Instead, the RESQUE model must obtain a hauler and, based on chaining rules similar to those used in the example above, determine the number of “holding” cycles (e.g., the number of scoops required). The Combi will only release the hauler after the Activity has been activated the specified number of times. The duration for each scoop is then determined based on the attribute of the loader (which is acquired and released each time, and is not guaranteed to be the same for each scoop or to be available immediately after it has loaded another scoop). This method does not allow more than one instance of the loading Combi to occur at the same time.

The Stroboscope network fragment in Figure 53 is analogous to the RESQUE fragment in Figure 52. It assumes that the loading duration data is available on a per-scoop basis. In order to model extra types of loaders and haulers, it is only necessary to
define SubTypes for them. **In addition, the model will work with any number of simultaneous Load instances without preempting them, and using the same loader.**

![Stroboscope Network Fragment Analogous to Figure 52](image)

```plaintext
CHARTYPE LoaderBucketSize LoadTimeLow LoadTimeHigh;
SUBTYPE Loader BigLD 2.5 7 9;
SUBTYPE Loader SmlLD 1.5 8.5 11;

CHARTYPE Hauler Size;
SUBTYPE Hauler BigHL 15;
SUBTYPE Hauler SmlHL 12;

GENTYPE Soil;

DRAWAMT SL1 BucketSize;
DRAWDUR SL1 Uniform[LoadTimeLow,LoadTimeHigh];
DURATION Load SL1.SumDrawDur;
```

**Figure 53 - Stroboscope Network Fragment Analogous to Figure 52**

Since RESQUE resources do not have real properties, it cannot model situations that depend of resource properties acquired during simulation runtime, such as the amount of fuel left in a loader’s tank, or the amount of dirt carried in a hauler. In addition, situations that require the aggregation of a property over several resources, such the total weight of the steel shapes carried in a flatbed, cannot be modeled.

The description above was based on determining the duration of an Activity that can use similar resources of different types. RESQUE models face many of the same issues that must be addressed by Stroboscope, such as resource selection, routing, assembly and disassembly. The comparison of the two systems along those lines is of the same nature as the one discussed here. For the sake of brevity, these comparisons will not be made here.
Another issue that merits comparison is how RESQUE uses the state of the simulation as a precondition to activate a Combi, which is analogous to a Stroboscope Semaphore. In RESQUE, the condition can only access four aspects of the state of the simulation.

1. The current value of the simulation clock (CT), which is equivalent to SimTime in Stroboscope.

2. The current number of resources at a Queue (CC), which is equivalent to QueueName.CurCount in Stroboscope.

3. The total number of resources that have entered a Queue (TC), which is equivalent to QueueName.TotCount in Stroboscope.

4. The total number of instantiations of an Activity (TA), which is equivalent to ActivityName.TotInst in Stroboscope.

Any of the above mentioned aspects of the state of the simulation can only be used to create a “CONDLIST” entry in RESQUE. A CONDLIST entry consists of three parts. The first is one of the values listed above. The second is one of the following relational operators: LT (<), LE (<=), GT (>), GE (>=), EQ (==) and NE (!=). The third is a number. An entry in the CONDLIST section, for example, can be “2, CC, 10, GE, 5”, which is interpreted as “condition number 2 is true if the current number of resources in Queue number 10 is greater than or equal to 5.

A condition in the CONDLIST can then be used as a precondition to activate a Combi. Because the conditions are restricted as described above, a RESQUE model implementing complex logic (or simple logic such as breaks during non-work hours) is only a little bit simpler than the equivalent CYCLONE model, and substantially more complex than the equivalent Stroboscope model. RESQUE, for example, cannot simplify the CYCLONE model of Figure 51 because there is no access to the current number of instances of an Activity.

In addition, RESQUE cannot use the state of the simulation for the determination of Activity durations or any other issues. RESQUE, for example, cannot model learning curves or non-stationary travel times.
The above discussion on RESQUE makes it obvious that it is a significant improvement over CYCLONE. At the same time, it shows some of its modeling limitations, which do not exist in Stroboscope.

**COOPS**

Construction Object-Oriented Process Simulation System (COOPS) (Liu 1991) was discussed in some detail in section 2.1.3.5.

COOPS’ design objectives were ease of use and the application of object-oriented technology and interactive graphics to the design of construction simulation systems. Additionally, its design incorporates facilities to “alleviate some of the existing difficulties in construction simulation [CYCLONE], such as break time modeling, resource tracking, and difficulty in modeling different resource units” (Liu 1991).

From the perspective of the user, COOPS’ graphical user interface is similar to Stroboscope’s. The user interfaces, however, are implemented differently. In COOPS, the graphical objects are the same as the modeling objects. Thus, for example, the rectangle that represents a Normal Activity is the same object that reacts to messages as the simulation runs. The objects in Stroboscope’s graphical user interface are separate from those that do simulation. They are based on a client-server architecture. In fact, the graphical objects are not essential.

COOPS has two classes of resources, generic and specific. Generic resources are similar to the resources in CYCLONE. Specific resources are tracked individually (i.e., the system keeps separate statistics for each such resource) but cannot be characterized. As a consequence, for example, a COOPS earth-moving model with two types of haulers must use different sets of Activities and Queues for each type.

COOPS has Link Resource Requirements (LRR) that are similar to Stroboscope’s Enough, DrawAmount, and ReleaseAmt. As a result, a COOPS network does not require “GENerates” and “CONsolidates”.

COOPS uses calendars, which are cyclic patterns of work time - idle time pairs. Calendars can be attached to Queues and to specific resources. An Activity will not start unless all its preceding Queues are currently in working time and all the specific
resources required for the Activity are also in working time. If a COOPS Activity has started, and resources are not available at particular times during the duration, the Activity ending time is adjusted according to common working time for all required resources, skipping all the break time on the resource’s calendars.

Thus, COOPS networks do not need to be complicated in order to consider break times. In terms of not allowing Activities to start, Stroboscope Semaphores accomplish the same task. In terms of extending an Activity’s duration, Stroboscope’s programmability must be used to specify an appropriate duration expression for the Activity.

A potential problem with COOPS’ approach to extending Activity durations is that it cannot be turned off. As a consequence, COOPS Activities always assume that interruptions in the work do not affect the total work time required to complete the task. For example, COOPS assumes that a 20 minute task can be accomplished by working 4 minutes at the end of the day, and 16 minutes at the beginning of the next day.

COOPS calendars are limited to modeling break times. They cannot be used to control Activity instantiations due to complex startup logic that depends on the state of the simulation, and which do not follow a predetermined on-off pattern based on time.

**CIPROS**

Knowledge-Based Construction Integrated Project and Process Planning Simulation System (CIPROS) (Odeh 1992), was discussed in some detail in section 2.1.3.6.

CIPROS is both a process level and project level planner. This section compares Stroboscope to CIPROS in terms of process level capabilities. A comparison at the project level is made in a later section of this appendix.

CIPROS models are like Stroboscope models in that the network only specifies high-level aspects of a process. The information that complements a CIPROS model is not contained in a text file. Instead, it is saved in binary form and can only be examined by opening appropriate dialog boxes via CIPROS’ interface. This is unlike RESQUE and Stroboscope, where the complementary information is contained in a text file following
the system’s syntax. As a consequence, it is very difficult to communicate a complete CIPROS model in printed form.

CIPROS has discrete as well as bulk resources. The bulk resources are similar to Stroboscope’s generic resources. They can exist in fractional amounts and represent construction materials and other bulk items. The discrete resources can be represented hierarchically and can have any number of real properties. They are similar to characterized resources in Stroboscope, except that all the properties must be defined along with their values before simulation runtime (i.e., they only support SubType properties). It is not possible to attach information to a resource while the simulation is running, or to define properties as a function of other properties. In addition, CIPROS does not maintain properties such as ResNum, TimeIn, and BirthTime, that are specific to individual resources.

In most respects, CIPROS’ process level architecture is similar to RESQUE’s. The CIPROS literature (Odeh 1992) enumerates several of CIPROS’ key advantages over RESQUE. Most of these advantages are matters of degree, such as being able to order the resources within a Queue based on any property and not just the “attribute” of the resource. A similar statement could be made about Stroboscope as compared to CIPROS: resources in Stroboscope Queues can be ordered according to any dynamic function that relates any number of properties of the resource with any aspect of the state of the simulation or any of the properties of resources located anywhere in the system. A similar statement can be made about every other aspect of a simulation model. Such issues are too numerous. They were not listed when comparing RESQUE and Stroboscope, and are not listed in this section either.

While CIPROS improves substantially over RESQUE in terms of resource characterization, the improvements are at a level that does not make a comparison between Stroboscope and CIPROS any different than a comparison between Stroboscope and RESQUE. For example, CIPROS cannot use the property of a resource as an argument to a probability sampling function. Instead, conditional comparisons must be chained and eventually lead to a probability sampling function in terms of constant numbers.
The method used by CIPROS to determine the duration of a hypothetical “haul” operation should make this clear. Assume one steel shape is hauled on a flatbed. Hauling time is normally distributed with a mean and standard deviation that are proportional to the combined weight of the flatbed and the steel shape, as determined by the formula “Normal[TotalWeight*5.2, TotalWeight*0.52]”. The weight of the flatbed is given by its GrossWgt property, and the weight of the steel shape is given by its Weight property.

Possible flatbed weights are 7.5 and 9.0. Possible steel shape weights are 1.2, 1.7 and 2.8. The duration for “haul” would be set to “ACD[1]”, and determined by inspecting Table 24.

Table 24 - CIPROS Attribute Conditional Dependency List

<table>
<thead>
<tr>
<th>ACD</th>
<th>Name</th>
<th>Attribute</th>
<th>Rank</th>
<th>Operator</th>
<th>Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>flatbed</td>
<td>GrossWgt</td>
<td>0</td>
<td>=</td>
<td>7.5</td>
<td>ACD[2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Default</td>
<td></td>
<td>ACD[3]</td>
</tr>
<tr>
<td>2</td>
<td>shape</td>
<td>Weight</td>
<td>0</td>
<td>=</td>
<td>1.2</td>
<td>Normal(45.24, 4.524)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>=</td>
<td>1.7</td>
<td>Normal(47.8, 4.784)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Default</td>
<td></td>
<td>Normal(53.56, 5.356)</td>
</tr>
<tr>
<td>3</td>
<td>shape</td>
<td>Weight</td>
<td>0</td>
<td>=</td>
<td>1.2</td>
<td>Normal(53.04, 5.304)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>=</td>
<td>1.7</td>
<td>Normal(55.64, 5.564)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Default</td>
<td></td>
<td>Normal(61.36, 6.136)</td>
</tr>
</tbody>
</table>

ACD[1] indicates that if the GrossWgt of the flatbed is equal to 7.5, then ACD[2] should be used, otherwise ACD[3] should be used. ACD[2] says that if the Weight of the shape is equal to 1.2, then the duration is sampled from a Normal distribution with mean 45.24 and standard deviation 4.524. The remaining ACD entries are similar.

The CIPROS approach requires a specific entry and search path for each possible combination of attributes (resource properties). If the flatbed can carry several shapes with different weights, it is not possible to establish an accurate distribution. This is because only the attributes of the “set header” and the shape count are available.

CIPROS does not incorporate the limited access to the state of the simulation provided by RESQUE, nor does it allow conditional activation of Combis. As a
consequence, a CIPROS model incorporating break times or complex logic is as complicated as the corresponding CYCLONE model.

**Relationship to Project Level Construction Simulation Tools**

Stroboscope was designed as a process level simulation tool. A process is a lower level representation (more detailed) of work than a project. The smallest unit of work in a project is generally an activity, which can in turn be represented as a process or the major part of a process.

Stroboscope’s process level orientation, combined with its representation power, enable it to address interesting project level issues. As examples of Stroboscope’s applicability to project level issues, the next couple of sections provide some guidelines on how to use Stroboscope to emulate MUD (Carr 1971) / DYNASTRAT (Morua Padilla 1986) and CIPROS (as a project level tool).

In general, using Stroboscope for project level analysis carries the typical tradeoffs (as compared to project level tools): greater flexibility at the expense of more elaborate model development. Project level model development can be substantially simplified with Stroboscope Add-Ons, as illustrated by the probabilistic CPM Add-On of Chapter 18.

**MUD and DYNASTRAT**

MUD (Carr 1971) is a project-level simulator that considers the impact of several variables on the daily progress of activities. These variables can be calendar-dependent (but mutually independent) (DECAD) such as temperature, precipitation, and wind; as well as calendar independent (INCAD) such as supervision. DYNASTRAT (Morua-Padilla 1986) builds upon MUD by incorporating resource allocation strategies and cost calculations.

MUD determines new values for its INCAD and DECAD variables for every project day. These values are common for all the CPM Activities in a model. When an Activity starts, MUD determines its “standard duration” by sampling from a probability distribution. Each Activity has its own sensitivity to the INCAD and DECAD variables.
The sensitivities and the corresponding variables are used in a formula to obtain a daily correction factor. The correction factor represents the Activity’s rate of progress relative to its standard rate of progress. For example, a correction factor of 0.50 on the first day of a 10 day Activity, means that at the end of the day 9.5 days of work remain to be performed. An Activity is considered complete when no work remains to be performed. Its duration is calculated as the number of days between its start and completion date, which is not necessarily the same as the sampled “standard duration”. In this manner, MUD appropriately considers the mutual dependency of Activities on common factors.

DYNASTRAT enhances MUD by assuming limited resource availability and incorporating resource allocation strategies. DYNASTRAT also determines cost. A resource allocation strategy is a series of rules on how to allocate the scarce resources to the different Activities based on available information, on a daily basis. This may include interrupting Activities that are underway in order to allocate resources to other Activities deemed more critical. The resources allocated to an Activity affect its daily progress in conjunction with the INCAD and DECAD variables. Different strategies can be simulated to determine the most convenient one.

The CPM Add-On described in chapter 18 can be used to achieve a simplified version of MUD’s functionality. Separate network fragments can run in parallel to the CPM network with the purpose of producing variables that represent the various INCAD and DECAD. This is similar to the use of clocks as separate network fragments for the production of variables such as WorkTime or WeekEnd (see section 7.2). The factors can then be used in the expressions that define the CPM Activity durations.

The above approach works well if factors existing at the beginning of the CPM Activity are assumed to remain constant throughout its entire duration. In order to consider the factors on a day to day basis, a different Add-On needs to be created. The new Add-On would represent the actual CPM Activity with a Consolidator, a Queue, a Combi, and a Fork (instead of with a Consolidator and a Normal). The Combi would have a fixed duration of one day. When the Combi finishes it would update the percent complete. The Fork would then activate the successors of the CPM Activity only if it is 100% complete. Otherwise the Fork would reactivate the Queue so that the Combi can start again to represent more work done on the same CPM Activity.
The new Add-On could be extended further to provide the functionality found in DYNASTRAT. Global Queues could be used to represent the pool of resources available to the project. The Combis would draw resources at the beginning of the day and release them at the end of the day. The Combi priorities and resource drawing capabilities of Stroboscope could be used to model complex resource allocation strategies.

**CIPROS (Project Level)**

CIPROS is also a project-level planning tool where each project-level Activity is represented by a process network. The different processes in each network can share resources and thus model resource interactions between project-level Activities.

CIPROS is designed to take advantage of modularity. In this respect, the processes in a project-level CIPROS model have many modeling elements with local scope. In addition, CIPROS incorporates a knowledge-based approach to the organization of the processes that makes it convenient to try different project plans that use different methods to accomplish the same project-level Activity.

In order to achieve the same effect in Stroboscope, the entire project must be modeled as a single large process. The automatic code generation techniques described in chapter 15 can be used to generate such models automatically, but still require significant amount of work. The approach would be similar to that used in section 16.4.1.3, where automatic code generation was used to build probabilistic CPM models from generic data, except that an entire sub-network would be used to represent a CPM Activity (instead of a Normal).

**General Approach to Project-Level Modeling**

Stroboscope’s lower-level process-orientation and representation power enable it to emulate almost any higher-level project-oriented model. Undertaking such a task usually requires significantly more effort than using a tool specifically designed for that purpose. The advantage is that the resulting Stroboscope model is more flexible and powerful. For example, once a project-level model has been developed (CIPROS style) it is possible to make its operations sensitive to INCAD and DECAD variables quite easily.
A project-level construction planning tool can be designed and implemented as a Stroboscope Add-On. In this case, the Add-On retains Stroboscope’s power of representation, but model development becomes at least as easy as using a dedicated tool (depending on the Add-On’s design). The simple probabilistic CPM Add-On presented in chapter 18 is an example.

While developing an Add-On to perform the combined tasks of CIPROS and DYNASTRAT is not a trivial endeavor, it is orders of magnitude easier than writing a dedicated program using a standard programming language. In this sense, Stroboscope and its Add-On interface specification are an attractive development option.
Appendix E

Trace Through a Short Simulation Run

This appendix traces through a short simulation based on the network of Figure 7 on page 27.

Assume that at the beginning of the operation described in Figure 7, `LoadersWait` contains 1 loader and `HaulersWait` contains 2 haulers. The unit of measure to use for the soil is the capacity of a hauler — a hauler-load (in this example all haulers are interchangeable and indistinguishable, and thus have the same capacity). The initial content of `SoilToMove` is 3 hauler-loads. The duration in minutes of the first three instances of the different Activities in this operation are shown in Table 25.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Load (min)</th>
<th>Haul (min)</th>
<th>Dump (min)</th>
<th>Return (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.8032</td>
<td>4.8781</td>
<td>0.5</td>
<td>3.5892</td>
</tr>
<tr>
<td>1</td>
<td>1.1888</td>
<td>2.5606</td>
<td>0.5</td>
<td>2.1305</td>
</tr>
<tr>
<td>2</td>
<td>1.8740</td>
<td>2.5136</td>
<td>0.5</td>
<td>4.1909</td>
</tr>
</tbody>
</table>

*Table 25 - Simple Haul Data*

Note from Table 25 that every instance of `Load` has a different duration. The same is true of `Haul` and `Return` instances.

**First Combi Instantiation Phase**

The entry point into the simulation loop is the first Combi Instantiation Phase (CIP). At this point, the state of the simulation is summarized in Figure 54.
Simulation Time : 0  
Phase : Combi Instantiation Phase #1  
Step : About to start the CIP

Contents of Queues:

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaulersWait</td>
<td>2 haulers</td>
</tr>
<tr>
<td>LoadersWait</td>
<td>1 loader</td>
</tr>
<tr>
<td>MovedSoil</td>
<td>** empty **</td>
</tr>
<tr>
<td>SoilToMove</td>
<td>3 hauler-loads of soil</td>
</tr>
</tbody>
</table>

Future Events List:

<table>
<thead>
<tr>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>(The FEL is empty)</td>
</tr>
</tbody>
</table>

*Figure 54 - About to Start CIP # 1*

In the CIP, the Combis in the model are scanned one by one to see if the conditions necessary for them to start are satisfied. Only one Combi exists in this example (*Load*), so it is the first and only one to be checked. The conditions necessary for *Load* to start are that:

1. *LoadersWait* contains enough loaders to support an instance of *Load*. This example uses the defaults; so as long as *LoadersWait* is not empty, it contains enough loaders.

2. *HaulersWait* contains enough haulers to support an instance of *Load*. This example uses the defaults; so as long as *HaulersWait* is not empty, it contains enough haulers.

3. *SoilToMove* contains enough soil to support an instance of *Load*. This example uses the defaults; so as long as *SoilToMove* is not empty, it contains enough soil.

These conditions do exist the very first time Stroboscope scans *Load* — *LoadersWait* contains 1 loader, *HaulersWait* contains 2 haulers, and *SoilToMove* contains 3 hauler-loads of soil.

Since *Load* can start, it will start. When it starts it removes 1 hauler from *HaulersWait*, 1 loader from *LoadersWait*, and 1 unit of soil from *SoilToMove*. It puts all three resources into instance number 0 of *Load*, which has a duration of 0.8032 minutes (from Table 25). Since the duration of *Load*(0) is known, its termination will effectively happen at minute 0.8032 of the simulation.
The termination of $Load(0)$ is the first event to be placed in the FEL, which now contains one entry. Important aspects of the state of the simulation just after the instantiation of $Load(0)$ are shown in Figure 55.

<table>
<thead>
<tr>
<th>Contents of Queues:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Queue</strong></td>
</tr>
<tr>
<td>HaulersWait</td>
</tr>
<tr>
<td>LoadersWait</td>
</tr>
<tr>
<td>MovedSoil</td>
</tr>
<tr>
<td>SoilToMove</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Future Events List:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instance</strong></td>
</tr>
<tr>
<td>Load(0)</td>
</tr>
</tbody>
</table>

*Figure 55 - CIP #1 After the Instantiation of Load(0)*

Because the first scan of $Load$ was successful it needs to be re-scanned — it is possible that $Load$ can start a second time. At the time of the second attempt, however, the conditions are no longer met. $LoadersWait$ is empty because $Load(0)$ removed the loader that was there. It is no longer possible to start any Combi. This marks the end of the first CIP.

**First Clock Advance Phase**

Given the current state of affairs nothing else can happen at time 0. The FEL contains only future events and no Combi can start. Thus, Stroboscope enters the Clock Advance Phase (CAP).

The earliest future event (the termination of $Load(0)$) occurs at minute 0.8032. Thus, Stroboscope advances the simulation clock to that time. Due to the new value of the simulation clock, the FEL contains events that will not happen in the future — events that are bound to happen at the current time. Stroboscope needs to process the FEL immediately by terminating $Load(0)$.

**First Activity Instance Termination**

Figure 56 shows the state of the simulation before the termination of $Load(0)$, but after the clock is advanced (notice that the FEL is ripe as shown by the shaded area):
Simulation Time: **0.8032**

**Phase:** Clock Advance Phase #1  
**Step:** About to terminate Load(0)

**Contents of Queues:**

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaulersWait</td>
<td>1 hauler</td>
</tr>
<tr>
<td>LoadersWait</td>
<td>** empty **</td>
</tr>
<tr>
<td>MovedSoil</td>
<td>** empty **</td>
</tr>
<tr>
<td>SoilToMove</td>
<td>2 hauler-loads of soil</td>
</tr>
</tbody>
</table>

**Future Events List:**

<table>
<thead>
<tr>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load(0)</td>
<td>0</td>
<td>0.8032</td>
</tr>
</tbody>
</table>

---

**Figure 56 - CAP #1 Just Before the Termination of Load(0)**

As soon as the termination of Load(0) begins, Stroboscope removes Load(0) from the FEL. Before Load(0) releases its resources, Stroboscope determines the successors to Load. Note that although 3 links leave Load, there are only 2 successors — Haul and LoadersWait. The successors are notified that a Load instance is about to terminate. LoadersWait does nothing. Haul gets ready to receive the resources that it will later package into Haul(0).

The termination of Load(0) releases a loader to LoadersWait, and a hauler and one unit of soil to Haul. After Load(0) releases all its resources, Haul packages the hauler and soil it received into Haul(0) (the first instance of Haul). Haul(0) has a duration of 4.8781 minutes (Table 25). Since it started at time 0.8032, it will be terminated at time 5.6813. The event for the termination of Haul(0) is placed in the FEL (the only event there). At this point, the termination of Load(0) is complete, and the state of the simulation is as shown in Figure 57.
Simulation Time : 0.8032  
Phase : Clock Advance Phase #1  
Step : Just after the termination of Load(0)

Contents of Queues:  

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaulersWait</td>
<td>1 hauler</td>
</tr>
<tr>
<td>LoadersWait</td>
<td>1 loader</td>
</tr>
<tr>
<td>MovedSoil</td>
<td>** empty **</td>
</tr>
<tr>
<td>SoilToMove</td>
<td>2 hauler-loads of soil</td>
</tr>
</tbody>
</table>

Future Events List:  

<table>
<thead>
<tr>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haul(0)</td>
<td>0.8032</td>
<td>5.6813</td>
</tr>
</tbody>
</table>

Figure 57 - CAP #1 After the Termination of Load(0)

Note that although the FEL is not empty, the event it contains will occur in the future. The FEL contains no current events and the first CAP is complete.

Subsequent Cycles in the Simulation Loop

The first cycle in the simulation loop is now complete. The processing of the simulation model proceeds with the second and subsequent cycles. The remaining steps in the processing of the model are summarized below.

Combi Instantiation Phase # 2

Simulation Time : 0.8032  
Phase : Combi Instantiation Phase #2  
Step : About to start the CIP

Contents of Queues:  

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaulersWait</td>
<td>1 hauler</td>
</tr>
<tr>
<td>LoadersWait</td>
<td>1 loader</td>
</tr>
<tr>
<td>MovedSoil</td>
<td>** empty **</td>
</tr>
<tr>
<td>SoilToMove</td>
<td>2 hauler-loads of soil</td>
</tr>
</tbody>
</table>

Future Events List:  

<table>
<thead>
<tr>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haul(0)</td>
<td>0.8032</td>
<td>5.6813</td>
</tr>
</tbody>
</table>

Figure 58 - About to Start CIP #2

Load is scanned. It is possible to start it, so Load(1) is intantiated with a duration of 1.1888. In the process, it removes 1 loader from LoadersWait, 1 hauler from HaulersWait, and 1 hauler-load of soil from SoilToMove. The state of the simulation is now as shown in Figure 59.
Simulation Time : 0.8032  
Phase : Combi Instantiation Phase #2  
Step : Just after the instantiation of Load(1)

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaulersWait</td>
<td>** empty **</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LoadersWait</td>
<td>** empty **</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MovedSoil</td>
<td>** empty **</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SoilToMove</td>
<td>1 hauler-load of soil</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Future Events List:

<table>
<thead>
<tr>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load(1)</td>
<td>0.8032</td>
<td>1.992</td>
</tr>
<tr>
<td>Haul(0)</td>
<td>0.8032</td>
<td>5.6813</td>
</tr>
</tbody>
</table>

Figure 59 - CIP #2 Just After the Instantiation of Load(1)

Stroboscope now scans Load again. Load cannot start because LoadersWait and HaulersWait are empty. Since it is no longer possible to start any Combi, the CIP ends.

Clock Advance Phase #2

The simulation clock is advanced to time 1.992.

Simulation Time : 1.992
Phase : Clock Advance Phase #2
Step : About to terminate Load(1)

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaulersWait</td>
<td>** empty **</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LoadersWait</td>
<td>** empty **</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MovedSoil</td>
<td>** empty **</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SoilToMove</td>
<td>1 hauler-load of soil</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Future Events List:

<table>
<thead>
<tr>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load(1)</td>
<td>0.8032</td>
<td>1.992</td>
</tr>
<tr>
<td>Haul(0)</td>
<td>0.8032</td>
<td>5.6813</td>
</tr>
</tbody>
</table>

Figure 60 - About to Terminate Load(1) in CAP #2

Load(1) is terminated. During the termination, Haul(1) is instantiated with a duration of 2.5606. The hauler and soil released by Load(1) are packaged with Haul(1). The loader returns to LoadersWait.
Simulation Time : 1.992  
Phase : Clock Advance Phase #2  
Step : Just after the termination of Load(1)

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
<th>Future Events List</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Instance</td>
</tr>
<tr>
<td>HaulersWait</td>
<td>** empty **</td>
<td>Haul(1)</td>
</tr>
<tr>
<td>LoadersWait</td>
<td>1 loader</td>
<td>Haul(0)</td>
</tr>
<tr>
<td>MovedSoil</td>
<td>** empty **</td>
<td></td>
</tr>
<tr>
<td>SoilToMove</td>
<td>1 hauler-load of soil</td>
<td></td>
</tr>
</tbody>
</table>

Figure 61 - CAP #2 After the Termination of Load(1)

The FEL contains no current events, so the CAP ends.

**Combi Instantiation Phase #3**

Simulation Time : 1.992  
Phase : Combi Instantiation Phase #3  
Step : About to start the CIP

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
<th>Future Events List</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Instance</td>
</tr>
<tr>
<td>HaulersWait</td>
<td>** empty **</td>
<td>Haul(1)</td>
</tr>
<tr>
<td>LoadersWait</td>
<td>1 loader</td>
<td>Haul(0)</td>
</tr>
<tr>
<td>MovedSoil</td>
<td>** empty **</td>
<td></td>
</tr>
<tr>
<td>SoilToMove</td>
<td>1 hauler-load of soil</td>
<td></td>
</tr>
</tbody>
</table>

Figure 62 - About to Start CIP #3

*Load* is scanned. It cannot start because *HaulersWait* is empty. CIP #3 is over.

**Clock Advance Phase #3**

The simulation clock is advanced to time 4.5526.
Simulation Time : 4.5526
Phase : Clock Advance Phase #3
Step : About to terminate Haul(1)

Contents of Queues:

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaulersWait</td>
<td>** empty **</td>
</tr>
<tr>
<td>LoadersWait</td>
<td>1 loader</td>
</tr>
<tr>
<td>MovedSoil</td>
<td>** empty **</td>
</tr>
<tr>
<td>SoilToMove</td>
<td>1 hauler-load of soil</td>
</tr>
</tbody>
</table>

Future Events List:

<table>
<thead>
<tr>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haul(1)</td>
<td>1.992</td>
<td>4.5526</td>
</tr>
<tr>
<td>Haul(0)</td>
<td>0.8032</td>
<td>5.6813</td>
</tr>
</tbody>
</table>

Figure 63 - About to Terminate Haul(1) in CAP #3

Haul(1) is terminated. During termination, Dump(0) is instantiated with a duration of 0.5.

Simulation Time : 4.5526
Phase : Clock Advance Phase #3
Step : Just after the termination of Haul(1)

Contents of Queues:

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaulersWait</td>
<td>** empty **</td>
</tr>
<tr>
<td>LoadersWait</td>
<td>1 loader</td>
</tr>
<tr>
<td>MovedSoil</td>
<td>** empty **</td>
</tr>
<tr>
<td>SoilToMove</td>
<td>1 hauler-load of soil</td>
</tr>
</tbody>
</table>

Future Events List:

<table>
<thead>
<tr>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dump(0)</td>
<td>4.5526</td>
<td>5.0526</td>
</tr>
<tr>
<td>Haul(0)</td>
<td>0.8032</td>
<td>5.6813</td>
</tr>
</tbody>
</table>

Figure 64 - CAP #3 After the Termination of Haul(1)

The FEL contains no current events. CAP #3 ends.
### Combi Instantiation Phase #4

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaulersWait</td>
<td>** empty **</td>
<td>Dump(0)</td>
<td>4.5526</td>
<td>5.0526</td>
</tr>
<tr>
<td>LoadersWait</td>
<td>1 loader</td>
<td>Haul(0)</td>
<td>0.8032</td>
<td>5.6813</td>
</tr>
<tr>
<td>MovedSoil</td>
<td>** empty **</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SoilToMove</td>
<td>1 hauler-load of soil</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 65 - About to Start CIP #3**

Load is scanned. It cannot start because HaulersWait is empty. CIP #4 is over.

### Clock Advance Phase #4

The simulation clock is advanced to time 5.0526.

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaulersWait</td>
<td>** empty **</td>
<td>Dump(0)</td>
<td>4.5526</td>
<td>5.0526</td>
</tr>
<tr>
<td>LoadersWait</td>
<td>1 loader</td>
<td>Haul(0)</td>
<td>0.8032</td>
<td>5.6813</td>
</tr>
<tr>
<td>MovedSoil</td>
<td>** empty **</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SoilToMove</td>
<td>1 hauler-load of soil</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 66 - About to Terminate Dump(0) in CAP #4**

Dump(0) is terminated. During termination, Return(0) is instantiated with a duration of 3.5892. The hauler released by Dump(0) is now with Return(0). The soil goes to MovedSoil.
Simulation Time : 5.0526
Phase : Clock Advance Phase #4
Step : Just after the termination of Dump(0)

Contents of Queues:

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaulersWait</td>
<td>** empty **</td>
<td>Haul(0)</td>
<td>0.8032</td>
<td>5.6813</td>
</tr>
<tr>
<td>LoadersWait</td>
<td>1 loader</td>
<td>Return(0)</td>
<td>5.0526</td>
<td>8.6418</td>
</tr>
<tr>
<td>MovedSoil</td>
<td>1 hauler-load of soil</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SoilToMove</td>
<td>1 hauler-load of soil</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 67 - CAP #4 After the Termination of Dump(0)

The FEL contains no current events. CAP #4 ends.

**Combi Instantiation Phase #5**

Simulation Time : 5.0526
Phase : Combi Instantiation Phase #5
Step : About to start the CIP

Contents of Queues:

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaulersWait</td>
<td>** empty **</td>
<td>Haul(0)</td>
<td>0.8032</td>
<td>5.6813</td>
</tr>
<tr>
<td>LoadersWait</td>
<td>1 loader</td>
<td>Return(0)</td>
<td>5.0526</td>
<td>8.6418</td>
</tr>
<tr>
<td>MovedSoil</td>
<td>1 hauler-load of soil</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SoilToMove</td>
<td>1 hauler-load of soil</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 68 - About to Start CIP #5

Load is scanned. It cannot start because HaulersWait is empty. CIP #5 is over.

**Clock Advance Phase #5**

The simulation clock is advanced to time 5.6813.
Simulation Time : 5.6813  
Phase : Clock Advance Phase #5  
Step : About to terminate Haul(0)  

Contents of Queues: 
<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaulersWait</td>
<td>** empty **</td>
</tr>
<tr>
<td>LoadersWait</td>
<td>1 loader</td>
</tr>
<tr>
<td>MovedSoil</td>
<td>1 hauler-load of soil</td>
</tr>
<tr>
<td>SoilToMove</td>
<td>1 hauler-load of soil</td>
</tr>
</tbody>
</table>

Future Events List:  
<table>
<thead>
<tr>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haul(0)</td>
<td>0.8032</td>
<td>5.6813</td>
</tr>
<tr>
<td>Return(0)</td>
<td>5.0526</td>
<td>8.6418</td>
</tr>
</tbody>
</table>

*Figure 69 - About to Terminate Haul(0) in CAP #5*

*Haul(0)* is terminated. During termination, *Dump(1)* is instantiated with a duration of 0.5.

Simulation Time : 5.6813  
Phase : Clock Advance Phase #5  
Step : Just after the termination of Haul(0)  

Contents of Queues:  
<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaulersWait</td>
<td>** empty **</td>
</tr>
<tr>
<td>LoadersWait</td>
<td>1 loader</td>
</tr>
<tr>
<td>MovedSoil</td>
<td>1 hauler-load of soil</td>
</tr>
<tr>
<td>SoilToMove</td>
<td>1 hauler-load of soil</td>
</tr>
</tbody>
</table>

Future Events List:  
<table>
<thead>
<tr>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dump(1)</td>
<td>5.6813</td>
<td>6.1813</td>
</tr>
<tr>
<td>Return(0)</td>
<td>5.0526</td>
<td>8.6418</td>
</tr>
</tbody>
</table>

*Figure 70 - CAP #5 After the Termination of Haul(0)*

The FEL contains no current events. CAP #5 ends.
Combi Instantiation Phase #6

Simulation Time: 5.6813
Phase: Combi Instantiation Phase #6
Step: About to start the CIP

Contents of Queues:

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaulersWait</td>
<td>** empty **</td>
</tr>
<tr>
<td>LoadersWait</td>
<td>1 loader</td>
</tr>
<tr>
<td>MovedSoil</td>
<td>1 hauler-load of soil</td>
</tr>
<tr>
<td>SoilToMove</td>
<td>1 hauler-load of soil</td>
</tr>
</tbody>
</table>

Future Events List:

<table>
<thead>
<tr>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dump(1)</td>
<td>5.6813</td>
<td>6.1813</td>
</tr>
<tr>
<td>Return(0)</td>
<td>5.0526</td>
<td>8.6418</td>
</tr>
</tbody>
</table>

Figure 71 - About to Start CIP #6

Load is scanned. It cannot start because HaulersWait is empty. CIP #6 is over.

Clock Advance Phase #6

The simulation clock is advanced to time 6.1813.

Simulation Time: 6.1813
Phase: Clock Advance Phase #6
Step: About to terminate Dump(1)

Contents of Queues:

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaulersWait</td>
<td>** empty **</td>
</tr>
<tr>
<td>LoadersWait</td>
<td>1 loader</td>
</tr>
<tr>
<td>MovedSoil</td>
<td>1 hauler-load of soil</td>
</tr>
<tr>
<td>SoilToMove</td>
<td>1 hauler-load of soil</td>
</tr>
</tbody>
</table>

Future Events List:

<table>
<thead>
<tr>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dump(1)</td>
<td>5.6813</td>
<td>6.1813</td>
</tr>
<tr>
<td>Return(0)</td>
<td>5.0526</td>
<td>8.6418</td>
</tr>
</tbody>
</table>

Figure 72 - About to Terminate Dump(1) in CAP #6

Dump(1) is terminated. During termination, Return(1) is instantiated with a duration of 2.1305 and MovedSoil receives another hauler-load of soil.
Simulation Time : 6.1813  
Phase : Clock Advance Phase #6  
Step : Just after the termination of Dump(1)

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaulersWait</td>
<td>** empty **</td>
<td>Return(1)</td>
<td>6.1813</td>
<td>8.3119</td>
</tr>
<tr>
<td>LoadersWait</td>
<td>1 loader</td>
<td>Return(0)</td>
<td>5.0526</td>
<td>8.6418</td>
</tr>
<tr>
<td>MovedSoil</td>
<td>2 hauler-loads of soil</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SoilToMove</td>
<td>1 hauler-load of soil</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 73 - CAP #6 Just After the Termination of Dump(1)

The FEL contains no current events. CAP #6 ends.

**Combi Instantiation Phase #7**

Simulation Time : 6.1813  
Phase : Combi Instantiation Phase #7  
Step : About to start the CIP

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaulersWait</td>
<td>** empty **</td>
<td>Return(1)</td>
<td>6.1813</td>
<td>8.3119</td>
</tr>
<tr>
<td>LoadersWait</td>
<td>1 loader</td>
<td>Return(0)</td>
<td>5.0526</td>
<td>8.6418</td>
</tr>
<tr>
<td>MovedSoil</td>
<td>2 hauler-loads of soil</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SoilToMove</td>
<td>1 hauler-load of soil</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 74 - About to Start CIP #7

Load is scanned. It cannot start because HaulersWait is empty. CIP #7 is over.

**Clock Advance Phase #7**

The simulation clock is advanced to time 8.3119.
Simulation Time : 8.3119
Phase : Clock Advance Phase #7
Step : About to terminate Return(1)

Contents of Queues:

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaulersWait</td>
<td>** empty **</td>
<td>Return(1)</td>
<td>6.1813</td>
<td>8.3119</td>
</tr>
<tr>
<td>LoadersWait</td>
<td>1 loader</td>
<td>Return(0)</td>
<td>5.0526</td>
<td>8.6418</td>
</tr>
<tr>
<td>MovedSoil</td>
<td>2 hauler-loads of soil</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SoilToMove</td>
<td>1 hauler-load of soil</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 75 - About to Terminate Return(1) in CAP #7**

Return(1) is terminated. During termination, a hauler enters HaulersWait.

Simulation Time : 8.3119
Phase : Clock Advance Phase #7
Step : Just after the termination of Return(1)

Contents of Queues:

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaulersWait</td>
<td>1 hauler</td>
<td>Return(0)</td>
<td>5.0526</td>
<td>8.6418</td>
</tr>
<tr>
<td>LoadersWait</td>
<td>1 loader</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MovedSoil</td>
<td>2 hauler-loads of soil</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SoilToMove</td>
<td>1 hauler-load of soil</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 76 - CAP #7 After the Termination of Return(1)**

The FEL contains no current events. CAP #7 ends.

**Combi Instantiation Phase #8**

Simulation Time : 8.3119
Phase : Combi Instantiation Phase #8
Step : About to start the CIP

Contents of Queues:

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaulersWait</td>
<td>1 hauler</td>
<td>Return(0)</td>
<td>5.0526</td>
<td>8.6418</td>
</tr>
<tr>
<td>LoadersWait</td>
<td>1 loader</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MovedSoil</td>
<td>2 hauler-loads of soil</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SoilToMove</td>
<td>1 hauler-load of soil</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 77 - About to Start CIP #8**
Load is scanned. It is possible to start it, so Load(2) is instantiated with a duration of 1.8740. The startup of Load(2) removes 1 loader from LoadersWait, 1 hauler from HaulersWait, and 1 hauler-load of soil from SoilToMove.

<table>
<thead>
<tr>
<th>Simulation Time</th>
<th>8.3119</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase</td>
<td>Combi Instantiation Phase #8</td>
</tr>
<tr>
<td>Step</td>
<td>Just after the instantiation of Load(2)</td>
</tr>
</tbody>
</table>

### Contents of Queues:

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaulersWait</td>
<td>** empty **</td>
</tr>
<tr>
<td>LoadersWait</td>
<td>** empty *</td>
</tr>
<tr>
<td>MovedSoil</td>
<td>2 hauler-loads of soil</td>
</tr>
<tr>
<td>SoilToMove</td>
<td>** empty **</td>
</tr>
</tbody>
</table>

### Future Events List:

<table>
<thead>
<tr>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return(0)</td>
<td>5.0526</td>
<td>8.6418</td>
</tr>
<tr>
<td>Load(2)</td>
<td>8.3119</td>
<td>10.1859</td>
</tr>
</tbody>
</table>

**Figure 78 - Just After the Instantiation of Load(2) in CIP #8**

Load is scanned again. It cannot start because all of its predecessor Queues are empty. Since it is no longer possible to start any Combi, the CIP ends. Notice that since there are no links that bring soil into SoilToMove, SoilToMove will remain empty until the end of the simulation. For this reason, Load will not be able to start anymore.

### Clock Advance Phase #8

The simulation clock is advanced to time 8.6418.

<table>
<thead>
<tr>
<th>Simulation Time</th>
<th>8.6418</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase</td>
<td>Clock Advance Phase #8</td>
</tr>
<tr>
<td>Step</td>
<td>About to terminate Return(0)</td>
</tr>
</tbody>
</table>

### Contents of Queues:

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaulersWait</td>
<td>** empty **</td>
</tr>
<tr>
<td>LoadersWait</td>
<td>** empty *</td>
</tr>
<tr>
<td>MovedSoil</td>
<td>2 hauler-loads of soil</td>
</tr>
<tr>
<td>SoilToMove</td>
<td>** empty **</td>
</tr>
</tbody>
</table>

### Future Events List:

<table>
<thead>
<tr>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return(0)</td>
<td>5.0526</td>
<td>8.6418</td>
</tr>
<tr>
<td>Load(2)</td>
<td>8.3119</td>
<td>10.1859</td>
</tr>
</tbody>
</table>

**Figure 79 - About to Terminate Return(0) in CAP #8**

Return(0) is terminated. During termination, HaulersWait receives a Hauler.
Simulation Time : 8.6418  
Phase : Clock Advance Phase #8  
Step : Just after the termination of Return(0)  

Contents of Queues:

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaulersWait</td>
<td>1 hauler</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LoadersWait</td>
<td>** empty *</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MovedSoil</td>
<td>2 hauler-loads of soil</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SoilToMove</td>
<td>** empty **</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Future Events List:

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td></td>
<td>Load(2)</td>
<td>8.3119</td>
<td>10.1859</td>
</tr>
</tbody>
</table>

Figure 80 - CAP #8 Just After the Termination of Return(0)

The FEL contains no current events, so the CAP ends.

**Combi Instantiation Phase #9**

Simulation Time : 8.6418  
Phase : Combi Instantiation Phase #9  
Step : About to start the CIP  

Contents of Queues:

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaulersWait</td>
<td>1 hauler</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LoadersWait</td>
<td>** empty *</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MovedSoil</td>
<td>2 hauler-loads of soil</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SoilToMove</td>
<td>** empty **</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Future Events List:

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td></td>
<td>Load(2)</td>
<td>8.3119</td>
<td>10.1859</td>
</tr>
</tbody>
</table>

Figure 81 - About to Start CIP #9

Load is scanned. It is not possible to start it because SoilToMove and LoadersWait are empty. The CIP ends.

**Clock Advance Phase #9**

The simulation clock is advanced to time 10.1859.
Simulation Time : 10.1859
Phase : Clock Advance Phase #9
Step : About to terminate Load(2)

Contents of Queues:

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaulersWait</td>
<td>1 hauler</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LoadersWait</td>
<td>** empty **</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MovedSoil</td>
<td>2 hauler-loads of soil</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SoilToMove</td>
<td>** empty **</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Future Events List:

- Load(2) 8.3119 10.1859

Figure 82 - About to Terminate Load(2) in CAP #9

Load(2) is terminated. During termination, Haul(2) is instantiated with a duration of 2.5136 and LoadersWait receives a loader.

Simulation Time : 10.1859
Phase : Clock Advance Phase #9
Step : Just after the termination of Load(2)

Contents of Queues:

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaulersWait</td>
<td>1 hauler</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LoadersWait</td>
<td>1 loader</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MovedSoil</td>
<td>2 hauler-loads of soil</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SoilToMove</td>
<td>** empty **</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 83 - CAP #9 Just After the Termination of Load(2)

The FEL contains no current events, so the CAP ends.
Combi Instantiation Phase #10

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaulersWait</td>
<td>1 hauler</td>
</tr>
<tr>
<td>LoadersWait</td>
<td>1 loader</td>
</tr>
<tr>
<td>MovedSoil</td>
<td>2 hauler-loads of soil</td>
</tr>
<tr>
<td>SoilToMove</td>
<td>** empty **</td>
</tr>
</tbody>
</table>

Future Events List:

<table>
<thead>
<tr>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haul(2)</td>
<td>10.1859</td>
<td>12.6994</td>
</tr>
</tbody>
</table>

Figure 84 - About to Start CIP #10

Load is scanned. It cannot start because SoilToMove is empty. The CIP ends.

Clock Advance Phase #10

The simulation clock is advanced to time 12.6994.

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaulersWait</td>
<td>1 hauler</td>
</tr>
<tr>
<td>LoadersWait</td>
<td>1 loader</td>
</tr>
<tr>
<td>MovedSoil</td>
<td>2 hauler-loads of soil</td>
</tr>
<tr>
<td>SoilToMove</td>
<td>** empty **</td>
</tr>
</tbody>
</table>

Future Events List:

<table>
<thead>
<tr>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haul(2)</td>
<td>10.1859</td>
<td>12.6994</td>
</tr>
</tbody>
</table>

Figure 85 - About to Terminate Haul(2) in CAP #10

Haul(2) is terminated. During termination, Dump(2) is instantiated with a duration of 0.5.
Simulation Time : 12.6994  
Phase : Clock Advance Phase #10  
Step : Just after the termination of *Haul*(2)  

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>HaulersWait</em></td>
<td>1 hauler</td>
<td></td>
<td>12.6994</td>
<td>13.1994</td>
</tr>
<tr>
<td><em>LoadersWait</em></td>
<td>1 loader</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>MovedSoil</em></td>
<td>2 hauler-loads of soil</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>SoilToMove</em></td>
<td>** empty **</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Figure 86 - CAP #10 Just After the Termination of Haul(2) *

The FEL contains no current events, so the CAP ends.

**Combi Instantiation Phase #11**

Simulation Time : 12.6994  
Phase : Combi Instantiation Phase #11  
Step : About to start the CIP  

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>HaulersWait</em></td>
<td>1 hauler</td>
<td></td>
<td>12.6994</td>
<td>13.1994</td>
</tr>
<tr>
<td><em>LoadersWait</em></td>
<td>1 loader</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>MovedSoil</em></td>
<td>2 hauler-loads of soil</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>SoilToMove</em></td>
<td>** empty **</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Figure 87 - About to Start CIP #11 *

*Load* is scanned. It cannot start because *SoilToMove* is empty. The CIP ends.

**Clock Advance Phase #11**

The simulation clock is advanced to time 13.1994.
Simulation Time: 13.1994  
Phase: Clock Advance Phase #11  
Step: About to terminate *Dump* (2)  

**Contents of Queues:**  

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaulersWait</td>
<td>1 hauler</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LoadersWait</td>
<td>1 loader</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MovedSoil</td>
<td>2 hauler-loads of soil</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SoilToMove</td>
<td>** empty **</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Future Events List:**  

<table>
<thead>
<tr>
<th>Event</th>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dump (2)</td>
<td>12.6994</td>
<td>13.1994</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 88* - About to Terminate *Dump* (2) in CAP #11

*Dump* (2) is terminated. During termination, 1 hauler-load of soil goes to *MovedSoil*, and *Return* (2) is instantiated with a duration of 4.1909.

Simulation Time: 13.1994  
Phase: Clock Advance Phase #11  
Step: Just after the termination of *Dump* (2)  

**Contents of Queues:**  

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaulersWait</td>
<td>1 hauler</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LoadersWait</td>
<td>1 loader</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MovedSoil</td>
<td>3 hauler-loads of soil</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SoilToMove</td>
<td>** empty **</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Figure 89* - CAP #11 Just After the Termination of *Dump* (2)

**Combi Instantiation Phase #12**

Simulation Time: 13.1994  
Phase: Combi Instantiation Phase #12  
Step: About to start the CIP  

**Contents of Queues:**  

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
<th>Instance</th>
<th>Started</th>
<th>Will End</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaulersWait</td>
<td>1 hauler</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LoadersWait</td>
<td>1 loader</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MovedSoil</td>
<td>3 hauler-loads of soil</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SoilToMove</td>
<td>** empty **</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Figure 90* - About to Start CIP #12
Load is scanned. It cannot start because SoilToMove is empty. The CIP ends.

Clock Advance Phase #12

The simulation clock is advanced to time 17.3903

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaulersWait</td>
<td>1 hauler</td>
</tr>
<tr>
<td>LoadersWait</td>
<td>1 loader</td>
</tr>
<tr>
<td>MovedSoil</td>
<td>3 hauler-loads of soil</td>
</tr>
<tr>
<td>SoilToMove</td>
<td>** empty **</td>
</tr>
</tbody>
</table>

**Figure 91 - About to Terminate Return(2) in CAP #12**

Return(2) is terminated. During termination, HaulersWait receives a hauler.

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaulersWait</td>
<td>2 haulers</td>
</tr>
<tr>
<td>LoadersWait</td>
<td>1 loader</td>
</tr>
<tr>
<td>MovedSoil</td>
<td>3 hauler-loads of soil</td>
</tr>
<tr>
<td>SoilToMove</td>
<td>** empty **</td>
</tr>
</tbody>
</table>

**Figure 92 - CAP #12 Just After the Termination of Return(2)**

The FEL contains no current events (but notice that it is empty), so the CAP ends.
Combi Instantiation Phase #13

Simulation Time : 17.3903
Phase : Combi Instantiation Phase #13
Step : About to start the CIP

Contents of Queues:

<table>
<thead>
<tr>
<th>Queue</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>HaulersWait</td>
<td>2 haulers</td>
</tr>
<tr>
<td>LoadersWait</td>
<td>1 loader</td>
</tr>
<tr>
<td>MovedSoil</td>
<td>3 hauler-loads of soil</td>
</tr>
<tr>
<td>SoilToMove</td>
<td>** empty **</td>
</tr>
</tbody>
</table>

Future Events List:

(The FEL is empty)

Figure 93 - About to Start CIP #13

Load is scanned. It cannot start because SoilToMove is empty. The CIP ends.

Clock Advance Phase #13

Stroboscope attempts to advance the clock. It looks at the FEL to get the next clock time from the Activity instance at the head of the FEL. Since it finds the FEL empty, it cannot advance the clock at all. The simulation ends. There are no more resources to enable further Activity instantiations. Simulation stops due to a lack of resources.

Trace Summary

The earth-moving process traced in this appendix is very short. It involves the load- haul-dump-return cycle three times. The processing of the model, however, involves many events. The events that occurred during the run can be summarized as follows:

<table>
<thead>
<tr>
<th>Clock</th>
<th>Event</th>
<th>Instance</th>
<th>Start</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0000</td>
<td>START:</td>
<td>Load(0)</td>
<td>S: 0.0000</td>
<td>E: 0.8032</td>
</tr>
<tr>
<td>0.8032</td>
<td>END:</td>
<td>Load(0)</td>
<td>S: 0.0000</td>
<td>E: 0.8032</td>
</tr>
<tr>
<td></td>
<td>START:</td>
<td>Haul(0)</td>
<td>S: 0.8032</td>
<td>E: 5.6813</td>
</tr>
<tr>
<td>1.9920</td>
<td>END:</td>
<td>Load(1)</td>
<td>S: 0.8032</td>
<td>E: 4.5526</td>
</tr>
<tr>
<td>4.5526</td>
<td>END:</td>
<td>Haul(1)</td>
<td>S: 1.9920</td>
<td>E: 5.0526</td>
</tr>
<tr>
<td>5.0526</td>
<td>END:</td>
<td>Dump(0)</td>
<td>S: 4.5526</td>
<td>E: 5.0526</td>
</tr>
<tr>
<td></td>
<td>START:</td>
<td>Return(0)</td>
<td>S: 5.0526</td>
<td>E: 8.6418</td>
</tr>
</tbody>
</table>
The listing above is a simplified version of the trace output that Stroboscope can produce automatically. The trace output is useful in verifying that a simulation model is behaving as intended.
Bibliography
Bibliography


