Old Dominion University
ODU Digital Commons

Electrical & Computer Engineering Theses & Disssertations

Electrical & Computer Engineering

Spring 2007

The Distributed Independent-Platform Event-Driven Simulation Engine Library (DIESEL)

Reejo Mathew
Old Dominion University

Follow this and additional works at: https://digitalcommons.odu.edu/ece_etds

Part of the Electrical and Computer Engineering Commons

Recommended Citation

Mathew, Reejo. "The Distributed Independent-Platform Event-Driven Simulation Engine Library (DIESEL)" (2007). Doctor of Philosophy (PhD), dissertation, Electrical/Computer Engineering, Old Dominion University, DOI: 10.25777/4mty-ka78
https://digitalcommons.odu.edu/ece_etds/97

This Dissertation is brought to you for free and open access by the Electrical & Computer Engineering at ODU Digital Commons. It has been accepted for inclusion in Electrical & Computer Engineering Theses & Disssertations by an authorized administrator of ODU Digital Commons. For more information, please contact digitalcommons@odu.edu.

THE DISTRIBUTED INDEPENDENT-PLATFORM EVENT-DRIVEN SIMULATION ENGINE LIBRARY (DIESEL)

by

Reejo Mathew
M.S. December 2002, Old Dominion University

A Dissertation submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirement for the Degree of

DOCTOR OF PHILOSOPHY

ELECTRICAL AND COMPUTER ENGINEERING

OLD DOMINION UNIVERSITY
May 2007

Approved by:

James F. Leathrum, Jr. (Director)

Roland R. Mielke (Member)

Lee A. Belfore II (Member)

C. Michael Overstreet (Member)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ABSTRACT

THE DISTRIBUTED INDEPENDENT-PLATFORM EVENT-DRIVEN SIMULATION ENGINE LIBRARY (DIESEL)

Reejo Mathew
Old Dominion University, 2007
Director: Dr. James F. Leathrum, Jr.

The Distributed, Independent-Platform, Event-Driven Simulation Engine Library (DIESEL) is a simulation executive, capable of supporting both sequential and distributed discrete-event simulations. A system level specification is provided along with the expected behavior of each component within DIESEL. This behavioral specification of each component, along with the interconnection and interaction between the different components, provides a complete description of the DIESEL behavioral model. The model provides a considerable amount of freedom for an application developer to partition the simulation model, when building sequential and distributed applications with respect to balancing the number of events generated across different components. It also allows a developer to modify underlying algorithms in the simulation executive, while causing no changes to the overall system behavior so long as the algorithms meet the behavioral specifications.

The behavioral model is object-oriented and developed using a hierarchical approach. The model is not targeted towards any programming language or hardware platform for implementation. The behavioral specification provides no specifics about how the model should be implemented. A complete and stable implementation of the behavioral model is provided as a proof-of-concept, and can be used to develop

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

commercial applications. New and independent implementations of the complete model can be developed to support specific commercial and research efforts. Specific components of the model can also be implemented by students in an educational environment, using strategies different from the ones used within the current implementation. DIESEL provides a research environment for studying different aspects of Parallel Discrete-Event Simulation, such as event management strategies, synchronization algorithms, communication mechanisms, and simulation state capture capabilities.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

This dissertation is dedicated to the saying
"The day you are through learning, you are through."

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ACKNOWLEDGMENTS

I would like to extend special thanks to Dr. James F. Leathrum, Jr., my mentor and dissertation advisor, for the knowledge and guidance that he has given me during the last seven years. I will fondly remember our spirited discussions about Yankees baseball (my passion) and Duke basketball (his passion), although we have yet to find common ground on either topic. I would like to believe that he has learned as much about soccer (or football as it is known in the rest of the world) and cricket from me, as I have learned about baseball traditions and the Princeton offense in basketball from him.

I would like to thank the other members of my advisory committee, Dr. Roland Mielke, Dr. Lee Belfore and Dr. Michael Overstreet, for all their help and support and for their amazing ability to adjust their schedules to be present at each one of the numerous examinations that I had to undergo as part of the PhD program. I would also like to thank my colleague, Mr. Saurav Mazumdar, of the Department of Electrical and Computer Engineering at Old Dominion University, for his prototypical implementations of the DIESEL model in C# and Java and for his valuable suggestions during the model development.

I would like to thank my friends Shruti, Paul, Avi, Anand, Hrishi and lots of others for helping me keep my sanity intact and my disposition cheerful during the times that I needed it most.

My family has played a big role in all my achievements in my life and my career. My brother, Rishi, though younger than me, has been a source of inspiration for me throughout my career. I owe everything in my life to my parents, Annie and V.C.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Mathew. This doctorate is a testament to everything that I have learned from them and my most cherished gift to them. Last but definitely not least, I dedicate this dissertation to my sister, Reenu for her words of encouragement, her voice of reason, her generosity and her inability to refuse whenever I asked her for monetary assistance. I love and respect her for who she is and what she has done for me.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE OF CONTENTS

Page

LIST OF TABLES xi

LIST OF FIGURES xii

LIST OF GRAPHS xv

ABBREVIATIONS AND ACRONYMS xvi

Chapter

I. INTRODUCTION........ 1
   1.1 Overview 2
   1.2 Background 2
   1.3 Related Work 13
      1.3.1 Parallel and Discrete-Event Simulation (PDES) 13
      1.3.2 Research / Educational Frameworks 16
   1.4 Chapter Plan 17

II. PROBLEM STATEMENT 18
   2.1 Problem Statement 18
   2.2 Research Purpose 20
   2.3 Model Requirements 23
   2.4 Definitions 24

III. INTERFACE SPECIFICATION FOR THE DIESEL SEQUENTIAL MODEL.... 26
   3.1 Interaction between Components within DIESEL 26
   3.2 DIESEL Engine Interfaces 31
      3.2.1 *Delegate* Interface 31
      3.2.2 *ArgumentList* Interface 33
      3.2.3 *SimulationEngineComponent* Interface 35
   3.3 DIESEL Delayed State Commitment Interfaces 36
   3.4 DIESEL State Capture and Restore Interfaces 38
   3.5 DIESEL Base Interfaces 40
   3.6 DIESEL Random Variate Model Interfaces 41
   3.7 DIESEL Synchronization Interfaces 44
   3.8 DIESEL Entity Management Interfaces 48
   3.9 DIESEL Sequential Simulation Executive 52
   3.10 Implementation Classes for the DIESEL Engine Interfaces 53
      3.10.1 *SimulationCluster* Class 53
      3.10.2 *SimulationEngine* Class 54

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.10.3 *SimulationEngineComponent* Class 56
3.10.4 *EventManager* Class 57
3.11 Examples 59
3.11.1 Timing diagram for execution of events 59
3.11.2 Dining Philosophers' Problem 66

IV. INTERFACE SPECIFICATION FOR THE DIESEL DISTRIBUTED MODEL... 69
4.1 Interaction between Components within DIESEL 69
4.2 Extensions to the DIESEL Interface 75
4.3 DIESEL Distributed Simulation Executive 75
4.3.1 *Barrier Synchronization* Algorithm 77
4.3.2 *Time Warp* Paradigm 77
4.4 Support Classes for the DIESEL Distributed Interface 79
4.4.1 *ObjectRegistry* Class 79
4.4.2 *Message* Class 80
4.4.3 *MessageQueue* Class 80
4.4.4 *StateSaveQueue* Class 81
4.5 DIESEL Analytical Support 81

V. CASE STUDIES 84
5.1 Event Management Study 84
5.1.1 Implementation of the Event Management Structure 85
5.1.2 Event Insertion Strategies for a *LinearPendingEventSet* 89
5.1.3 Timing Results 96
5.2 Distributed Simulation Study 99
5.2.1 Distributed Implementation 99
5.2.2 Communication Study 101
5.2.3 Results 103
5.3 Port Simulation (PORTSIM) 105
5.4 Platform Independence Study 110

VI. CONCLUSION 113
6.1 Achievements 113
6.2 Enhancements 114

REFERENCES 117

APPENDICES 125
A DIESEL Sequential Interface Specification 125
A.1 State Capture Interfaces 125
A.1.1 *StateSave* Interface 125
A.1.2 *StateSaveSupport* Interface 126
A.2 Application Support Interfaces 131
A.2.1 *Replicable* Interface 131

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A.2.2 *ProgrammaticEvent* Interface 132
A.2.3 *Routine* Interface 133
A.2.4 *Condition* Interface 134
A.3 Application Interfaces 135
A.3.1 *SequentialSimulationExecutive* Interface 135
A.3.2 *ArgumentList* Interface 137
A.3.3 *Delegate* Interface 142
A.3.4 *SimulationEngineComponent* Interface 145
A.3.5 *DelayedCommit* Interface 150
A.4 Synchronization Interfaces 152
A.4.1 *Trigger* Interface 152
A.4.2 *TriggerCounter* Interface 155
A.4.3 *Join* Interface 158
A.4.4 *JoinCounter* Interface 161
A.5 Random Number Generation Interfaces 164
A.5.1 *RNG* Interface 164
A.5.2 *AdvancedDistributions* Interface 168
A.6 Entity Management Interfaces 174
A.6.1 *Set* Interface 174
A.6.2 *FIFO* Interface 180
A.6.3 *LIFO* Interface 183
A.6.4 *BinaryTree* Interface 187
A.6.5 *PriorityQueue* Interface 192
A.6.6 *QueueWithStatistics* Interface 195
A.6.7 *EntityCounter* Interface 200
A.6.8 *EntityPool* Interface 210
B DIESEL Distributed Interface Specification................................ 220
B.1 *DistributedSimulationExecutive* Interface 220
B.2 Modified *SimulationEngineComponent* Interface 223
B.3 *SimulationCluster* Interface 224

VITA 228

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

LIST OF TABLES

Table Page

1. Messages passed and average pending events with 6 *SimulationClusters* 103

2. Messages passed and average pending events with 3 *SimulationClusters* 103

3. Messages passed and average pending events with 2 *SimulationClusters* 104

4. Messages passed and average pending events with 2 *SimulationClusters* with L-shape organization 104

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

LIST OF FIGURES

Figure Page

1. Distribution Simulation Methodology 5

2. Time Warp Logical Process (TWLP) 11

3. Communication between TWLPs 12

4. DIESEL Interface 19

5. *SimulationCluster* 27

6. Inherited *SimulationEngineComponent* 28

7. Global *SimulationEngineComponent* 28

8. Static *SimulationEngineComponent* 30

9. Dynamic *SimulationEngineComponents* 30

10. *Delegate* Interface 33

11. *ArgumentList* Interface 34

12. *SimulationEngineComponent* Interface 36

13. *DelayedCommit* Interface 38

14. *StateSave* Interface 39

15. *StateSaveSupport* Interface 40

16. *Replicable* Interface 41

17. *ProgrammaticEvent* Interface 41

18. *RNG* Interface 43

19. *AdvancedDistributions* Interface 44

20. Dependent Triggers 45

21. *Trigger* Interface 46

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

22. *TriggerCounter* Interface 46

23. Dependent Joins 47

24. *Join* Interface 47

25. *JoinCounter* Interface 47

26. *Set* Interface 49

27. *FIFO* Interface 49

28. *LIFO* Interface 49

29. *BinaryTree* Interface 49

30. *PriorityQueue* Interface 50

31. *QueueWithStatistics* Interface 50

32. *EntityCounter* Interface 51

33. *EntityPool* Interface 51

34. *SequentialSimulationExecutive* Interface 52

35. Basic Sequential Simulation 53

36. *SimulationCluster* Class 54

37. *SimulationEngine* Class 54

38. *SimulationEngineComponent* Class 56

39. *EventManager* Class 58

40. Timing Diagram 61

41. Snapshot of Dining Philosophers 67

42. *SimulationForest* 70

43. Registry of processors within simulation executive 71

44. Registry of objects within a *SimulationCluster* 72

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

45. Reference resolution between *SimulationClusters* 73

46. Communication between *SimulationClusters* 74

47. Modified *SimulationEngineComponent* Interface 76

48. *SimulationCluster* Interface 76

49. *DistributedSimulationExecutive* Interface 76

50. *ObjectRegistry* Class 80

51. *Message* Class 80

52. *MessageQueue* Class 81

53. *StateSaveQueue* Class 81

54. *EventList* Structure 86

55. *EventList* and *EventListNode* Classes 86

56. *LinearPendingEventSet* Structure 87

57. *LinearPendingEventSet* and *LinearPendingEventSetNode* Classes 88

58. *NonLinearPendingEventSet* and *NonLinearPendingEventSetNode* Classes 89

59. Snooker table divided into BallGroups 96

60. *DistributedExecutiveSupport* Interface 100

61. Mapping of BallGroups to *SimulationClusters* 102

62. PORTSIM Architecture 106

63. MapNode within PORTSIM Architecture 107

64. Example PORTSIM ProcessFlow using PPFN 119

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

LIST OF GRAPHS

Graph Page

1. Time spent by Philosophers eating, thinking and waiting 68
2. Average execution time for different distributions 98
3. Average nodes searched for different distributions 98

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface
BTB	Breathing Time Buckets
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
DARPA	Defense Advanced Research Projects Agency
DCOM	Distributed Component Object Model
DES	Discrete-Event Simulation
DFD	Data Flow Diagram
DIESEL	Distributed, Independent-Platform, Event-Driven Simulation Engine Library
DIS	Distributed Interactive Simulation
FOM	Federation Object Model
GTW	Georgia Time Warp
GVT	Global Virtual Time
HLA	High Level Architecture
IDE	Integrated Development Environment
IID	Independent Identically Distributed
IOC	Initial Operational Capability
LP	Logical Process
MIS	Management Information System
MPI	Message Passing Interface
NACHOS	Not Another Completely Heuristic Operating System

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ODU	Old Dominion University
OMG	Object Management Group
OMT	Object Model Template
PDES	Parallel Discrete-Event Simulation
PORTSIM	Port Simulation
PPFN	Programmable Process Flow Network
RMI	Remote Method Invocation
RTI	Runtime Infrastructure
SDDC-TEA	Surface Deployment and Distribution Command - Transportation Engineering Agency
SIMNET	Simulator Networking
SOM	Simulation Object Model
SPEEDES	Synchronous Parallel Environment for Emulation and Discrete-Event Simulation
TWLP	Time Warp Logical Process

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter I

INTRODUCTION

The Distributed, Independent-Platform, Event-Driven Simulation Engine Library (DIESEL) is a behavioral model for a simulation executive, capable of supporting both sequential and distributed Discrete-Event Simulation (DES). The behavioral model provides a system level specification for the simulation executive by describing the different components of the system, specifying the behavior of each component, and detailing the interconnection and interaction between the components. The model is generic, language-independent and platform-independent, and is developed using an object-oriented approach.

DIESEL specifies interfaces for each component of the behavioral model, along with the expected behavior of each procedure within the component interface. This can be used to develop new and independent implementations of one or more components of the model. It also provides support for developing commercial applications by using the same interfaces to build the application in a hierarchical manner.

DIESEL has been developed to bring research into different aspects of sequential and distributed DES under a common research and development platform. A new implementation can be developed for one or more components within the model and can then be inserted into the complete system with no change in system behavior. At a system level, the entire implementation of the behavioral model can be redefined by using

---

The reference model for this work is "An Object-Oriented Architecture for the Simulation of Networks of Cargo Terminal Operations." *Journal of Defense Modeling and Simulation (JDMS)* 2, no. 2, 101-116, 2005.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

different algorithms and data structures without affecting any applications that are built using the simulation executive.

1.1 Overview

Simulations are used extensively to model real-world systems, both existing and proposed, and analyze problems and scenarios that occur within them. Simulations are often a cost-effective method for modeling a complex physical system with a level of detail that adequately represents the real-world system under consideration. Simulations are often a relatively cheap approach for experimenting with an existing or proposed system and for evaluating future modifications and additions to the system when compared to building the actual system. The processing involved in the system needs to be modeled with appropriate detail so that the developed model provides an accurate representation of the system necessary to meet the stated objectives of the simulation. An analysis of the simulation results can then be used to aid in the decision-making to improve the performance and efficiency of the system. Simulations can also provide a safe alternative to evaluating scenarios such as war planning, where it is too dangerous and costly to conduct actual exercises during planning and training.

1.2 Background

The following terms are commonly used within a DES:

- ***Physical Time:*** Physical Time is the time within the real-world or proposed system being simulated. In a digital circuit, the physical time might be in terms of microseconds or nanoseconds, while in a port simulation, the physical time might

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

be in terms of days, weeks, or months.

- ***Simulation (Virtual) Time:*** Simulation or Virtual Time is an abstract concept used by a simulation to represent physical time. Fujimoto in [1] describes simulation time as a totally ordered set of values where each value represents a unique instant of physical time in the system being simulated.
- ***Wallclock Time:*** Wallclock time refers to the advancement of real time during the execution of the simulation. For analytical simulations, wallclock time could start from zero, while for real-time simulations wallclock time could be synchronized with a true wallclock.
- ***Event:*** At any instant within a simulation, the state of the system can be derived from the simulation state variables. An event is defined as an act which changes the state of the system by modifying its state variables. In addition, an event can schedule other events to be executed in the future.

A DES uses events to model discrete changes of state within a modeled system, and the simulation progresses with events being processed in order of their timestamps. A DES can be executed on a single processor with common time management for all components of the simulation. A DES can also be mapped to a collection of processors connected by a network (with each processor on the network modeling a particular component) and then executed in parallel. This is known as a Parallel Discrete-Event Simulation (PDES) [1, 2]. PDES can offer several advantages over executing a simulation on a single sequential processor, such as model parallelism and reduced execution time for the simulation.

Simulations are usually composed of different components interacting with each

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

other. Each component represents an individual process or set of processes within the modeled system. These components can be intuitively mapped onto processors in the network within a PDES with a single processor simulating one component or a group of related components, thus preserving their natural structure. This structure can also take advantage of any potential parallelism within the model, with each component executing in isolation, independent of another component, until it needs to interact or synchronize with another component.

Mapping a large simulation model to a network of processors by dividing the model into distinct components and providing each component within the simulation local access to its data and code potentially speeds up the execution of the whole simulation model. The maximum speedup possible within a PDES is a factor of n, where n is the number of processors available. It is rarely possible to achieve this ideal speedup due to interconnection and synchronization delays among components. However, the goal of a PDES is to generally execute faster than the equivalent sequential simulation.

Simulation users might need access to individual components of the simulation, without having to worry about other components, or interfering with their execution. A PDES can provide this isolation of components. This property is even more beneficial when the users are at separate geographical locations, with the PDES providing the user with access to the relevant components of the simulation model. Distributed interaction by different users also requires distributed time management within the simulation which is provided by the PDES.

The development of a PDES system can be described with a hierarchical, top-down approach as shown in Figure 1. Given a system of interest to be modeled, the first

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

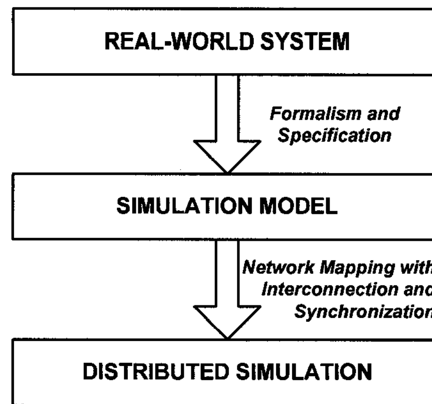


Figure 1. Distributed Simulation Methodology

step is to develop an abstraction of the system, i.e., a model specification. The components of the model and the input/output relationship among them need to be clearly defined to develop a valid and correct model of the system. The model specification can then be mapped onto a distributed simulation architecture by designing the interconnection and synchronization mechanism among the components.

A PDES system consists of multiple processes interacting with each other to complete the simulation task. Each such process simulates either an actual process within the system of interest or a process to support simulation execution, such as complex statistical analysis or construction of graphical images. Such a process is called a logical process (LP). A PDES can be executed either on a single processor or a network of processors with a communication medium connecting the processors. The number of LPs is independent of the number of processors available for the simulation. All the LPs can reside on the same processor in a single-processor environment or can be distributed across the network. In concept, LPs interact with each other by transmitting messages among each other. Messages are used by LPs to schedule events on other LPs, to transmit synchronization information, to report errors, etc. The actual content of a message might

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

include additional information unique to the synchronization paradigm being employed.

Within any simulation, the *happened before* relation needs to be defined for the ordering of events. Let $\rightarrow$ denote the *happened before* relation. Then

- If a and b are events within the same LP, and a comes before b in simulation time, then $a \rightarrow b$.
- If a is the act of sending a message by one LP, and b is the receipt of the same message by another LP, then $a \rightarrow b$.
- If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.
- Two distinct events a and b are said to be concurrent if $a \not\rightarrow b$ and $b \not\rightarrow a$.

Lamport [3] defines a virtual clock as:

- A virtual clock C is a logical structure with no relation to physical time.
- C_i exists in each LP P_i.
- C_i assigns a timestamp $C_i\langle a\rangle$ to each event in P_i.

Then, Lamport's clock conditions state that:

For any events a, b

if $a \rightarrow b$, then $C\langle a\rangle < C\langle b\rangle$

This can be expanded to include the *happened before* relation as:

- If a and b are events within the LP P_i and $a \rightarrow b$, then $C_i\langle a\rangle < C_i\langle b\rangle$.
- If a is the act of sending a message by LP P_i, and b is the receipt of the same message by LP P_j, then $C_i\langle a\rangle < C_j\langle b\rangle$.
- If $C\langle a\rangle < C\langle b\rangle$ and $C\langle b\rangle < C\langle c\rangle$, then $C\langle a\rangle < C\langle c\rangle$.
- If a and b are simultaneous events, i.e. events that have the same timestamp, then no *happened before* relation exists between $C\langle a\rangle$ and $C\langle b\rangle$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A DES when executed as a sequential simulation should process events in increasing order of their timestamps so that the *happened before* relation can be maintained for the events. A DES when executed as a distributed simulation should also process events, whether generated locally within a LP or generated by other LPs, in increasing order of their timestamps. This is known as the *Local Causality Constraint* [1] and is a sufficient condition for a DES. If each LP within a distributed simulation adheres to the *Local Causality Constraint*, then the distributed simulation will produce identical results to the equivalent sequential simulation, provided that both the sequential and distributed simulations execute simultaneous events in the same order.

A distributed simulation can reorder events in violation of the *Local Causality Constraint* provided that the reordering does not violate inherent computational dependencies between events. This is rarely done since the simulation executive requires knowledge of the computational dependencies between events in order to select an appropriate event to execute. This knowledge requires a higher level of interaction than that required by the *Local Causality Constraint* between the simulation executive and the application, and requires the application to have a higher level of understanding of these dependencies.

A PDES system can employ either of two types of synchronization algorithms to satisfy the *Local Causality Constraint*, conservative and optimistic, as discussed below.

Conservative Synchronization Algorithms

Conservative synchronization algorithms strongly enforce the *Local Causality Constraint* by blocking all LPs from processing any event until it is determined to be safe

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

by the simulation executive. Determining what events are safe to process is an important part of any conservative algorithm. A simple method is for each LP to send a *null message* to every other LP indicating the lower bound on the time stamp of messages it will send at any point in the future. This lower bound is calculated by adding a quantity called *lookahead* to the local simulation time of the LP. Fujimoto [1] states that if a LP at simulation time T can only schedule new events with timestamp of at least $(T + L)$, then L is the *lookahead* for the LP. The *lookahead* is determined by the simulation application, and can change during the simulation. However, this method can lead to deadlock within the simulation, with LPs waiting for each other to process events further.

An approach known as *barrier synchronization* can be used to address deadlock, where each LP executes its safe events and then waits for every other LP to reach an agreed upon synchronization point. Once each LP has reached that point, all LPs are allowed to execute their next batch of safe events. Barriers can be implemented in a variety of ways by organizing LPs to reduce the messages that need to be sent between LPs to execute the barrier primitive [1].

The advantages of conservative algorithms are decreased complexity of the simulation executive, the absence of causality errors, and modest memory requirements. The disadvantages of these algorithms are LPs waiting for a notification that its next event is safe to be processed and the possibility of deadlocks with each LP waiting for a notification from all other LPs.

Optimistic Synchronization Algorithms

Optimistic synchronization algorithms, on the other hand, allow violations of the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Local Causality Constraint, by allowing the LPs to continue executing events and then returning to a safe state when causality errors occur. Optimistic algorithms account for causality violations by rolling back to a saved safe state (which has been checkpointed), undoing all state changes which occurred after the checkpoint, and by sending special messages called anti-messages to reverse the effect of improperly sent messages after the checkpoint. The execution of an event is provisional; it is conditional on the fact that no message arrives at the LP with a timestamp less than the event's timestamp. Optimistic algorithms have been developed on the premise that improved simulation performance can be achieved even with the overhead of undoing and possibly repeating one or more event computations.

The *Time Warp* paradigm [4, 5] is one of the most widely discussed optimistic synchronizing mechanisms for a PDES system. The *Time Warp* paradigm derives its name from the behavior that virtual clocks of different LPs go back and forth, independent of each other, reverting back to correct causality errors, while generally progressing for the duration of the simulation. At any point in the simulation, some LPs are allowed to progress ahead in terms of virtual time, while other LPs lag behind in terms of their local simulation times. The paradigm is independent of any underlying computer architecture. However, a reliable communication medium is assumed so that no messages are lost in transit.

The *Time Warp* paradigm requires each LP to have the following:

- A *process identifier* unique to an LP indicating the virtual space co-ordinate of the LP. The set of all *process identifiers* constitute the virtual space of the simulation.
- A *local virtual clock* C_i showing the current virtual time of the LP to indicate the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

virtual time co-ordinate of the LP.

- A *state* denoting the current state of the LP including its state variables, execution stack, etc.
- A *state queue* containing saved states of the LP called checkpoints for rollback purposes. It is assumed that the state of each LP is saved after every event computation.
- An *input queue* containing all processed and unprocessed messages received from other LPs along with their timestamps. Messages which have been processed are saved in case a rollback is necessary and the messages need to be reprocessed.
- An *output queue* containing faithful copies (with a negative sign) of messages sent to other LPs during an event computation, called anti-messages. Anti-messages are stored to undo the effect of positive messages in the event of a rollback.

A Time Warp Logical Process (TWLP) is shown in Figure 2. Causality of executed events is achieved within *Time Warp* by two major components: the *local control mechanism* and the *global control mechanism.* The *local control mechanism* within *Time Warp* deals with the execution of events within a TWLP and receipt of messages from other TWLPs in order of their timestamps. The value of C_i at a TWLP never changes during the processing of an event, it only changes between events. However, in the absence of a global clock, it is likely that one TWLP will send a message to another TWLP with a timestamp that is in the past of the destination TWLP. Such a message is called a *straggler* message. In this case, the destination TWLP has to roll back to a state earlier than the timestamp of the *straggler* message, and then start processing all

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

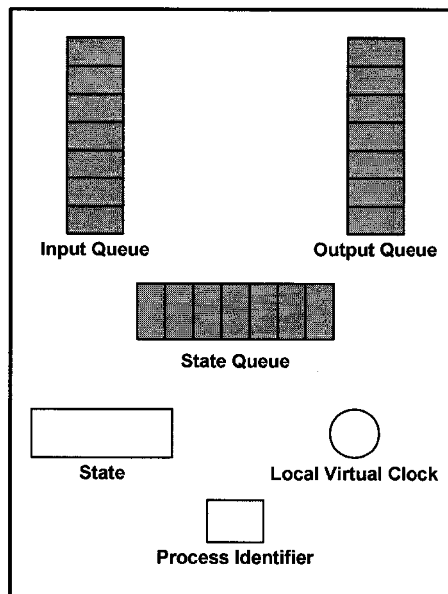


Figure 2. Time Warp Logical Process (TWLP)

events in its input queue that have timestamps greater than that of the *straggler* message. When a *straggler* message arrives, all events that have been processed might have modified state variables as well as scheduled new events by sending messages to other TWLPs. Therefore, to undo the incorrect modification of state variables, the state of the TWLP needs to be restored to a state prior to the incorrect modification. Various state saving techniques exist in literature [6 – 11].

It also becomes necessary to undo the effects of messages that may have been sent incorrectly because the straggler message was not processed. Therefore, for every message that a TWLP sends to another TWLP, the original message is stored in the *input queue* at the destination TWLP, and a faithful copy of the message with a negative sign, called an anti-message, is stored in the *output queue* of the source TWLP. This is shown in Figure 3. Whenever a message and an anti-message encounter each other in the same queue, they cancel out each other. This is known as *message annihilation.* Since

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

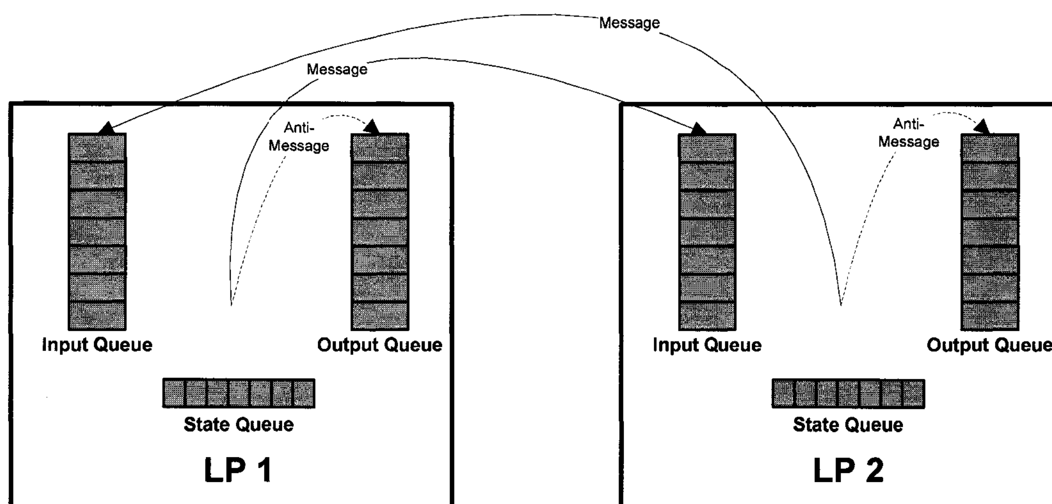


Figure 3. Communication between TWLPs

messages and anti-messages are created in pairs, the algebraic sum of all messages and anti-messages in a *Time Warp* system at any instant of time is zero.

The *local control mechanism* consumes more and more memory as the simulation progresses because states need to be saved for rollback purposes, and positive messages and anti-messages need to be saved even after they have been executed. A more global mechanism is required to reclaim memory allotted for history information. Moreover, certain event operations such as I/O operations cannot be rolled back easily, and the TWLP needs to be certain that no rollback is necessary for the current point before it proceeds with such operations. Therefore, a lower bound on the timestamp of any future rollback needs to be determined both for claiming memory allotted for state saving before the lower bound and for safely performing I/O operations in the past of the lower bound. This lower bound is known as the *Global Virtual Time* (GVT).

Fujimoto [1] describes *Global Virtual Time* at wallclock time T (GVT_T), as the minimum time stamp among all unprocessed and partially processed messages and anti-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

messages in the system at wallclock time *T*. GVT can be computed periodically or in emergency situations, when the simulation is running low on memory or an I/O or interrupt operation needs to be performed. Various techniques for computing the GVT exist in literature [12, 13, 14].

1.3 Related Work

This section describes past work that has been done in Parallel and Discrete-Event Simulation (PDES) and other related fields.

1.3.1 Parallel and Discrete-Event Simulation (PDES)

Research in the field of PDES is focused towards different aspects of the system. Simulation tools and languages have been developed to support building computer simulations. Tools have also been developed to help an application developer build a simulation more efficiently. To support simulation execution, many simulations executives and environments that support sequential and parallel discrete-event simulations have been developed. Simulation frameworks to support linking together simulations have also been defined. Most of these applications are focused towards distinct aspects of the field of PDES.

Object-oriented programming was originally developed with the basic intention of supporting the simulation of the states and activities of different entities [15, 16]. *Attributes* represent the state of an entity, while a *method* modifies the behavior of the entity by manipulating its attributes. A real-world system usually involves multiple entities interacting with each other through the lifetime of the system. Object-oriented

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

methodology aids in simulating this process by dynamically instantiating multiple objects to represent the different entities and representing the interaction among the objects as objects invoking methods on other objects. Commercial simulation languages such as MODSIM [17] and SIMSCRIPT [18] and simulation tools such as PROMODEL [19], ARENA [20] and MODELICA [21] have been developed to build simulations. Many of these languages are built using a specific general-purpose programming language and are focused on simplifying the development of stable simulation applications. Distributed simulation languages such as Maisie [22], MOOSE [23], APOSTLE [24], PARSEC [25], POMSim [26], Silk [27] and Sim++ [28] have also been developed before. Maisie and PARSEC are purely C-based simulation languages, Silk is Java-based, Sim++ is C++-based, while MOOSE is an object-oriented extension to Maisie. APOSTLE employs a single optimistic algorithm, the Breathing Time Buckets (BTB) synchronization algorithm. POMSim exclusively simulates manufacturing systems.

Considerable work has also been done to provide a formal framework for specifying and building application models. The DEVS formalism was conceived by Zeigler [29, 30] to provide a rigorous approach for discrete-event modeling and simulation. The DEVS formalism facilitates modular, hierarchical model specification, using a top-down approach to model development. DEVS allows for the description of system behavior at two levels: the basic model which can be used to develop larger models and how basic models are interconnected to form a complete system. Frey et al. [5] present a formal specification of the *Time Warp* paradigm, the structure of its components, the properties necessary for its validity and correctness, and a formal verification of the *Time Warp* specification. A reusable specification framework to extend

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the basic *Time Warp* specification and define and verify new properties is also described.

Simulation environments focus at the core simulation executive and support different aspects of the parallel and distributed simulation paradigm by providing the ability to use different algorithms and mechanisms. The Georgia Time Warp (GTW) [31] is a parallel simulation executive which implements the *Time Warp* specification for shared-memory multiprocessors. SPEEDES (Synchronous Parallel Environment for Emulation and Discrete-Event Simulation) [32] is an object-oriented distributed discrete-event simulation framework written in C++. SPEEDES supports a variety of conservative and optimistic synchronization schemes for distributed simulations and has been designed to run on a variety of hardware platforms. It also provides interfaces for developing external simulations to interact asynchronously with the SPEEDES simulation. However, SPEEDES is not a general purpose programming language nor does it provide a behavioral specification to develop new implementations of various components. It provides a modeling framework that allows other simulation scripting languages to be written on top.

Simulator Networking (SIMNET) [33, 34], developed by Defense Advanced Research Projects Agency (DARPA) provides an architecture for networking processors in real-time for combat simulation and wargaming. Distributed Interactive Simulation (DIS) [35] extends SIMNET by connecting diverse simulations using a communication medium to provide a common battlefield on which the different simulations can interact with each other in real-time. The High Level Architecture (HLA) [36, 37, 38] that evolved from DIS provides a specification of a common technical architecture to address the need for interoperability among new and existing simulations within the US

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Department of Defense. HLA defines a set of rules to link diverse existing simulations (each called a *federate*) while providing a framework to incorporate new simulations in order to create a collection of simulations operating together (called a *federation*). The Object Model Template (OMT) within HLA provides a standard format for describing information of common interest to more than one *federate*, while the Runtime Infrastructure (RTI) provides the means for *federates* to coordinate the execution and exchange of information.

1.3.2 Research / Educational Frameworks

A strength of the DIESEL model is that it provides a platform for further research, allowing new implementations to be developed and evaluated within the prescribed behavior. Similar work has been done in the field of operating systems to simulate a real operating system, while providing a platform for changing components within the simulation. NACHOS (Not Another Completely Heuristic Operating System) [39] developed at the University of Berkeley, California is used to teach operating systems at an undergraduate level. NACHOS allows the study and modification of an operating system by providing a simulation environment for an instructional operating system. It implements a CPU and device simulators, and simulates the general low-level facilities of typical machines, including file systems, threads, remote procedure calls, interrupts, virtual memory, and interrupt-driven device I/O. NACHOS allows students to write new implementations for each major component of an operating system, enabling them to understand aspects of real operating systems in terms of reliability, performance, and simplicity of the system.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Research frameworks have also been developed in the field of Management Information Systems (MIS) [40] for understanding and classifying existing research into different categories and for generating potential hypotheses for future research.

1.4 Chapter Plan

Chapter II explains the rationale behind this research and lists the requirements of the behavioral model. Chapter III provides a complete specification for the sequential simulation executive. It also includes a brief description of the implementation of the main components of the simulation executive. Chapter IV provides a complete specification for the distributed simulation executive. Chapter V describes various case studies that have been conducted to demonstrate the use of the behavioral model. Chapter VI concludes the dissertation and offers suggestions on improvements and additions to the model.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter II

PROBLEM STATEMENT

An architectural model defines the components of a system, specifies the manner in which they interact with one another, and details the interconnection among these components. The Distributed, Independent-Platform, Event-Driven Simulation Engine Library (DIESEL) is a behavioral model for a simulation executive that aims to provide a common platform for research and development in the field of Parallel and Discrete-Event Simulation (PDES). The model has been designed to be generic, language-independent and platform-independent using a hierarchical, object-oriented approach. The model defines how time management will be handled in the simulation. It also provides the application developer a tremendous amount of flexibility in the partitioning of his model from an event management standpoint. The model is defined behaviorally so that various algorithms and data structures can be implemented internally without requiring modification of applications that are developed using the model.

The rest of this chapter is organized as follows. Section 2.1 explains the rationale behind this research, Section 2.2 outlines the requirements of the architectural model, and Section 2.3 defines various terms used in this dissertation.

2.1 Problem Statement

DIESEL has been developed to allow research into the development of simulation executives, under a common research and development platform. It does not attempt to address simulation modeling or the associated formalisms as done in DEVS [29, 30].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

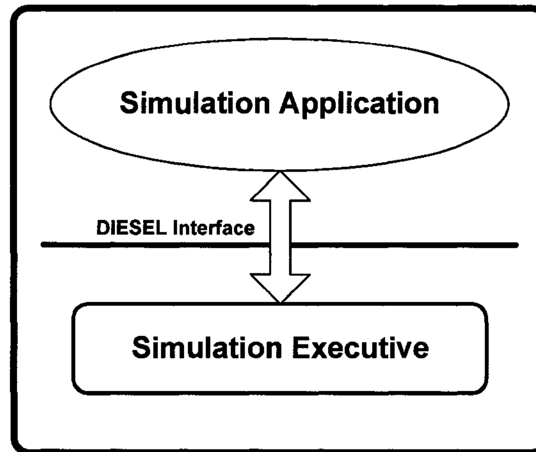


Figure 4. DIESEL Interface

DIESEL focuses on providing a software framework for algorithmic development to support time management, entity management, and other simulation management issues within the simulation executive. The DIESEL behavioral model is an abstraction of the simulation executive not previously found in literature.

DIESEL aims to provide a standard Application Programming Interface (API) as shown in Figure 4 for a simulation executive, capable of supporting both a sequential and distributed DES. The DIESEL API separates the development of applications from the underlying structure of the core simulation executive. The executive manages the advancement of simulation time within the application for the duration of the simulation. The application schedules events to be executed in the simulation future and the executive executes the events at the appropriate simulation time. A different implementation can be developed for the DIESEL API, without affecting any applications that are built using the API. The DIESEL interface provides an API for developing both sequential and distributed simulation applications. Any component implementing the DIESEL interface can be modified or changed as long as it does not

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

modify the expected behavior of the interface.

The DIESEL interface described in Chapters III and IV provide various structures for developing applications. The interface also manages simulation time for both a sequential and distributed application. Events are executed in increasing order of their timestamps to satisfy the *Local Causality Constraint*. The interface also provides a mechanism to deal with simultaneous events, i.e. events with the same timestamp. This mechanism allows the next state of the simulation to be computed totally based on the current state of the simulation, and not on any intermediate computations while transitioning from the current state to the next state. The interface however, does not address race conditions between simultaneous events modifying the same state variable. Resolving these race conditions is expected to be the responsibility of an application developer or a specific implementation of the DIESEL interface.

2.2 Research Purpose

The research described in Chapter I addresses different distinct issues in the field of PDES both at the application development level and the core simulation executive level. SPEEDES provides a framework for developing discrete-event simulations using the C++ programming language and provides interfaces to link external simulations written in C++, to itself. However, SPEEDES does not specify a behavioral model or interfaces to support the development of alternative implementations on different platforms. HLA provides both a behavioral specification and an interface specification to support simulations using different languages like C++, Ada, Java, and CORBA among others. The interface specification, however, requires the application to have a high level

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of knowledge of the underlying time management. The specification requires each simulation to know its own internal time management and how it will interact with other simulations employing different time management strategies. HLA focuses on linking existing simulations rather than partitioning a new simulation model into components for parallelism.

At the core simulation executive level, DIESEL proposes to provide interfaces to support development of different implementations of the behavioral model thereby allowing different implementations to interact with each other. It would support conservative synchronization algorithms, as well as optimistic synchronization algorithms such as *Time Warp*. DIESEL does not attempt to mix synchronization algorithms within a single simulation, but rather proposes to create a research environment to study simulation behavior and interaction while utilizing a single time management paradigm and communication mechanism for the whole simulation.

At an application level, DIESEL proposes to provide an application developer the ability to partition the model in different ways and map different components of the model to the processors available for the simulation.

DIESEL provides a common, standardized interface for developing applications, for both sequential and distributed discrete-event simulations. A common interface is specified for both a sequential and distributed simulation executive. DIESEL also provides a fair amount of flexibility to the developer to partition the simulation model into components in terms of event handling. The developer can employ a central event management scheme to handle all events generated within the simulation, group together similar components, and handle their events together, or designate that each object within

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the simulation handles its own events. For research purposes, this varies the size of the event list and can aid in the development of appropriate strategies to manage them.

DIESEL does not specify a particular algorithm or strategy to implement different components of the behavioral model. This is extremely useful to:

- Compare application behavior while partitioning the model in different ways.
- Compare application behavior on a sequential and distributed executive.
- Compare existing event insertion and management algorithms on a common platform.
- Develop and compare new event management algorithms under the same platform.
- Compare different synchronization algorithms (conservative and optimistic) for a distributed simulation executive.
- Verify simulation behavior while employing a new synchronization algorithm.
- Compare different communication mechanisms for a distributed executive.

The only constraints imposed on any employed algorithm or mechanism is that it satisfy the expected behavior of that component in the behavioral model and not modify the interconnections among components. Moreover, DIESEL functionality is defined using a behavioral model and is generic. It defines the expected behavior of the simulation executive and is not focused on any platform or programming language. DIESEL can be implemented using any programming language suitable for a particular research or commercial purpose.

The DIESEL behavioral model has been primarily developed as a research tool to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

aid in comparing existing solutions while also providing a great level of support for research into implementation strategies for different components of both a sequential and distributed simulation executive. Implementations are provided for all components specified in the behavioral model, with each component implementation managed separately and independently in order to isolate errors and also to encourage further development. Implementations can be developed for specific components within the model with no change to other components, thus ensuring that the complete system continues to work. New implementations can also be developed for the whole model using various programming languages and on different hardware platforms.

A stable environment and implementation for the DIESEL behavioral model is also provided for developing commercial applications, even though DIESEL has been primarily developed as a research tool.

2.3 Model Requirements

The architectural model for DIESEL has been developed with the intention of satisfying the following basic requirements:

- ***Support for Discrete-Event Simulation*:** DIESEL should be capable of supporting discrete-event simulations.
- ***Support for a Sequential and Distributed Simulation Executive*:** DIESEL should support both a sequential and distributed simulation executive, with the same interface to the programming world.
- ***Generic Design*:** The DIESEL behavioral model should be as generic as possible

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

with simple interfaces between DIESEL and the outside world, as well as within DIESEL itself.

- ***Object-Oriented Design***: The DIESEL architecture should be defined to be object-oriented and developed using a hierarchical, top-down approach.
- ***Language and Platform Independent***: The DIESEL behavioral model should not be tailored to any particular language or platform for implementation. It is a research goal to implement the model in different languages on various platforms to provide proof of its generic design.

2.4 Definitions

The following terms are used in this document to describe various components of the DIESEL behavioral model:

- ***SimObject:*** A *SimObject* is any object created and used within a simulation that needs to pass simulation time during the simulation.
- ***SimulationCluster:*** A *SimulationCluster* is a set of *SimObjects* interacting with a single *SimulationEngine*.
- ***SimulationEngine:*** A *SimulationEngine* handles all the time management for the entire duration of a simulation. It keeps track of all simulation objects within a simulation and controls the execution of events by the respective objects.
- ***SimulationEngineComponent:*** A *SimulationEngineComponent* handles the scheduling and execution of events on an individual *SimObject*. All interaction between the application layer (*SimObjects)* and the *SimulationEngine* are performed through the *SimulationEngineComponent* as a communication layer.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Every *SimObject* should be associated with a *SimulationEngineComponent*.

- ***Interrupt:*** An interrupt is used to stop the execution of a particular event method within a *SimObject* before its scheduled execution time. The event is removed from the list of events scheduled to be executed. If an interrupt routine has been defined for the particular event, then that routine is executed.
- ***Delegate:*** A *Delegate* is the primary method of scheduling events on different *SimObjects* within this architecture. It encapsulates all the information required to make a method call for an event at the scheduled time.
- ***SimulationForest:*** A *SimulationForest* is a set of *SimulationClusters*, with the *SimulationEngine* within each *SimulationCluster* interacting with *SimulationEngines* of other *SimulationClusters* to manage simulation time.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter III

INTERFACE SPECIFICATION FOR THE DIESEL SEQUENTIAL MODEL

This chapter provides a complete description of the sequential component of the DIESEL behavioral model. Section 3.1 describes the interaction between different components within the DIESEL architectural model. Sections 3.2 to 3.9 provide the specification for various interfaces for the DIESEL sequential model. The names for different attributes and methods within an interface or class definition are based on recommended guidelines [41]. Although these guidelines are primarily for standard C++ programming, the same guidelines have been used here.

The complete DIESEL behavioral model has been implemented in C++ and is fully operational. All the structures described in this chapter have been implemented and extensively tested. Section 3.10 describes the implementation classes for the DIESEL Engine interface and also discusses event management within DIESEL. Section 3.11 discusses examples to demonstrate the operation and correctness of the behavioral model.

3.1 Interaction between components within DIESEL

A simulation involves different *SimObjects* interacting with the simulation executive for scheduling and executing events. The simulation executive for a sequential simulation has a single global *SimulationCluster* controlling the simulation. A *SimObject* interacts with the *SimulationCluster* in a controlled manner through the DIESEL Interface as shown in Figure 5. The DIESEL Interface requires that each

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

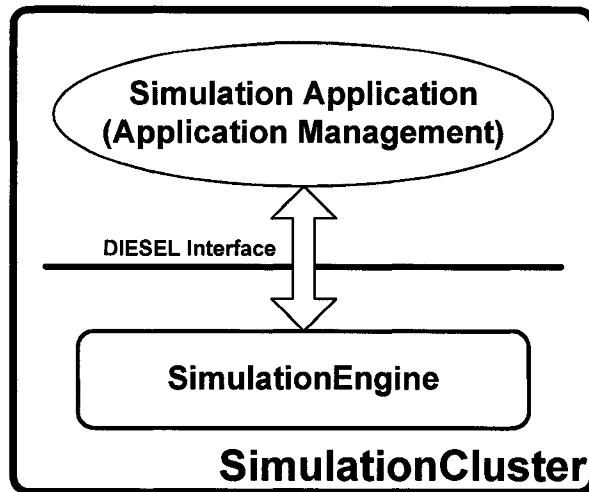


Figure 5. *SimulationCluster*

SimulationEngineComponent should be registered with the *SimulationEngine* when it is initialized. It also requires that every *SimObject* be associated with a single *SimulationEngineComponent*. A *SimulationEngineComponent* handles the scheduling and execution of events on an individual *SimObject*. All interaction between the application layer (*SimObjects*) and the *SimulationEngine* are performed through the *SimulationEngineComponent* as a communication layer.

A *SimObject* can be associated with a *SimulationEngineComponent* in different ways allowing an application developer a considerable level of freedom to partition the simulation.

- A *SimObject* can inherit a *SimulationEngineComponent* as shown in Figure 6. This results in each *SimObject* having its own *SimulationEngineComponent*.
- A single global *SimulationEngineComponent* can exist for all *SimObjects* in the simulation as shown in Figure 7.
- A single static *SimulationEngineComponent* can exist for all *SimObjects*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

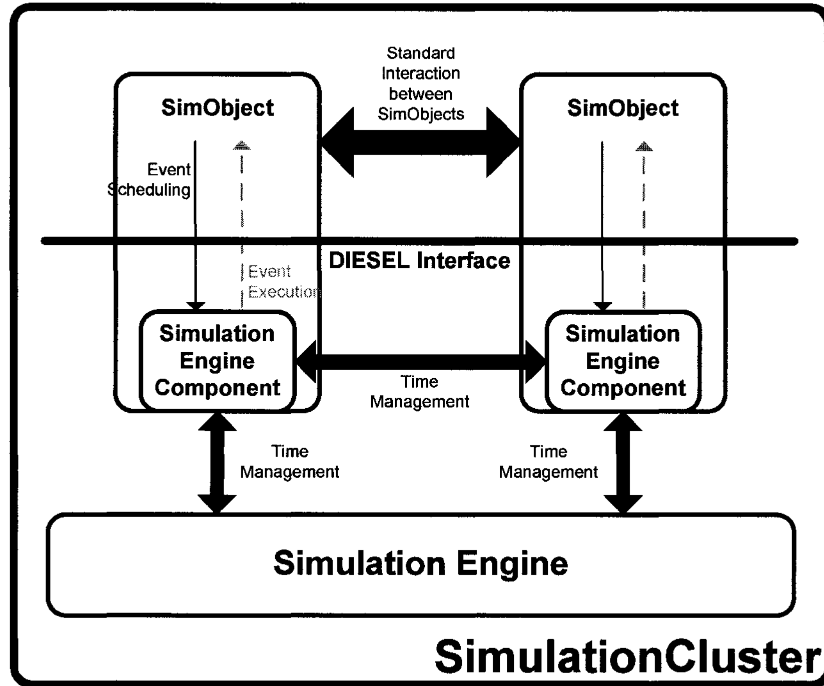


Figure 6. Inherited *SimulationEngineComponent*

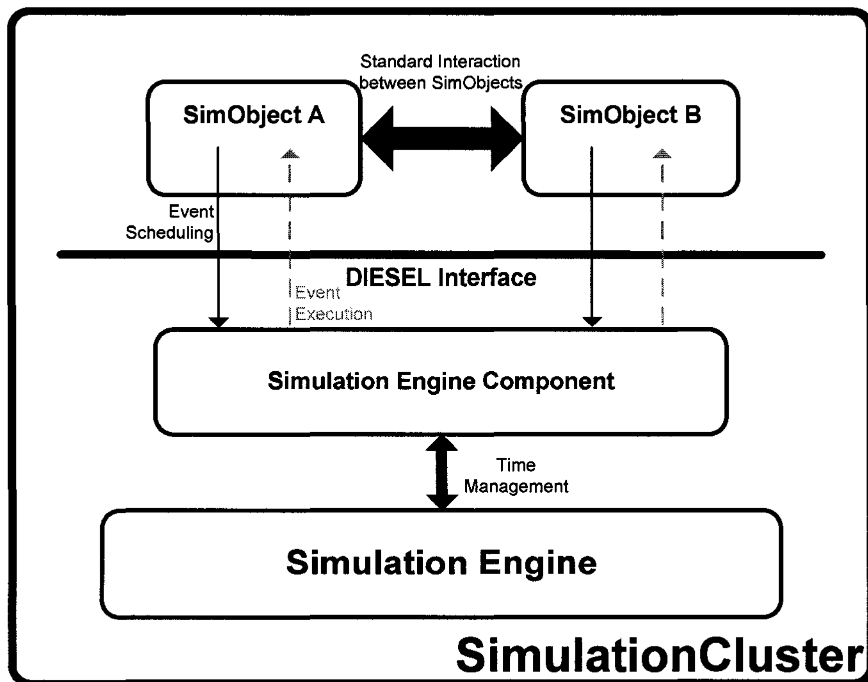


Figure 7. Global *SimulationEngineComponent*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

belonging to a particular class as shown in Figure 8. All object instantiations of a class will in this case, share a common *SimulationEngineComponent*.

- *SimulationEngineComponents* can be dynamically created during simulation execution and *SimObjects* existing within the simulation can be partitioned across the created *SimulationEngineComponents* as shown in Figure 9. This provides a high level of flexibility for an application to define the model partitioning.

The different ways that a *SimObject* can be associated with a *SimulationEngineComponent* enables application developers to partition events that are expected to be generated within the simulation, in any suitable manner. A single global *SimulationEngineComponent* implies that all generated events are stored in a single event list even if they are to be executed on different *SimObjects*. In this case, a single *SimulationEngineComponent* is registered with the *SimulationEngine*, and the *SimulationEngine* calls it continuously to execute events in order of their timestamps.

Alternatively, each *SimObject* inheriting a separate *SimulationEngineComponent* implies that only events scheduled on that *SimObject* will be stored on that *SimulationEngineComponent*. This means that multiple event lists will exist within the simulation, with the number of events being distributed across the event lists. Moreover, the number of *SimulationEngineComponents* registered with the *SimulationEngine* depends on the number of *SimObjects* within the simulation. This reduces the overhead related with the number of events within each event list for any event management strategy compared to a central list with a single global *SimulationEngineComponent*, while increasing the overhead within a *SimulationEngine* to manage the increased number of *SimulationEngineComponents*.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

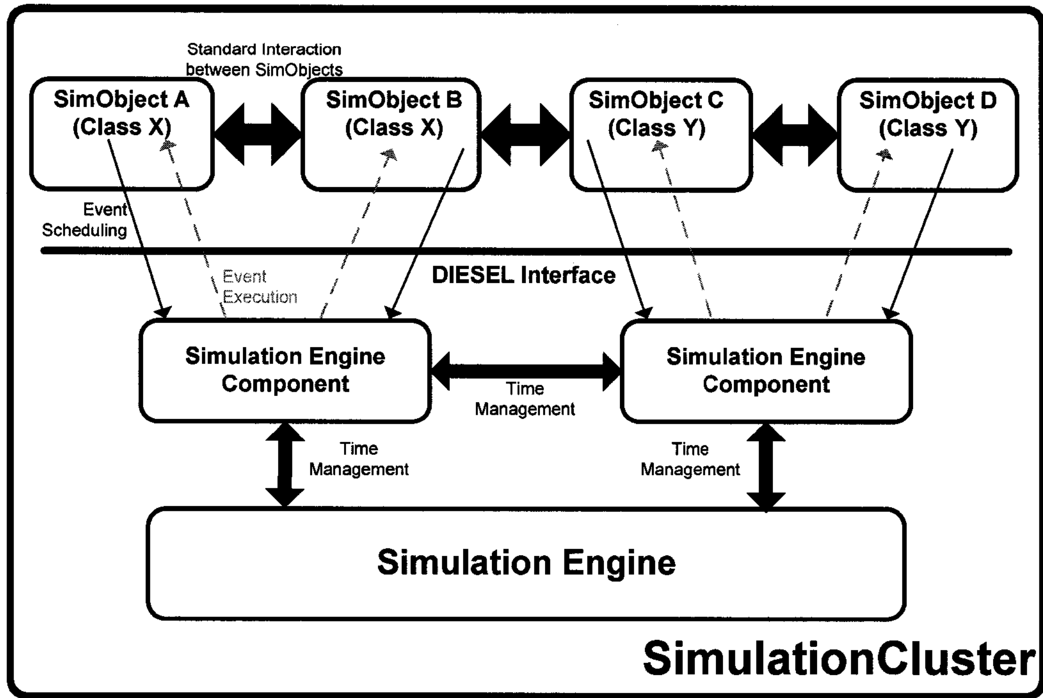


Figure 8. Static *SimulationEngineComponent*

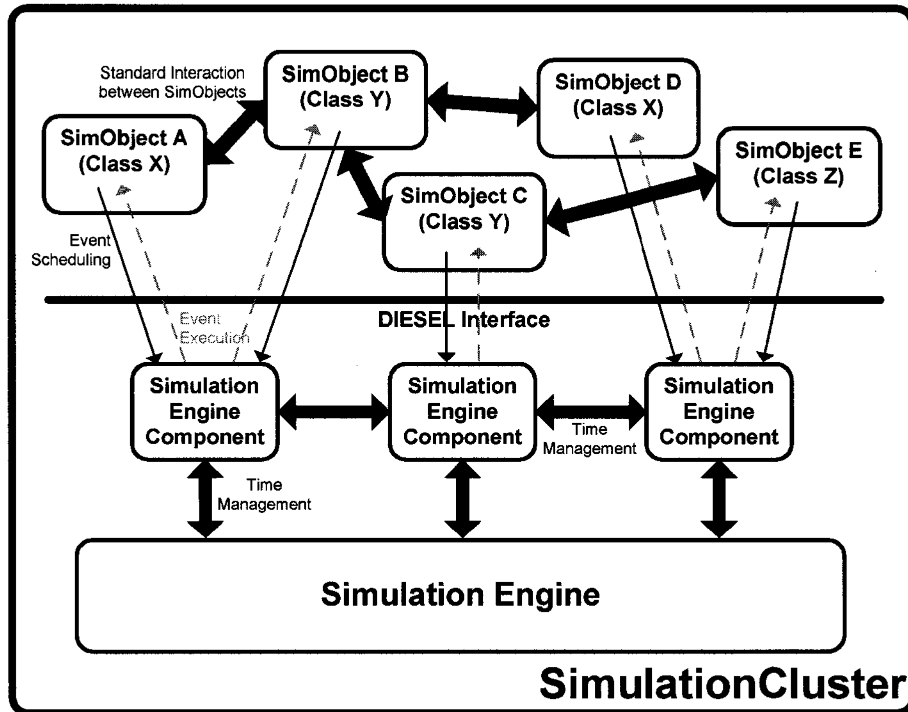


Figure 9. Dynamic *SimulationEngineComponents*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

There is no particular association between a *SimObject* and a *SimulationEngineComponent* described earlier, that performs significantly better than any other type for all simulations and, therefore, can be recommended as the best possible association. The most likely scenario in any simulation would be the existence of all the types of associations described earlier, with the application developer choosing one that suits his purpose the best for different model components.

A *SimObject* can schedule events to be executed on itself or on other *SimObjects*. Events are always scheduled on the *SimulationEngineComponent* associated with the *SimObject* on which the event is to be executed.

3.2 DIESEL Engine Interfaces

This section defines interfaces to introduce initial *SimObjects* into the simulation, to schedule events to be executed on those *SimObjects* and to manage arguments to be used when the events are executed.

3.2.1 *Delegate* Interface

A *Delegate* is the primary method of scheduling events on different *SimObjects* within this architecture. A *Delegate* is the information required to execute an event at a future point in wallclock time. Since an event is not directly invoked but scheduled to be executed, sufficient information needs to be encapsulated within the *Delegate* to make a method call for an event at a later point in wallclock time. A *Delegate* is stored in the event list of the *SimulationEngineComponent* associated with the *SimObject* when it is scheduled. The *Delegate* is later invoked at its scheduled simulation time. The execution

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of a *Delegate* can be cancelled before its execution. When an event is scheduled, an interrupt routine can be specified along with a set of arguments to be used for the interrupt routine. This interrupt routine is executed if and when the execution of the *Delegate* is cancelled. A *Delegate* holds the following information:

- The *SimObject* on which the event method should be executed.
- The event method to be executed by the *SimObject*.
- The list of arguments to be used during the execution of the method at its scheduled execution time.
- The interrupt routine to be executed if the execution of the event method is interrupted before its scheduled execution time. *(Optional)*
- The list of arguments to be used during the execution of the interrupt method. *(Optional)*
- Priority of execution of event method relative to other event methods scheduled at the same simulation time. *(Optional)*

All *Delegates* are scheduled to be executed with next-to-highest priority among events scheduled on a *SimulationEngineComponent* at the same simulation time, unless a priority is specified for the *Delegate*. The highest priority for execution of events is reserved for DIESEL. A priority can be specified for a *Delegate* to lower the priority of the event relative to other *Delegates* scheduled at the same simulation time. If two *Delegates* have the same priority, then the *Delegates* are executed in first-come-first-served order.

The *Delegate* interface is shown in Figure 10. The *StateSave* and *Replicable* interfaces, which the *Delegate* interface inherits, are discussed in Sections 3.4 and 3.5.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

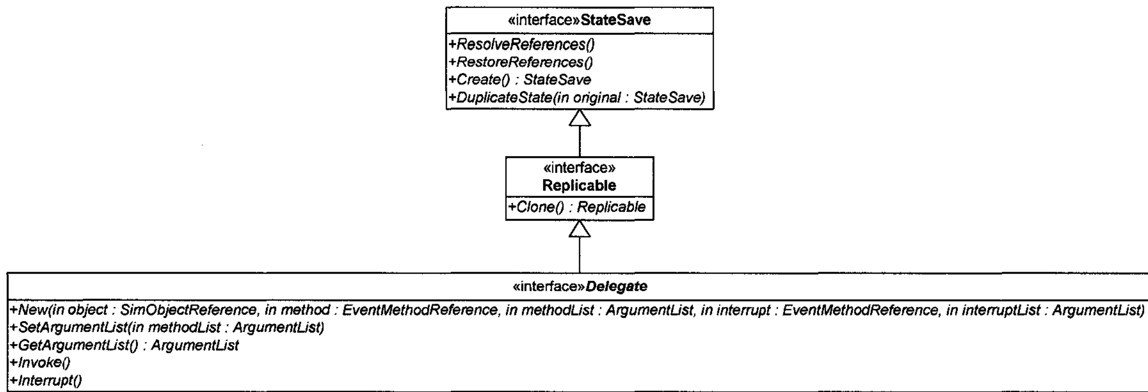


Figure 10. *Delegate* Interface

The "Invoke" method executes the event method within a *Delegate* with the specified list of arguments. The "Interrupt" method cancels the event method invocation within a *Delegate*, with the interrupt method being executed using the specified list of arguments for the interrupt.

3.2.2 *ArgumentList* Interface

Argument lists manage the arguments to be used when a scheduled event is executed by invoking a method within a *SimObject*. DIESEL supports three types of argument lists: ARGUMENT_FIFO, ARGUMENT_LIFO and ARGUMENT_INDEXED, with a single type allowed to be used with a particular argument list. The behavior of an argument list is defined by its type. Once a type is defined, arguments must be entered and retrieved in the same order. An argument list of type ARGUMENT_FIFO or ARGUMENT_LIFO does not require an index to be specified when arguments are added or accessed from the list. The arguments are removed in first-in-first-out manner for an ARGUMENT_FIFO list, and in last-in-first-out manner for an ARGUMENT_LIFO list. In the case of an ARGUMENT_INDEXED

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

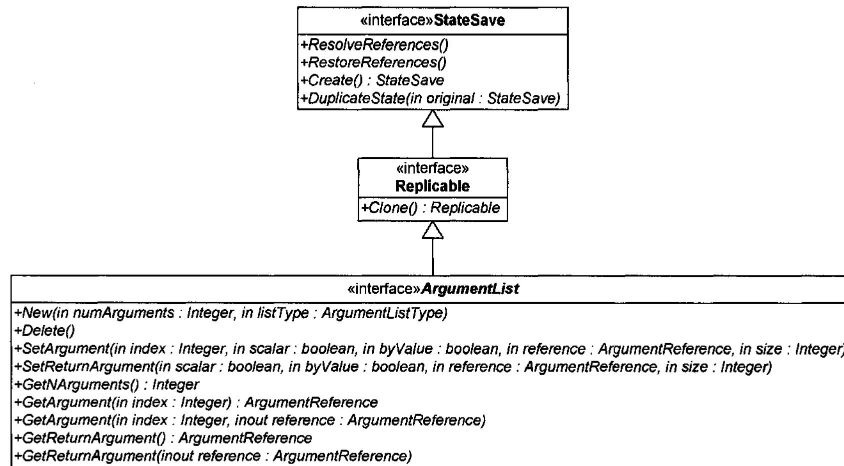


Figure 11. *ArgumentList* Interface

list, an index needs to be specified each time an argument is added or accessed from the list. An ARGUMENT_INDEXED list can be accessed in any arbitrary order.

Arguments are provided either by reference in which a pointer is passed or by value where a pointer is again passed, but the value is copied to an internal data structure. Arguments that are passed by value while scheduling a method need to be accessed by value when the method is executed. The same is true for arguments passed by reference. A single return value can be assigned to the argument list before the list is used during the execution of the event method.

The *ArgumentList* interface is shown in Figure 11. The "SetArgument" and "GetArgument" methods are used to store and retrieve arguments within the argument list, while the "SetReturnArgument" and "GetReturnArgument" methods are used to access the return value within the list.

A drawback of the *ArgumentList* interface is that is stores no type information for an argument, just its size and a reference to it. The result is that there exists no type checking on the arguments when the arguments are used during *Delegate* invocation,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

since the arguments are treated as a block of memory identified by a reference to the argument and the size of memory that the argument occupies. The type checking is assumed to be done by the application, when the arguments are retrieved by the application. The DIESEL behavioral model does not explicitly require support for type checking. This is a common approach in parallel programming languages [42] since it is difficult, if not impossible, to enforce type checking of arguments across networked computers. However, the lack of an explicit specification for type checking within the model does not preclude any implementations from using strong type checking on the arguments. An extension to the interface would be to store the type information of the argument when it is added to the *ArgumentList*, similar to the C# programming language. The type of the argument that would be application based can then be used to perform type checking when the arguments are stored in or retrieved from the *ArgumentList*.

3.2.3 *SimulationEngineComponent* Interface

The *SimulationEngineComponent* interface defines scheduling *Delegates* within the simulation. Each *SimulationEngineComponent* maintains its own list of *Delegates* to be executed on the associated *SimObjects*. The *SimulationEngineComponent* interface is shown in Figure 12. The "ScheduleEventAtTime" method is used to schedule a particular method in a *SimObject* at a specific simulation time, while the "ScheduleEventInTime" method is used to schedule a particular method in a *SimObject* after a given simulation time has elapsed. The "GetAveragePendingEventSetSize" and "GetAverageEventWaitTime" methods are used to gather various statistics associated with the event list within the *SimulationEngineComponent*.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

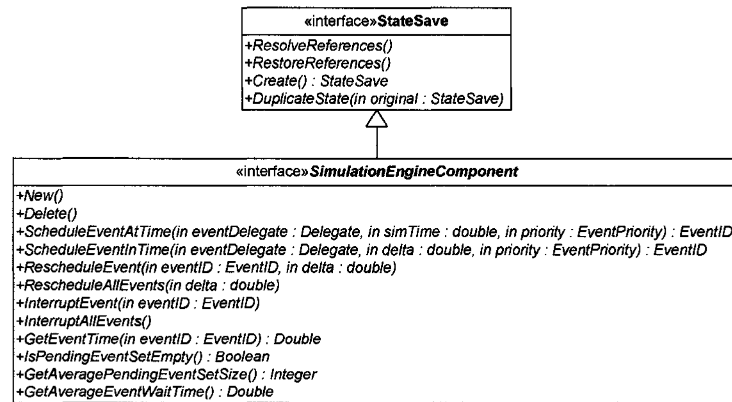


Figure 12. *SimulationEngineComponent* Interface

3.3 DIESEL Delayed State Commitment Interfaces

An important component of the DIESEL model is the *Delayed State Commitment* for attributes of objects within a simulation. If one event modifies a state variable and then a second event executed at the same simulation time uses the same state variable, then the order in which the events are executed determines the final state of the state variable after that simulation time. The result of the second event can be different depending on whether the first event has been executed or not. For example,

a is an event that assigns the value of variable "y" to "x" i.e. a: $x \leftarrow y$

b is an event that assigns the value of variable "z" to "y" i.e. b: $y \leftarrow z$

If a and b are both scheduled for execution at simulation time t and a is executed before b, then the values of x and y after both events have been executed, i.e. at simulation time (t+δ) are:

$x = y$ and $y = z$.

If a is executed after b, then the values of x and y at simulation time (t+δ) are:

$y = z$ and $x = z$

One solution to the problem is to define the current state (CS) to be the set of state

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

variable values at a time ($t-\delta$) immediately prior to the current simulation time t. Then the next state (NS) is defined as the resulting set of state variable values at a time immediately ($t+\delta$) after the current simulation time t. The *Delayed State Commitment* presented here defines the NS state variable values (SV(NS)) to be solely a function of the CS state variable values (SV(CS)),

$$SV(NS) = f(SV(CS))$$

and not a function of the order of computation of events at simulation time t.

Thus, the *Delayed State Commitment* ensures a more consistent behavior, by making all events act on the same value of the state variable and updating the state variable with its new value after all events have finished executing at that simulation time. *Delayed State Commitment* makes the next state of a simulation truly dependent on its current state, instead of the transition from the current state to the next state. An attribute can be initialized as a *DelayedCommit* attribute instead of a standard type like Integer, Double, Boolean, etc. with an initial value. When the value of the *DelayedCommit* attribute is changed, it is not updated immediately. The *DelayedCommit* attribute is added to a list maintained by the simulation executive. After all events scheduled to be executed at the current simulation time have been executed by their respective *SimObjects*, the simulation executive updates the value of every *DelayedCommit* attribute within its list with its new value. If there are multiple updates to a *DelayedCommit* attribute at the same simulation time, the last update to the attribute has permanence beyond that simulation time.

Delayed State Commitment does not resolve race conditions when two events modify the same state variable at the same simulation time. DIESEL currently does not

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

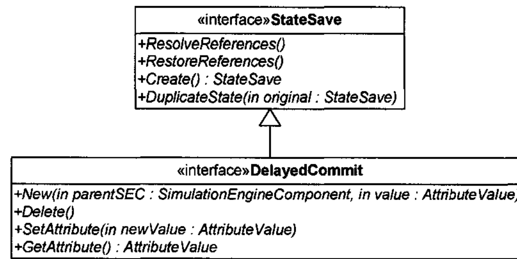


Figure 13. *DelayedCommit* Interface

support the breaking of ties to resolve which event to execute first at a particular simulation time. Race conditions can be resolved by an application developer by lowering the priority of one event with respect to another event scheduled at the same simulation time when scheduling the delegate.

The *DelayedCommit* interface is shown in Figure 13. The "SetAttribute" and "GetAttribute" methods store and retrieve values of the *DelayedCommit* attribute.

3.4 DIESEL State Capture and Restore Interfaces

DIESEL has the ability to capture the state of the simulation at a desired simulation time. DIESEL can then resume execution of the simulation from that saved state at a later time. This ability to save the simulation state is an important tool for simulation analysis at a state other than its final state.

Every interface and class defined within DIESEL inherits the *StateSave* base interface shown in Figure 14. It is also required that every class or interface defined within an application should inherit the *StateSave* interface. The *StateSave* interface should be the first class in the inheritance tree of any user-defined class or interface. During a state capture operation, DIESEL treats every object within the simulation as a *StateSave* object and calls its "ResolveReferences" method to resolve references for

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

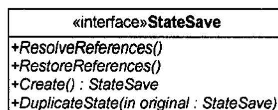


Figure 14. *StateSave* Interface

dynamically initialized attributes within the object. The "RestoreReferences" method restores these same references within an object during a state restore operation. The "Create" method creates an empty shell for a new object, and the "DuplicateState" copies the state of the original entity to the newly created entity during a state duplication process for a non-terminating state save operation. These methods should be defined for each class or interface that inherits the *StateSave* interface.

A state capture operation can be scheduled at an offset from the current simulation time, similar to a regular event, before the start of the simulation or at any point during simulation execution. When a state capture event is executed, all references within the simulation must be resolved so that they can be later restored. DIESEL resolves all references within the simulation executive, by calling "ResolveReferences" for all of its components. DIESEL then calls "ResolveReferences" for all entities within the application to resolve their references. After all references have been resolved, DIESEL writes the entire state of the simulation to an output file.

DIESEL has the ability to perform two types of state capture: *terminating* and *non-terminating*. A *terminating* state capture terminates the simulation after the state of the simulation has been captured and saved. A non-terminating state capture is identical to checkpointing the state of the simulation. During a *non-terminating* state capture, the entire simulation is duplicated by creating a clone for the simulation. Since references are resolved in place, a clone is created so the references can be resolved for the clone. The

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

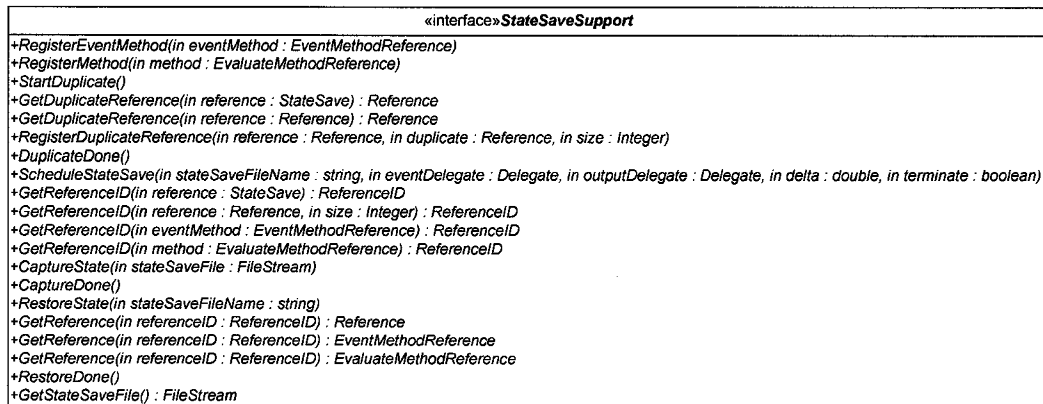


Figure 15. *StateSaveSupport* Interface

state of the simulation is captured by saving the state of the clone, while the original simulation can continue execution since its references are still valid. Once the state of the simulation has been saved, the clone is disposed and the simulation continues execution.

The *StateSaveSupport* interface shown in Figure 15 provides various procedures for the simulation application to access the state save support infrastructure for both a state capture and state restore operation. Every event method within the application should be registered with the state save support infrastructure using the "RegisterEventMethod" procedure. The "GetReferenceID" method registers an object with the state save support infrastructure as well as retrieves the ID for a registered event method. The "GetReference" method restores the reference for an object registered with the state save support infrastructure as well as retrieves a registered event method.

3.5 DIESEL Base Interfaces

DIESEL defines base interfaces to support cloning of objects and programmatically changing the properties of an object during simulation execution. Each object that needs to be cloned should inherit the *Replicable* interface shown in Figure 16.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

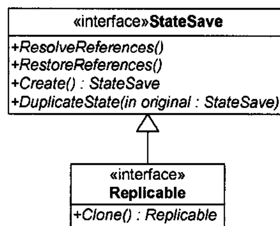


Figure 16. *Replicable* Interface

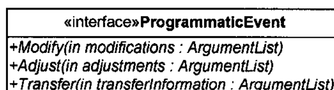


Figure 17. *ProgrammaticEvent* Interface

The "Clone" method creates a clone of the object and should be defined for every object that inherits the *Replicable* interface.

The *ProgrammaticEvent* interface shown in Figure 17 allows a user to schedule the modification or adjustment of the attributes of an object within a simulation. The "Modify" method modifies the attributes of an object, the "Adjust" method adjusts the attributes by a delta value (positive or negative), and the "Transfer" method copies properties of one object to another object. All these methods should be defined for every object that inherits the *ProgrammaticEvent* interface.

3.6 DIESEL Random Variate Model Interfaces

Any stochastic process has a certain amount of uncertainty associated with it. This uncertainty or randomness is introduced in a simulation through random numbers. Characteristics such as unknown delay times, process times, and inter-arrival times are often represented in a program in the form of random numbers. These random numbers

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

are generated using a random number generator. The generated random stream is not actually random in nature, but rather derived using a mathematical formula. The random stream is therefore said to be generating pseudo-random numbers. A random number generator consists of an algorithm to generate independent, identically distributed (IID) random numbers from the continuous distribution Uniform (0, 1). A good random number generator should have a long cycle length, so that it would be highly unlikely that the pseudo-random numbers would be repeated with a simulation. A good random number generator should also have long sub-streams (sub-segments of the main random stream) for variance reduction.

Any existing random number generator can be used to generated random numbers from the Uniform distribution (0, 1) [43, 44]. Random numbers from other distributions namely, Triangular, Unit, Normal Gamma, Beta, Weibull, etc, are derived by performing computations on the random number obtained from the random number generator with the computed random number still remaining random [45].

DIESEL supports complex distributions including the ability for a random number requested from a particular distribution during the execution of one event to be dependent on a previous request to the same distribution during the execution of a previous event. The capabilities include:

- ***Single mode distributions:*** A wide variety of distributions is available including Constant, Unit, Exponential, Triangular, Normal, Beta and Weibull.
- ***Multimodal distributions:*** Multimodal distributions support situations where with one probability, a given distribution is employed, and with another probability, a different distribution is employed.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

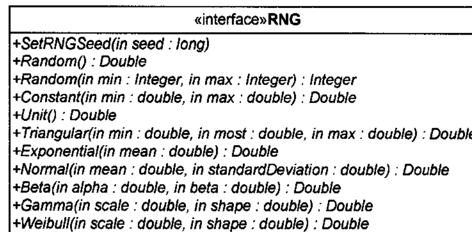


Figure 18. *RNG* Interface

- ***Ordered distributions:*** Ordered distributions guarantee that subsequent events cannot get out of order by preventing the next event from completing prior to the current event.
- ***Blocking distributions:*** A blocking distribution is a multimodal distribution where once a given mode is entered, all following calls will remain in that mode for a period defined by another distribution.

The *RNG* interface shown in Figure 18 encapsulates the functionality to generate integer random numbers, floating-point random numbers, and random numbers from specific distributions within DIESEL. The *AdvancedDistributions* interface is used to generate complex distributions within DIESEL, and supports the following modes of operation:

- ***Multimodal:*** This mode essentially provides a combination of two or more simple distributions that cannot be represented using a single simple distribution. An associated probability is used to choose between then different distributions.
- ***Ordered:*** The entities that arrive for a certain process complete the process and leave in a first-in-first-out order. No entity can complete its processing before a preceding entity, in this mode.
- ***Blocked:*** In any process, if an entity were delayed in its processing time, it would

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

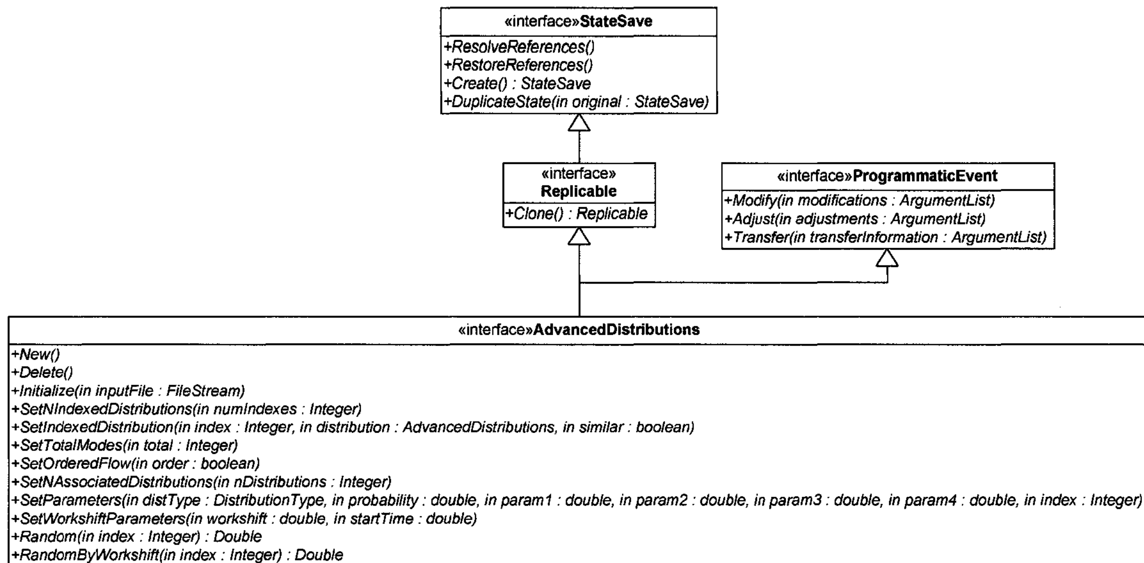


Figure 19. *AdvancedDistributions* Interface

have a delaying effect on entities that follow. The delayed entity is a special case in the process and would be represented by a certain mode of the process.

- ***Indexed:*** Two or more distributions can be used to specify a process. An associated index is used to choose between the different distributions.
- ***Work Shift Adjusted:*** A random number obtained from a distribution can be adjusted by a work shift for a day within the simulation ($0 < \text{work shift} \leq 24$). A start time can also be specified for a work day within the simulation.

The *AdvancedDistributions* interface is shown in Figure 19. The interface defines various methods to define a complex distribution and each distribution within a complex distribution. It also defines methods to generate random numbers from the specified distribution which can also be adjusted by a work shift time.

3.7 DIESEL Synchronization Interfaces

Certain processes within a simulation need to wait until some specified event or

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

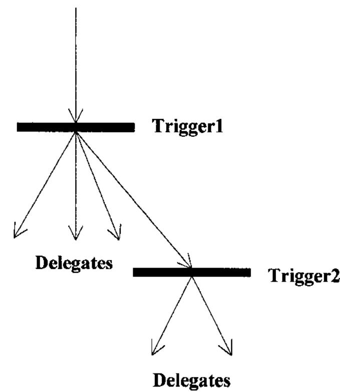


Figure 20. Dependent Triggers

another process has occurred. The time that the process needs to wait within the simulation is mostly unspecified; only the event that needs to occur for the process to proceed is specified. This is similar to a Petri network where input and output functions specify the transition to a finite set of places within the network. Triggers and joins are used within DIESEL to achieve the purpose of arbitrary synchronization within a simulation.

During a simulation, *Delegates* that need to wait for another event to be executed before being scheduled for execution are added to a trigger. When the event is executed and the trigger is fired, the trigger schedules the immediate execution of all *Delegates* waiting for the specified event to be executed on their associated *SimObjects*. A trigger can also have a collection of other triggers waiting for the parent trigger to fire as shown in Figure 20. DIESEL supports two types of triggers:

- *Trigger:* A *Trigger* can be set up to trigger an arbitrary number of *Delegates* and an arbitrary number of other *Triggers*. When a *Trigger* is triggered, it triggers all *Delegates* and other *Triggers* that have been registered with it until that instant. The *Trigger* interface is shown in Figure 21.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

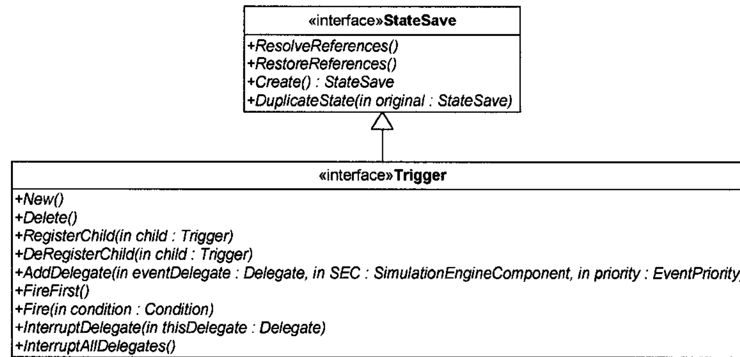


Figure 21. *Trigger* Interface

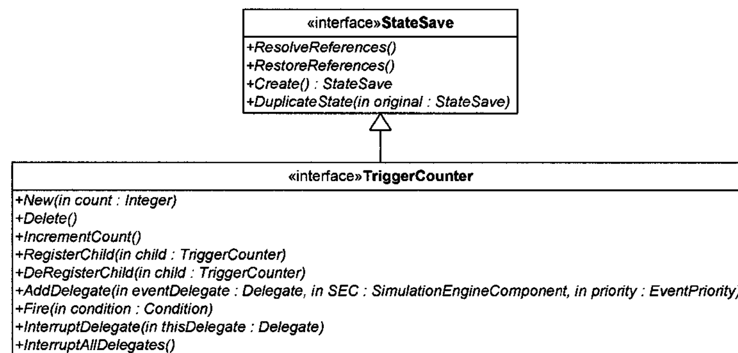


Figure 22. *TriggerCounter* Interface

- *TriggerCounter:* A *TriggerCounter* can be set up to trigger a certain specified number of *Delegates*. When a *TriggerCounter* is triggered, it triggers all *Delegates* and other *TriggerCounters* that have been registered with it. If not all the specified number of *Delegates* have been added to it when it is triggered, all *Delegates* added to it after it has been triggered are immediately executed. The *TriggerCounter* interface is shown in Figure 22.

Certain processes within a simulation need to wait for multiple events to be executed before proceeding further. This is an extension of triggers where the process waits for a single event. DIESEL provides this capability by allowing multiple events to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

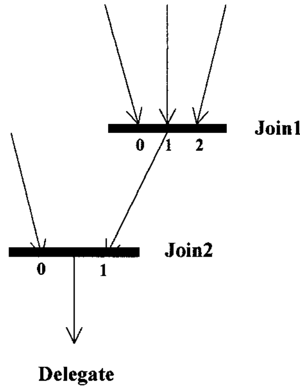


Figure 23. Dependent Joins

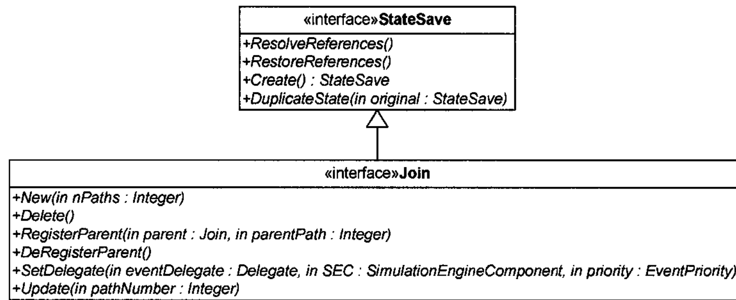


Figure 24. *Join* Interface

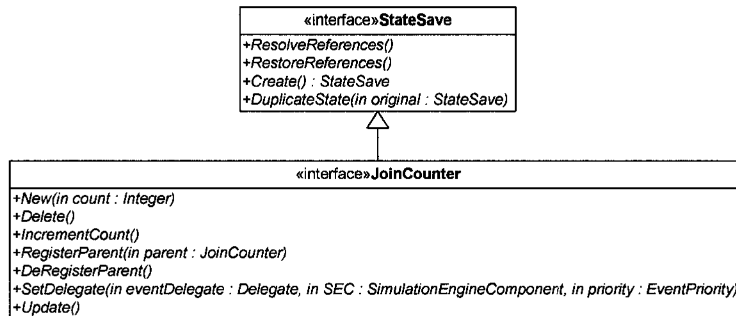


Figure 25. *JoinCounter* Interface

control the execution of a single event method. The event method is executed only when all of the events controlling it have been executed. A join can also be set up with another

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

join representing one of the events to be executed as shown in Figure 23. There are two types of joins within the DIESEL architecture:

- *Join:* A *Join* is set up to wait for specific events to be satisfied. The *Join* interface is shown in Figure 24.
- *JoinCounter:* A *JoinCounter* is set up to wait for just a number of events to be executed. The *JoinCounter* interface is shown in Figure 25.

3.8 DIESEL Entity Management Interfaces

Simulations regularly need to group together similar entities for better management and to direct them towards specific processes within the simulation. Another common requirement of simulations is the notion of requesting and acquiring entities (resources) from a holding pool for a particular task and releasing them back to the pool after the task has been completed. When a request is made for a certain number of entities, the request should be immediately satisfied if the entities are available. If not, then the request should be queued and satisfied on first-come-first-served basis. If a new request should arrive when there are pending requests, then the new request should be queued. This should occur even if the number of entities requested by the second request may be less than the first waiting request and there may be enough entities to serve the second request but not the first request. This ensures that a request for a large number of entities cannot be blocked by numerous requests for fewer entities [46].

DIESEL provides the following interfaces for management of entities.

- ***Set:*** A *Set* is a container capable of maintaining a collection of arbitrary entities. The *Set* interface is shown in Figure 26.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

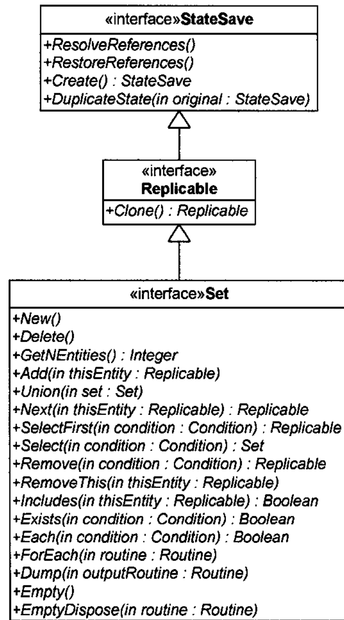


Figure 26. *Set* Interface

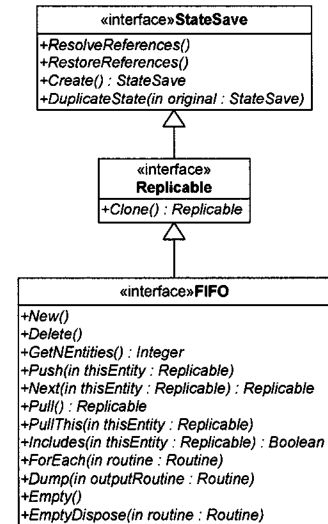


Figure 27. *FIFO* Interface

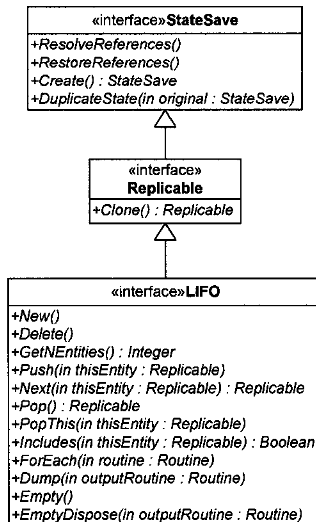


Figure 28. *LIFO* Interface

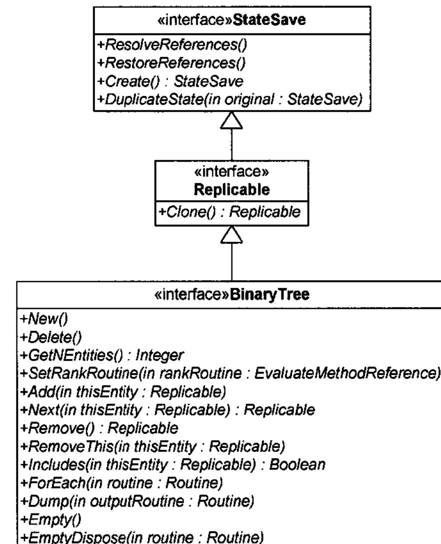


Figure 29. *BinaryTree* Interface

- ***FIFO:*** A *FIFO* maintains a collection of arbitrary entities in a first-in-first-out manner. The *FIFO* interface is shown in Figure 27.
- ***LIFO:*** A *LIFO* maintains a collection of arbitrary entities in a last-in-first-out

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

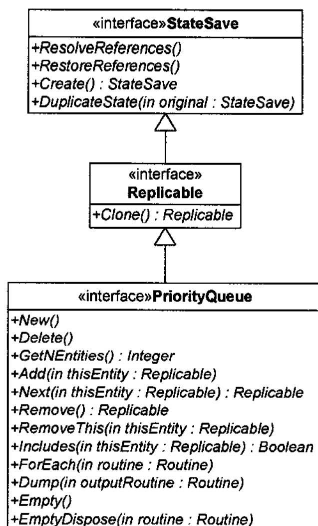


Figure 30. *PriorityQueue* Interface

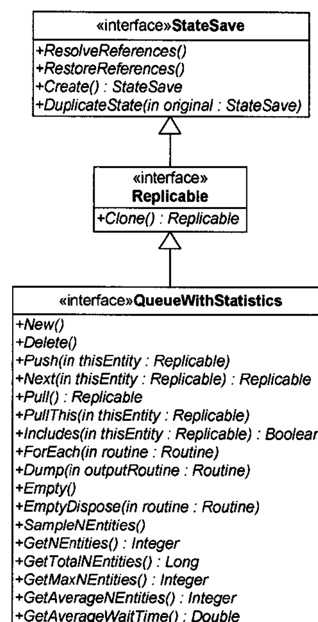


Figure 31. *QueueWithStatistics* Interface

manner (stack). The *LIFO* interface is shown in Figure 28.

- ***BinaryTree:*** A *BinaryTree* maintains a ranked collection of arbitrary entities. The *BinaryTree* interface is shown in Figure 29.
- ***PriorityQueue:*** A *PriorityQueue* maintains a collection of arbitrary entities based on a given priority. The *PriorityQueue* interface is shown in Figure 30.
- ***QueueWithStatistics:*** A *QueueWithStatistics* maintains a collection of arbitrary entities in a first-in-first-out manner, and tracks and reports various statistics associated with itself. The *QueueWithStatistics* interface is shown in Figure 31.
- ***EntityCounter:*** An *EntityCounter* holds a count of available entities, but does not hold actual entities. The availability of resources is indicated by a count within the *EntityCounter*. It satisfies requests on a first-come-first-served basis. The *EntityCounter* interface is shown in Figure 32.
- ***EntityPool:*** An *EntityPool* behaves identically to an *EntityCounter* except that it

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

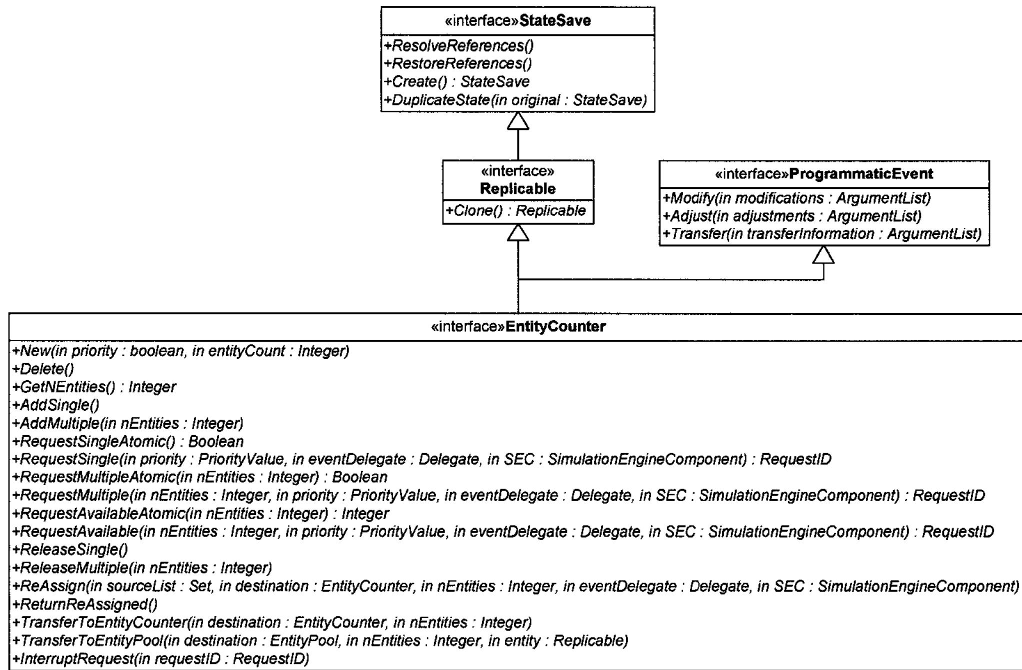


Figure 32. *EntityCounter* Interface

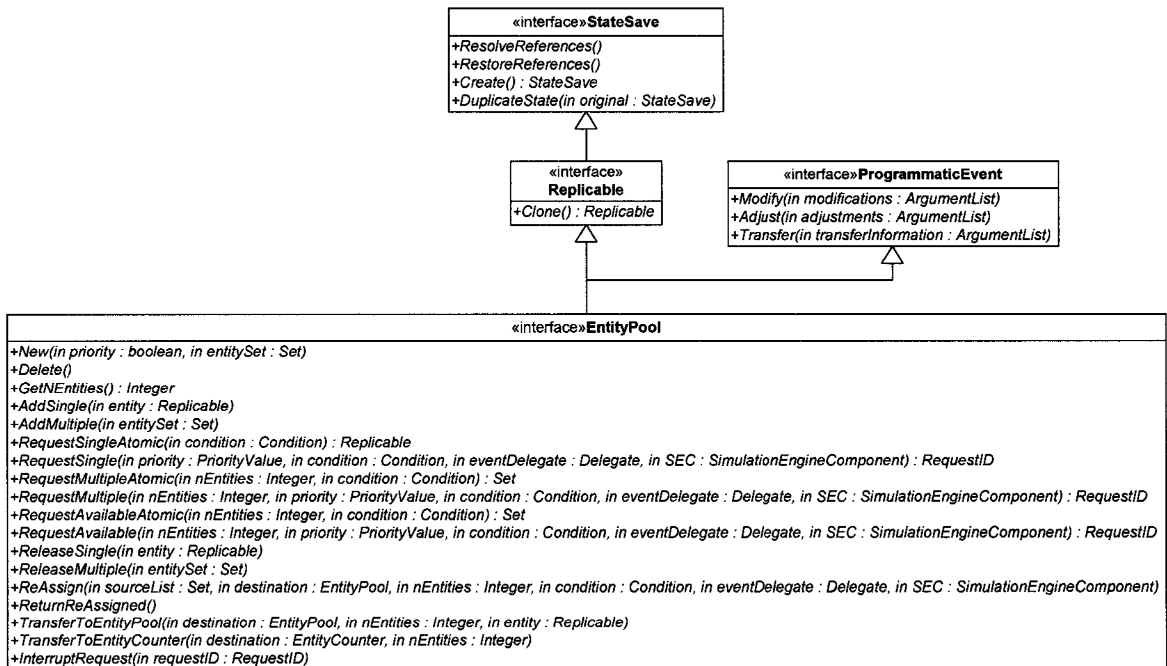


Figure 33. *EntityPool* Interface

holds a collection of actual available entities as shown in Figure 33.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

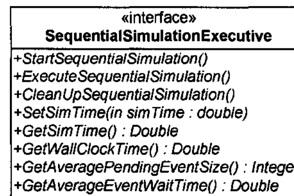


Figure 34. *SequentialSimulationExecutive* Interface

3.9 DIESEL Sequential Simulation Executive

The DIESEL simulation executive provides various methods for an application to initialize a simulation and to control and track its progress. The *SequentialSimulationExecutive* interface is shown in Figure 34.

An application can be built using sequential DIESEL by first initializing the DIESEL simulation executive using the "StartSequentialSimulation" method. This method initializes a single global *SimulationCluster* for the simulation. It also initializes the state support infrastructure in case a state save operation (terminating or non-terminating) needs to be performed during simulation execution. The next step is to create *SimObjects* and their associated *SimulationEngineComponents* and to schedule events to be executed on the *SimObjects* on their respective *SimulationEngineComponents*. The simulation can then be executed using the "ExecuteSequentialSimulation" method. The "CleanUpSequentialSimulation" method cleans up all DIESEL structures after executing a sequential simulation. This basic procedure to build an application using DIESEL is shown in Figure 35. The "SetSimTime" and "GetSimTime" methods provide access to the current simulation time, while the "GetWallClockTime" method returns the current wallclock time within a sequential simulation.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

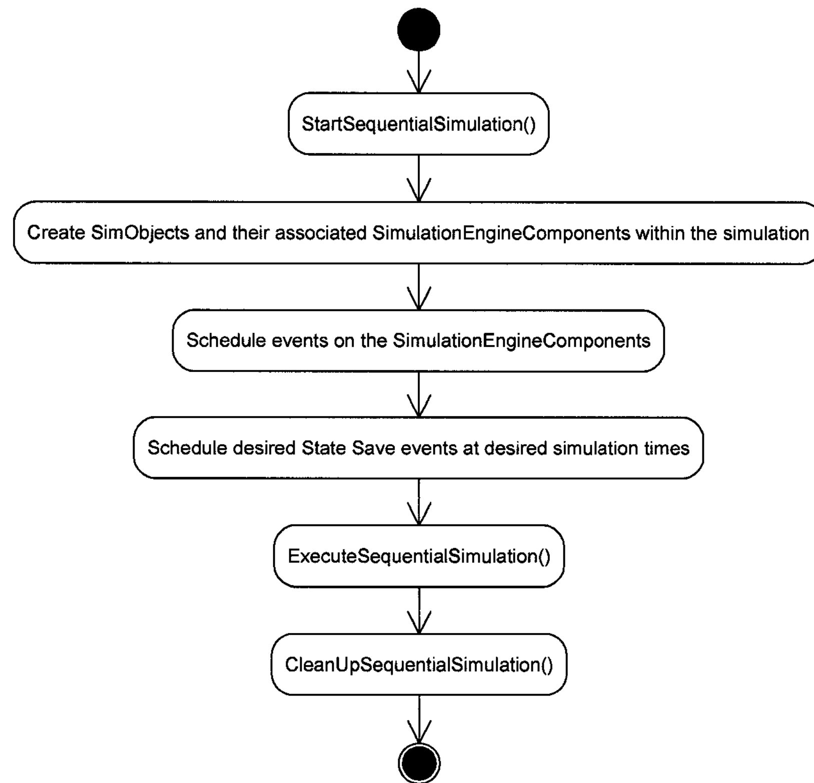


Figure 35. Basic Sequential Simulation

3.10 Implementation Classes for the DIESEL Engine Interfaces

This section provides the class definitions for implementing the core functionality for the DIESEL simulation executive. A basic simulation executive can be built by implementing the class definitions in this section.

3.10.1 *SimulationCluster* Class

The *SimulationCluster* class shown in Figure 36, implements most of the functions required of the simulation executive described in Section 3.9. A single global *SimulationCluster* object is automatically initialized when the "StartSequentialSimulation" method within the simulation executive interface is invoked.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

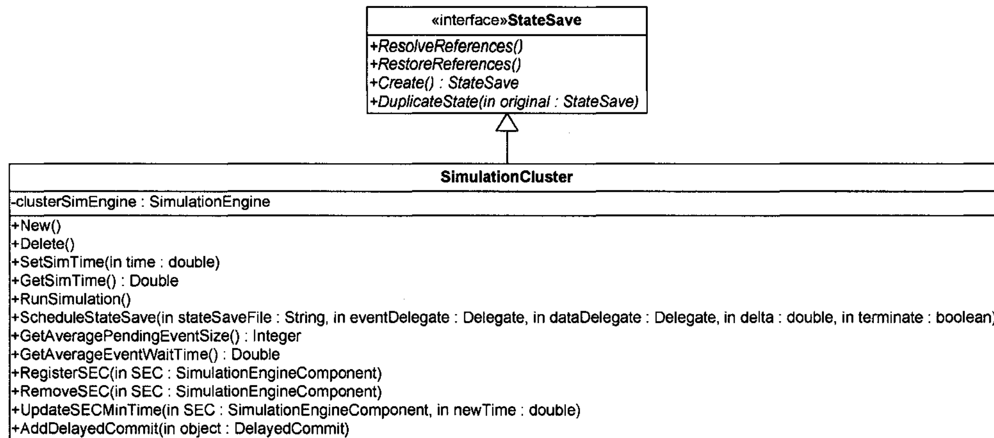


Figure 36. *SimulationCluster* Class

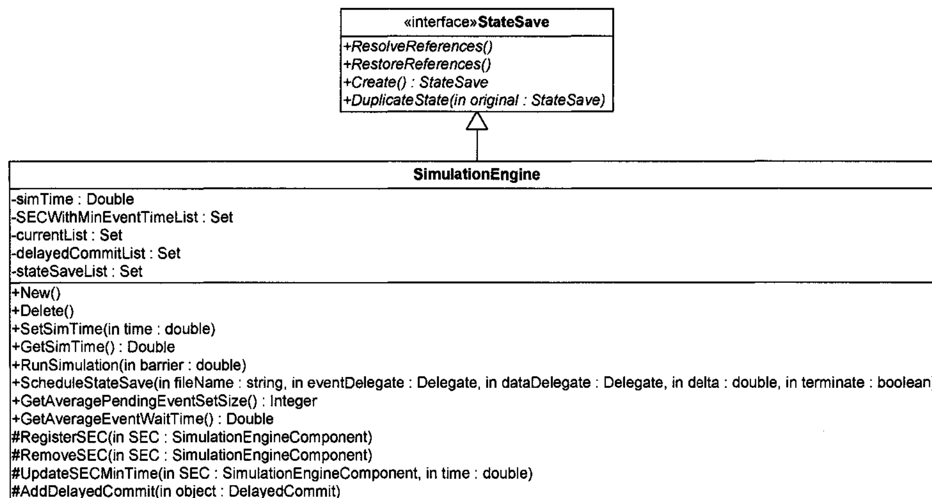


Figure 37. *SimulationEngine* Class

3.10.2 *SimulationEngine* Class

The *SimulationEngine* class shown in Figure 37 is used to implement each individual *SimulationEngine* within a *SimulationCluster*. It holds information about the state of the simulation as well as all the *SimObjects* within the simulation. It maintains a record of all *SimulationEngineComponents* in the simulation at the current simulation time within "SECWithMinEventTimeList". Each entry in the list has a reference to the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

SimulationEngineComponent as well as the minimum execution time of all events for the *SimulationEngineComponent*. The "SECWithMinEventTimeList" list should ideally be sorted based on the minimum event execution time of the events within the *SimulationEngineComponents* so that the next *SimulationEngineComponent* to be allowed to execute events can be determined easily. Other techniques can be also be implemented to identify the *SimulationEngineComponent* with the minimum event execution time.

The *SimulationEngine* uses the "RunSimulation" method to execute the simulation. The method first calculates the current simulation time, which is the smallest execution time of all the events scheduled on all *SimulationEngineComponents*. If multiple events are found with the same least scheduled execution time, then the *SimulationEngineComponents* on which they are scheduled are added to the "currentList". If a state save event exists with a scheduled time less than the calculated current simulation time, then the state save event is executed and the state of the simulation is saved. If not, the *SimulationEngine* proceeds to inform each *SimulationEngineComponent* in turn to execute these events. While each *SimulationEngineComponent* is executing events at the current simulation time, the *SimulationEngine* populates and maintains the "delayedCommitList" that holds all *DelayedCommits* whose values have to be updated after executing all events scheduled to be executed at the current simulation time. After all *SimulationEngineComponents* have executed their events at the current simulation time, the *SimulationEngine* informs all *DelayedCommits* within its "delayedCommitList" to commit their values. This process is continued until no more events exist within the simulation.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

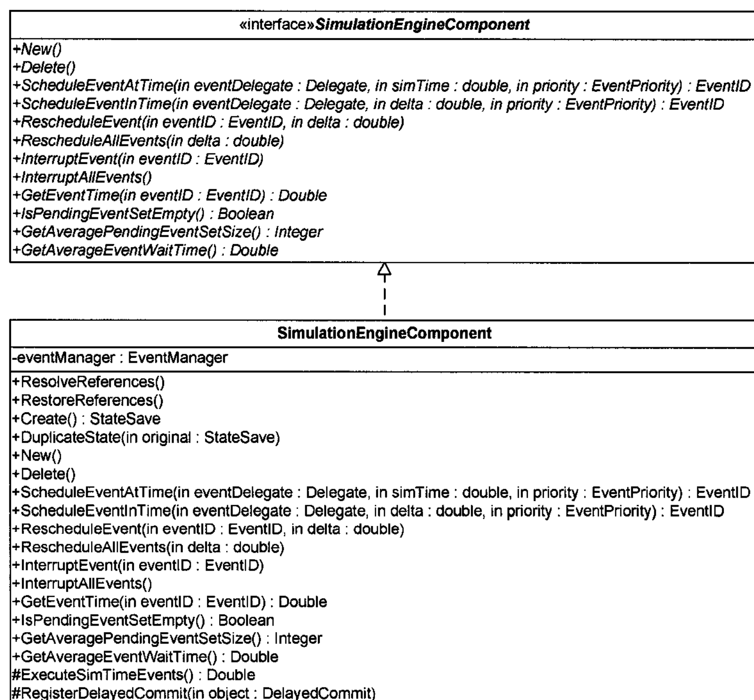


Figure 38. *SimulationEngineComponent* Class

3.10.3 *SimulationEngineComponent* Class

The *SimulationEngineComponent* class shown in Figure 38 implements the *SimulationEngineComponent* interface. It maintains its own list of events to be executed on SimObjects associated with it within "eventManager". The "ExecuteSimTimeEvents" method is used to execute event(s) in "eventManager" that have an execution time equal to the current simulation time.

The *SimulationEngine* class interacts with the *SimulationEngineComponent* class using the "ExecuteSimTimeEvents" method. The *DelayedCommit* class interacts with the *SimulationEngineComponent* class using the "RegisterDelayedCommit" method. These methods are protected and are not accessible to the application layer.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.10.4 *EventManager* Class

An important component of any simulation executive is the management of events scheduled to be executed on various objects within the simulation. Various issues need to be addressed to make event management as fast and efficient as possible so that the simulation runs more efficiently.

Event management within DIESEL is the shared responsibility of all *SimulationEngineComponents* within the simulation. The event management structure is distributed, i.e., each *SimulationEngineComponent* manages the collection of events to be executed by it for the duration of the simulation itself. This structure avoids a central event management mechanism within the *SimulationEngine*. There are advantages and disadvantages to avoiding a central structure; every *SimulationEngineComponent* does not need to communicate with the *SimulationEngine*, while scheduling an event within the simulation, and the size of a single list of events may not become too large. At a particular simulation time, the *SimulationEngine* pings each *SimulationEngineComponent* to execute all its events scheduled at that simulation time. However, an application developer, if he so chooses, can still have a single event list within the simulation, by initializing a single global *SimulationEngineComponent* for all *SimObjects* to schedule events on.

An *EventManager* within a *SimulationEngineComponent* encapsulates all the functionality to handle events to be executed by each *SimObject*. The *EventManager* stores all the events to be executed by the *SimObject* within "pendingEventSet". The "pendingEventSet" can be implemented in any manner from a simple linked list to a more complex structure, as long as it can interact with the *EventManager* and support all

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

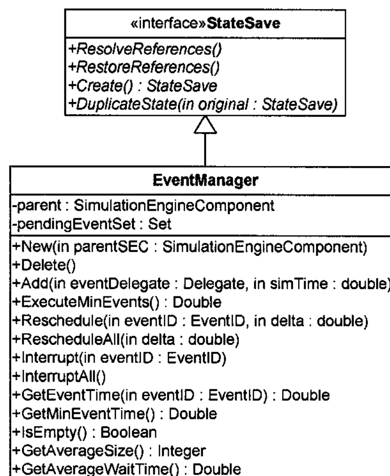


Figure 39. *EventManager* Class

functions to be performed by the *EventManager*.

It is a DIESEL design decision to keep event handling completely isolated within a *SimulationEngineComponent*. The *EventManager* has a reference to the *SimulationEngineComponent* that is associated with its parent *SimObject(s)* (Multiple *SimObjects* might be associated with the same *SimulationEngineComponent*). The "ExecuteMinEvents" method is used by the *SimulationEngineComponent* to ask the *EventManager* to execute all its events at the current simulation time. After it is done executing all events at the current simulation time, it returns the next lowest simulation time that it has events to be executed. This aids the *SimulationEngine* to know when to call the *SimulationEngineComponent* again to execute events. The *EventManager* class is shown in Figure 39.

Communication is allowed among various *SimObjects* in the simulation, as part of the standard interaction between *SimObjects*. Communication is allowed between a *SimObject* and its associated *SimulationEngineComponent* allowing a *SimObject* to schedule events on itself. Communication is also allowed between a *SimObject* and the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

SimulationEngineComponents of other *SimObjects*, allowing a *SimObject* to schedule events to be executed on other *SimObjects*.

Different *SimulationEngineComponents* can communicate with each other directly or through the *SimulationEngine* both for event scheduling and time management. A *SimulationEngineComponent* can communicate with *SimObjects* associated with it for execution of events scheduled on the *SimObjects*. There can be no direct communication between a *SimObject* and the *SimulationEngine*. Communication between a *SimObject* and the *SimulationEngine* is only possible through the *SimulationEngineComponent* the *SimObject* it is associated with.

3.11 Examples

This section describes two examples to demonstrate the operation of the DIESEL behavioral model. The first example shows a simple timing diagram for event execution while the Dining Philosophers' problem is implemented in the second example.

3.11.1 Timing diagram for execution of events

This example demonstrates the interaction between the *SimulationEngine* and *SimulationEngineComponent* classes and various *SimObjects* with an example scenario. The scenario shows the timing characteristics of the interaction between these classes. The exact order of execution of events scheduled at a particular simulation time depends on the strategy used to maintain the list of objects within the simulation, i.e. the strategy used to implement "SECWithMinEventTimeList" and "currentList" within the *SimulationEngine* class.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The following representation is used in the example scenario:

- Each *SimObject* ($SO_1 - SO_3$) is associated with a single *SimulationEngineComponent* ($SEC_1 - SEC_3$).
- Each entry in the table represents an event with its scheduled execution time as the first integer.
- [ ] represents the set of events created when the current event is executed.
- (a, b) provides details about the created event. "a" is the particular *SimObject* on which the event is to be scheduled, and "b" is the execution time of the event.

At SimTime = 0, the list of events to be executed by each *SimulationEngineComponent* is as shown in Figure 40(a).

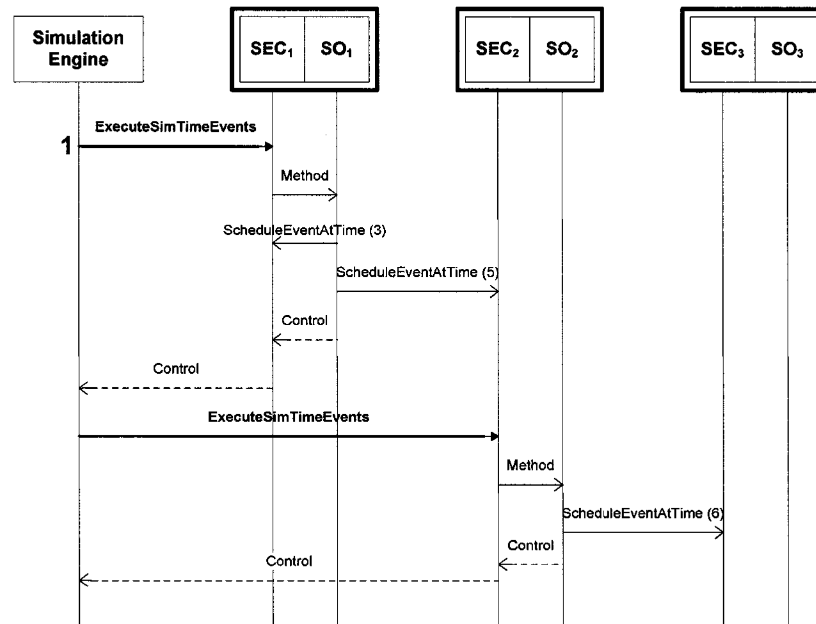
At SimTime = 1, the *SimulationEngine* goes through the list of *SimulationEngineComponents* ($SEC_1 - SEC_3$) to find the minimum scheduled execution time of all events. *SimulationEngineComponents* are added to *currentList* within the *SimulationEngine*, if they have events scheduled at the minimum execution time. In this case, SEC_1 and SEC_2 are added to the *CurrentList*. The *SimulationEngine* then passes control to the first *SimulationEngineComponent* in the list, i.e., SEC_1 using the *ExecuteSimTimeEvents* method. SEC_1 executes its event and schedules two events, an event at SimTime = 3 on itself and an event at SimTime = 5 on SEC_2. SEC_1 then returns control back to the *SimulationEngine*, which in turn passes control to SEC_2 again, using the *ExecuteSimTimeEvents* method. SEC_2 executes its event and schedules a single event at SimTime = 6 on SO_3. SEC_2 then returns control back to the *SimulationEngine*. This process is shown in Figure 40(b).

At SimTime = 2, SEC_3 is added to the *currentList*. The *SimulationEngine* passes

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

SO1	SO2	SO3
1 [(1,3), (2,5)]	1 [(3,6)]	2 [(2,5), (3,5)]
4 [(1,6), (2,4)]	5 [(1,5)]	4 [(1,6), (2,7)]
5 [(1,5)]		
5 [--]		

Figure 40a. Timing Diagram: Events at SimTime = 0

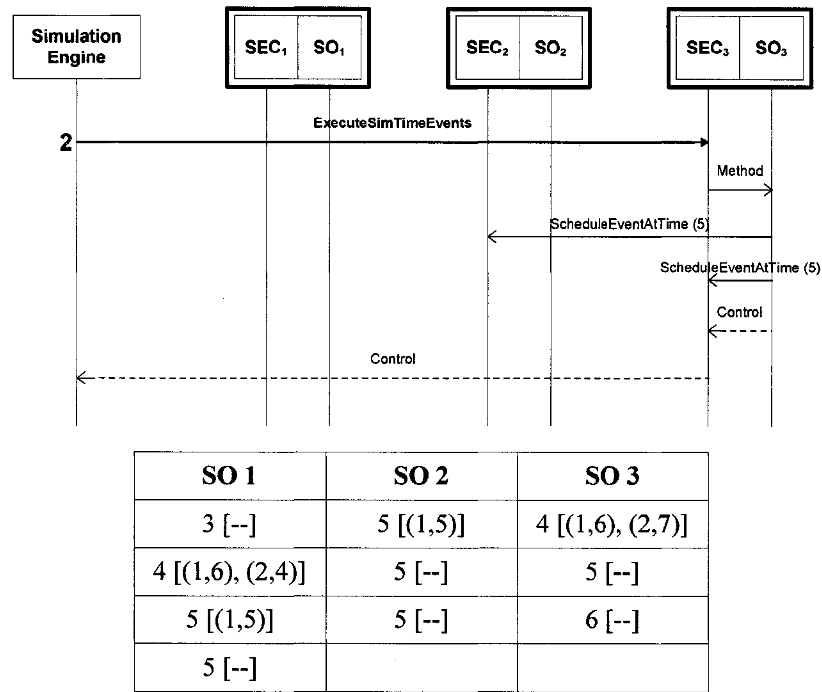


SO1	SO2	SO3
3 [--]	5 [(1,5)]	2 [(2,5), (3,5)]
4 [(1,6), (2,4)]	5 [--]	4 [(1,6), (2,7)]
5 [(1,5)]		6 [--]
5 [--]		

Figure 40b. Timing Diagram: Event execution and events at SimTime = 1

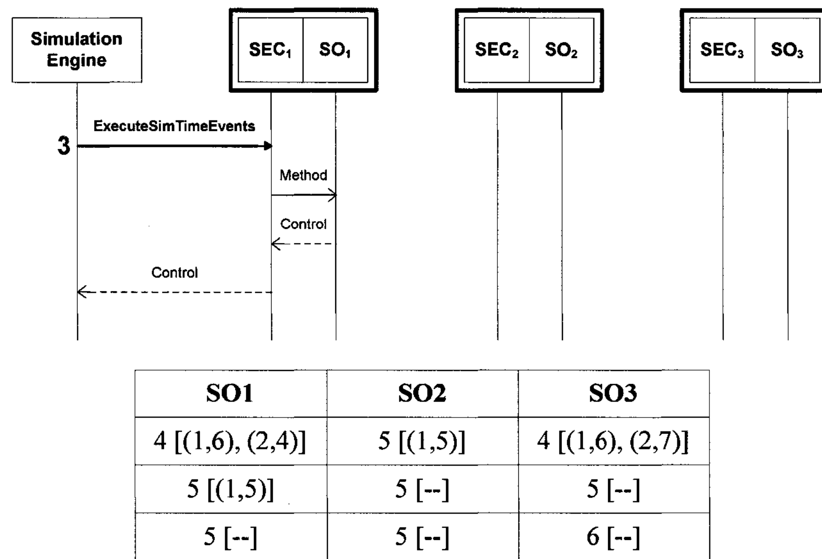
control to SEC_3 again using the *ExecuteSimTimeEvents* method. SEC_3 executes its event and schedules two events, an event at SimTime = 5 on itself and an event at SimTime = 5 on SEC_2. SEC_3 then returns control back to the *SimulationEngine*. This process is shown

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



SO 1	SO 2	SO 3
3 [--]	5 [(1,5)]	4 [(1,6), (2,7)]
4 [(1,6), (2,4)]	5 [--]	5 [--]
5 [(1,5)]	5 [--]	6 [--]
5 [--]		

Figure 40c. Timing Diagram: Event execution and events at SimTime = 2

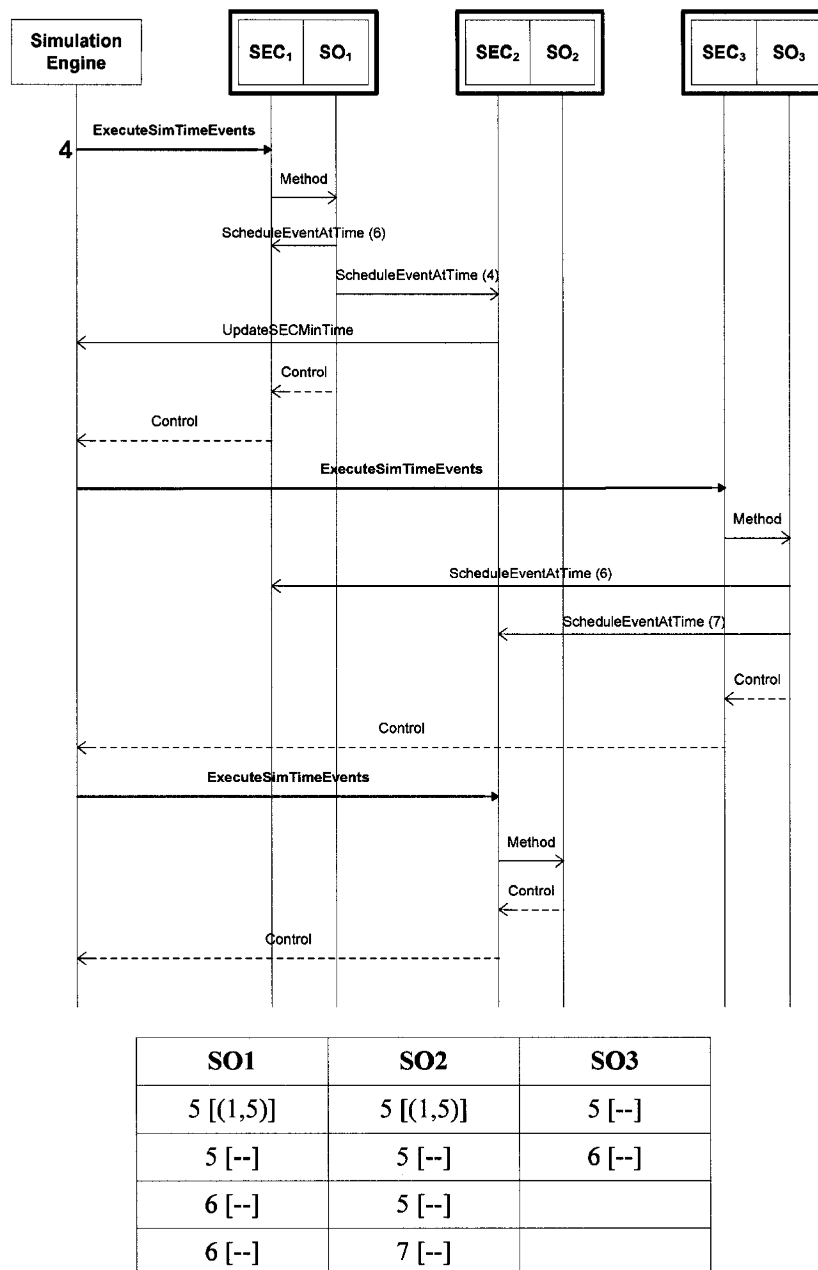


SO1	SO2	SO3
4 [(1,6), (2,4)]	5 [(1,5)]	4 [(1,6), (2,7)]
5 [(1,5)]	5 [--]	5 [--]
5 [--]	5 [--]	6 [--]

Figure 40d. Timing Diagram: Event execution and events at SimTime = 3

in Figure 40(c).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



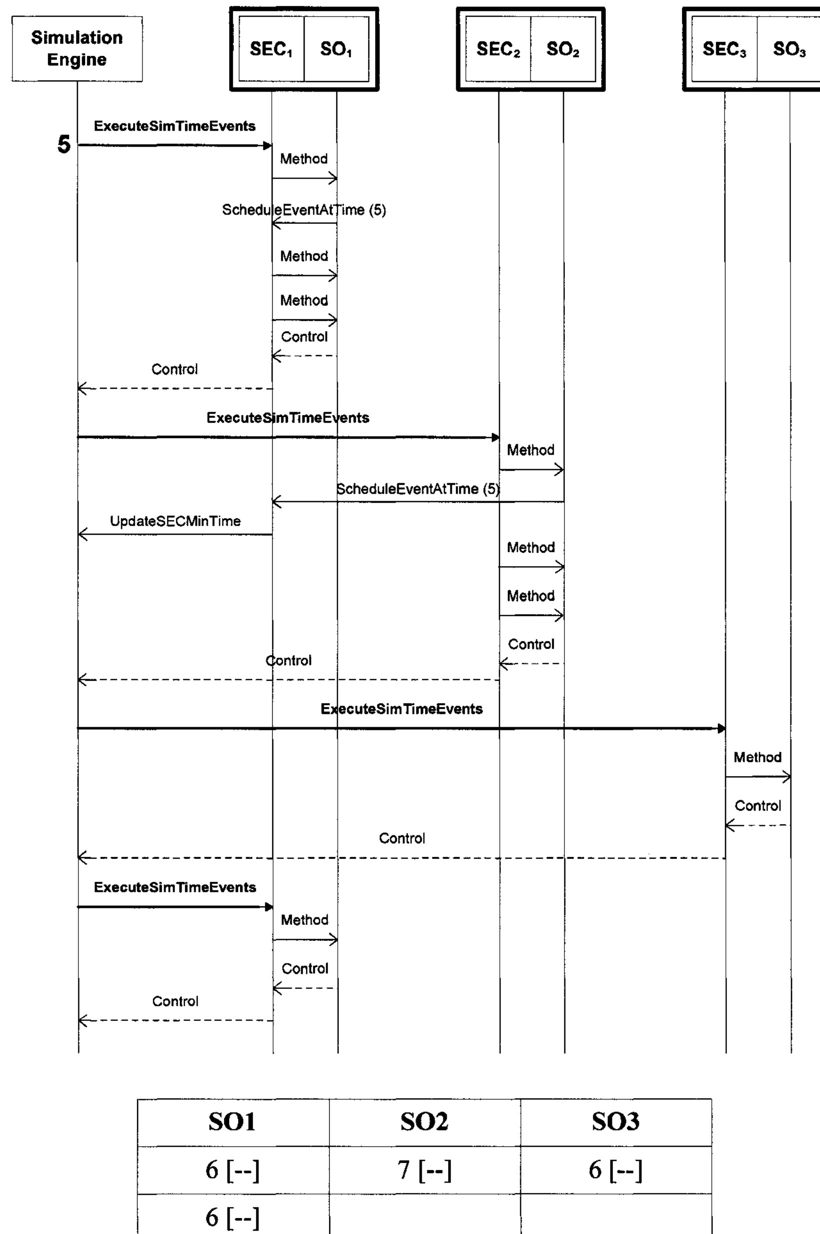
SO1	SO2	SO3
5 [(1,5)]	5 [(1,5)]	5 [--]
5 [--]	5 [--]	6 [--]
6 [--]	5 [--]	
6 [--]	7 [--]	

Figure 40e. Timing Diagram: Event execution and events at SimTime = 4

At SimTime = 3, SEC_1 is added to the *currentList*. SEC_1 executes its event and schedules no events as shown in Figure 40(d).

At SimTime = 4, SEC_1 and SEC_3 are added to the *currentList*. The *SimulationEngine* passes control to SEC_1 again using the *ExecuteSimTimeEvents* method.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

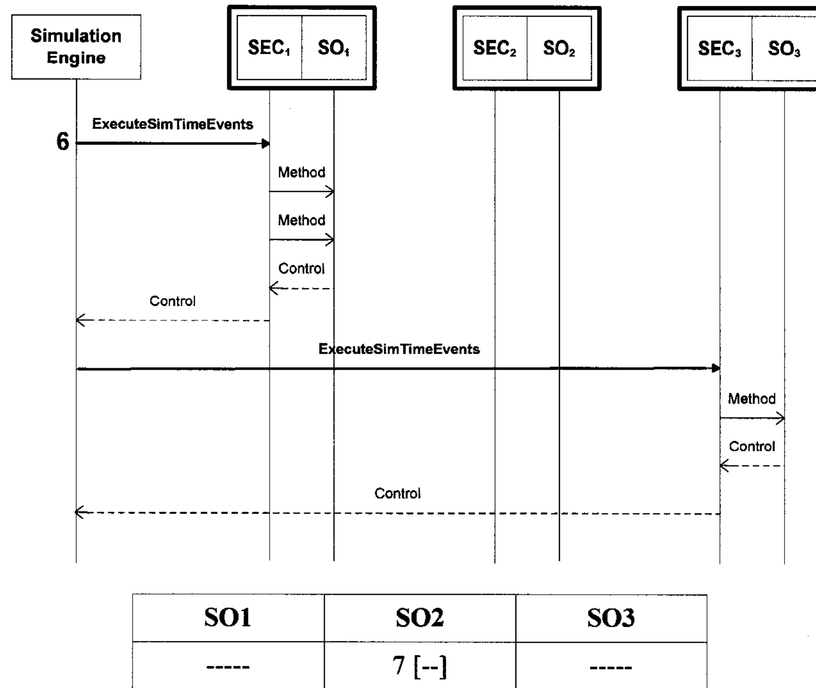


SO1	SO2	SO3
6 [--]	7 [--]	6 [--]
6 [--]		

Figure 40f. Timing Diagram: Event execution and events at SimTime = 5

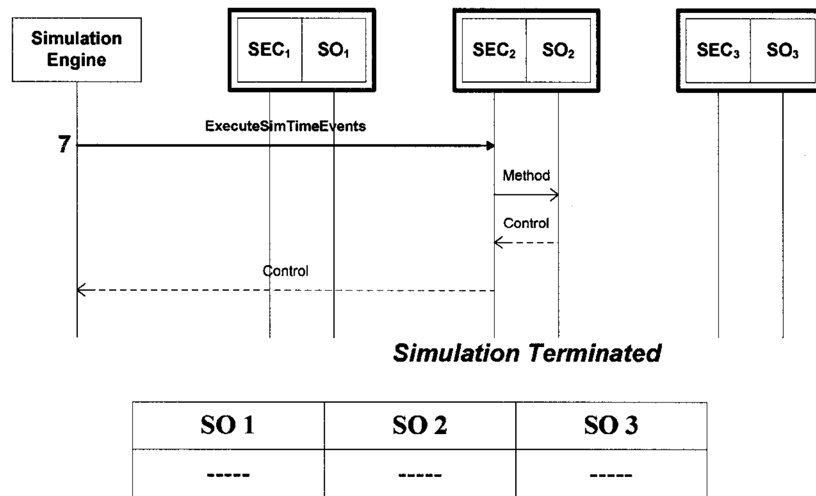
SEC_1 executes its event and schedules two events, an event at the current SimTime = 4 on SEC_2 and an event at SimTime = 6 on itself. Since an event has been scheduled at the current SimTime, SEC_2 calls the *SimulationEngine* to update its *currentList* using the *UpdateSECMinTime* method. SEC_2 has its minimum event execution time updated and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



SO1	SO2	SO3
-----	7 [--]	-----

Figure 40g. Timing Diagram: Event execution and events at SimTime = 6



SO 1	SO 2	SO 3
-----	-----	-----

Figure 40h. Timing Diagram: Event execution and events at SimTime = 7

has now been added to the *currentList*. SEC_2 returns control to SEC_1, which in turn returns control to the *SimulationEngine*. The *SimulationEngine* then passes control to the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

next *SimObject* in the list, SEC_3 which executes it event and schedules two events, an event at SimTime = 6 on SEC_1 and an event at SimTime = 7 on SEC_2. SEC_3 then returns control to the *SimulationEngine*. The *SimulationEngine* then passes control to the next *SimulationEngineComponent* in the list, i.e., SEC_2. SEC_2 now executes its event and then returns control back to the *SimulationEngine*. This process is shown in Figure 40(e).

At SimTime = 5, SEC_1, SEC_2 and SEC_3 are added to the *currentList*. The execution of events proceeds as shown in Figure 40(f).

At SimTime = 6, SEC_1 and SEC_3 are added to the *currentList*. The execution of events proceeds as shown in Figure 40(g).

At SimTime = 7, SEC_2 is added to the *currentList*. The execution of events proceeds as shown in Figure 40(h).

3.11.2 Dining Philosophers' Problem

The dining philosophers' problem is a common concurrency problem set forth by Edsger Dijkstra [47] and demonstrates the problems associated with multi-process synchronization. The dining philosophers' problem has five philosophers seated at a table. They spend their time alternating between thinking and eating a bowl of spaghetti at the center of the table. Each philosopher requires a pair of chopsticks to eat the spaghetti. However, there are only five chopsticks at the table, one between each pair of philosophers. This implies that a maximum of two philosophers can be eating at any point of time. After thinking for a certain time, each philosopher tries to eat, by picking up one chopstick at a time. A potential for deadlock exists with adjoining philosophers picking up the chopsticks on their same sides and then waiting for the other chopstick to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

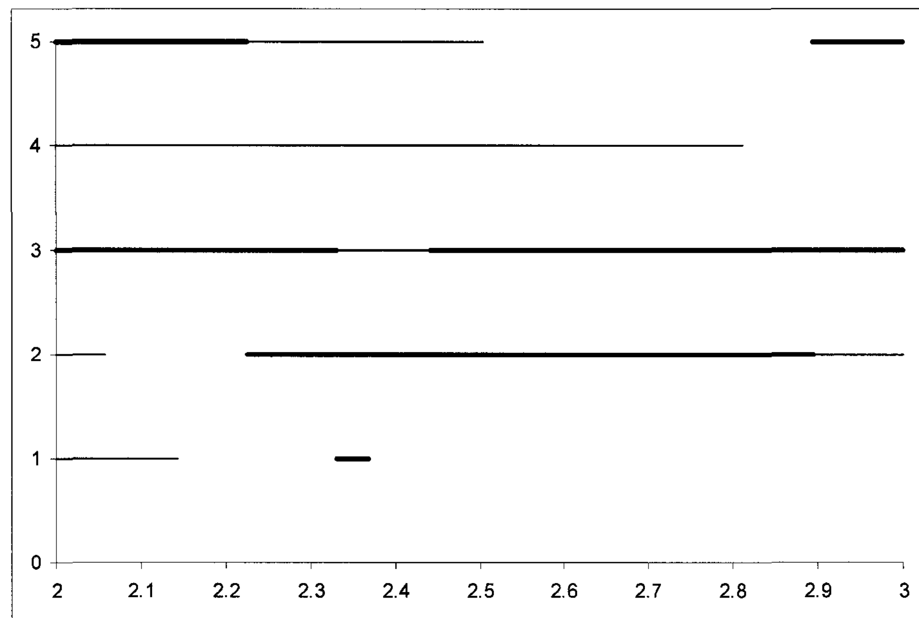


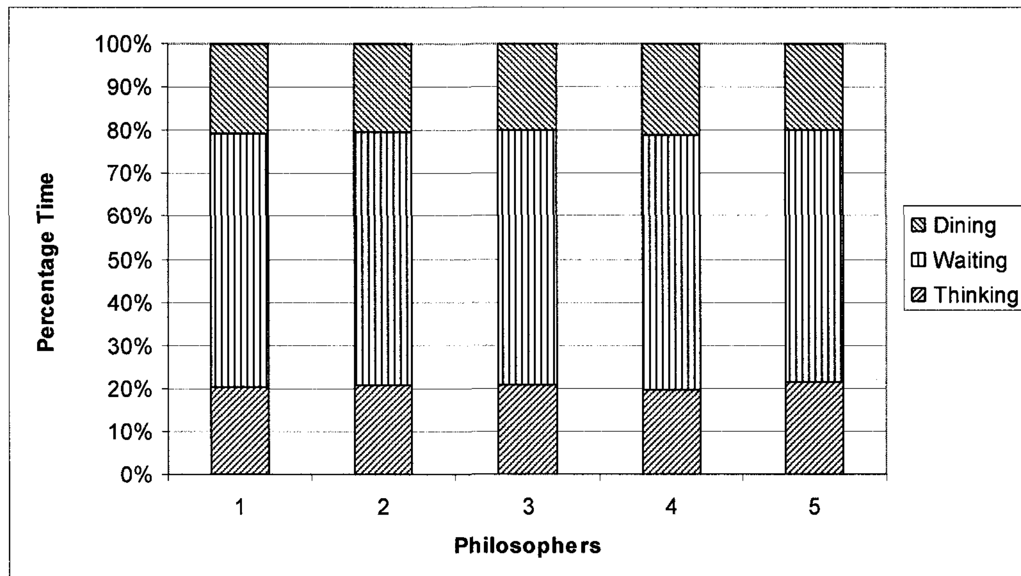
Figure 41. Snapshot of Dining Philosophers

become available, without releasing the chopstick that they have already acquired. A possible solution to avoid such deadlock is that if only one chopstick is available, then the philosopher puts down the chopstick, waits for a while and then attempts to eat again.

There are various modifications of this basic problem. In this example, the basic problem has been modified to include a centralized resource pool. The chopsticks are placed at the center of the table in an *EntityCounter* beside the spaghetti. Each philosopher starts by thinking for a random amount of time. Then, the philosopher requests the *EntityCounter* for 2 chopsticks. If the chopsticks are available, then each philosopher eats for a random amount of time and then returns the chopsticks to the *EntityCounter*. The requests, if not satisfied immediately, are queued in the order that they are received. The aim is to demonstrate that no philosopher is starved by waiting an inordinately long time for the required chopsticks to eat.

The times that a philosopher spends thinking and eating are both random numbers

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



Graph 1. Time spent by Philosophers eating, thinking and waiting

from Uniform (0, 1). Figure 41 shows a snapshot of the time interval between SimTime = 1 and SimTime = 3. The thick lines shows a philosopher eating, a thin line shows a philosopher thinking, while the gaps show the time that a philosopher is waiting for chopsticks to start eating. The figure shows that not more than two philosophers are eating at the same instant of time. Graph 1 shows the percentage of time that each philosopher spends thinking, eating, and waiting for chopsticks to eat. The graph shows that each philosopher spends approximately the same amount of time eating and thinking for the entire duration of the simulation.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter IV

INTERFACE SPECIFICATION FOR THE DIESEL DISTRIBUTED MODEL

This chapter provides a complete description of the distributed component of the DIESEL behavioral model. The sequential model described in Chapter III describes the interfaces required to build a sequential application using a single *SimulationCluster* and details the communication between components within a *SimulationCluster*. The distributed model describes the initialization of multiple *SimulationClusters* and the required communication between different *SimulationClusters* to manage and execute a distributed simulation. Most of the interfaces described in Chapter III can be used with no changes within the distributed application. Some new interfaces are defined to support communication between *SimulationClusters*. An interface for a distributed simulation executive that creates *SimulationClusters*, registers *SimulationClusters* and application objects, and executes the distributed simulation is specified. This interface can be extended to support different synchronization algorithms and communication mechanisms.

4.1 Interaction between components within DIESEL

A distributed simulation involves *SimObjects* residing on different *SimulationClusters* interacting with each other. This collection of *SimulationClusters* is known as a *SimulationForest* and is shown in Figure 42. These *SimulationClusters* can exist on the same processor or different processors depending on the number of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

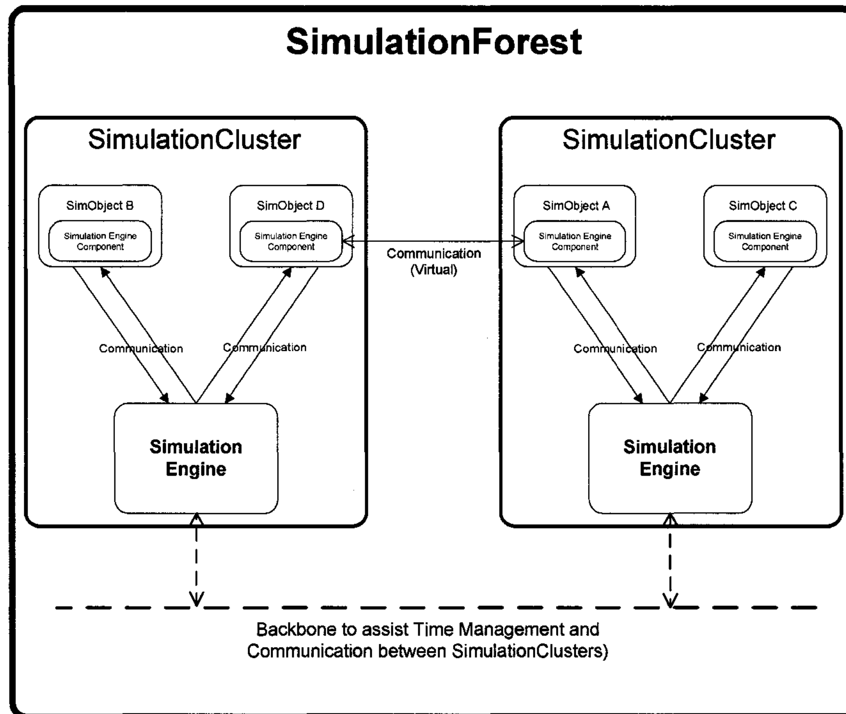


Figure 42. *SimulationForest*

processors used for the simulation.

The first step in building a distributed application is to create the required number of *SimulationClusters* on the available processors and make the simulation executive aware of each of them. An application can also be built using a single *SimulationCluster* with the extra overhead associated with *SimulationCluster* interconnection and management. A simulation executive should exist on each processor in the simulation and hold the location of all *SimulationClusters* on all processors in the simulation. A simulation executive receives messages from another processor and relays it to the appropriate *SimulationCluster* existing on that processor. This registry of processors within every simulation executive is shown in Figure 43.

A distributed simulation differs from a sequential simulation in the manner a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

SimulationCluster ID	Processor Name
0	MATRIX-1
1	MATRIX-2
2	MATRIX-2
3	MATRIX-1

Figure 43. Registry of processors within simulation executive

SimObject accesses other *SimObjects*. *SimObjects* within a sequential simulation access other *SimObjects* directly, since they exist on a single processor for the entire simulation. *SimObjects* within a distributed simulation, on the other hand, can exist within different *SimulationClusters* and on different processors. A *SimObject* has no knowledge of the location of another *SimObject* and, therefore, cannot access it directly.

A distributed application implemented with DIESEL is required to generate unique references for all *SimObjects* and other objects that are created by the application and interact with each other. The unique references can be generated by DIESEL or by the application. This unique reference is required to be public knowledge across the simulation for an object within one *SimulationCluster* to refer to an object from another *SimulationCluster*. Publishing of this reference is currently left to the application. As an alternative, HLA [36, 37, 38] manages its objects by publishing shareable federation objects in the Federation Object Model (FOM) and Simulation Object Model (SOM). An object that is completely private to another object and interacts with no other object does not need a unique reference, since no other application object will ever attempt to access it.

Each unique reference that is generated for an application object must be registered with the simulation executive by the application with the following

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Unique Reference	SimulationCluster ID	Object	SimulationEngine Component
1	0	NULL	NULL
22	1	Obj1	SEC1
9	1	Obj2	SEC2
13	0	NULL	NULL

Figure 44. Registry of objects within a *SimulationCluster*

information:

- the unique reference
- the application object
- the *SimulationCluster* on which the application object has been created
- the associated *SimulationEngineComponent* if the object is a *SimObject*

The simulation executive informs each *SimulationCluster* of the new object that has been registered. Each *SimulationCluster* has a registry of all registered objects along with their unique references as shown in Figure 44. The object registry is shared among all *SimulationClusters* and is synchronized every time a reference is created or removed from an object registry within any *SimulationCluster*. The registry has a valid entry for each object that has been created on that *SimulationCluster* and a null value in the object and *SimulationEngineComponent* fields for objects on other *SimulationClusters*.

Each application object when requiring access to another object, asks its parent *SimulationCluster* to resolve the reference of the object and pass the required data to that object as shown in Figure 45. The *SimulationCluster* resolves all references by accessing its object registry. If the object exists within the *SimulationCluster*, it passes the message to the object directly. If the object does not exist within the *SimulationCluster*, it asks the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

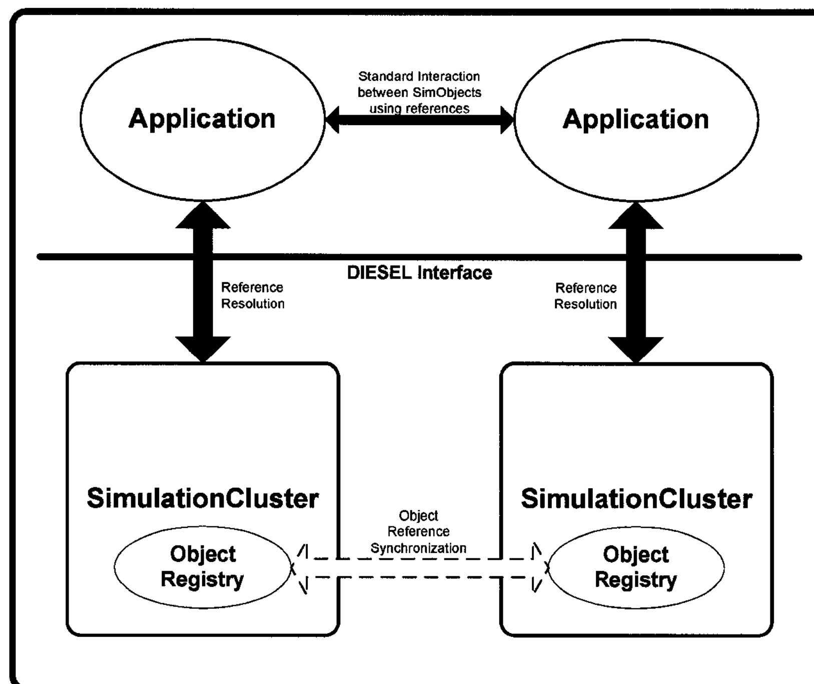


Figure 45. Reference resolution between *SimulationClusters*

simulation executive resident on the processor it exists on to relay the message to the appropriate *SimulationCluster*, which holds the object that is being accessed.

There is an overhead attached to the access between application objects within the distributed model. Every application object needs to use its parent *SimulationCluster* as an intermediary to access another object, irrespective of the location of both objects. This is different from the direct access method within the sequential model where objects can access each other directly. However, this type of access does not require the application to know what type of access it is requesting, local or remote, thereby making it completely transparent to the application.

The simulation executive, when asked to relay a message, determines which *SimulationCluster* holds the object that is being asked for. If the *SimulationCluster* exists on the current processor, it relays the message directly to the *SimulationCluster*. If the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

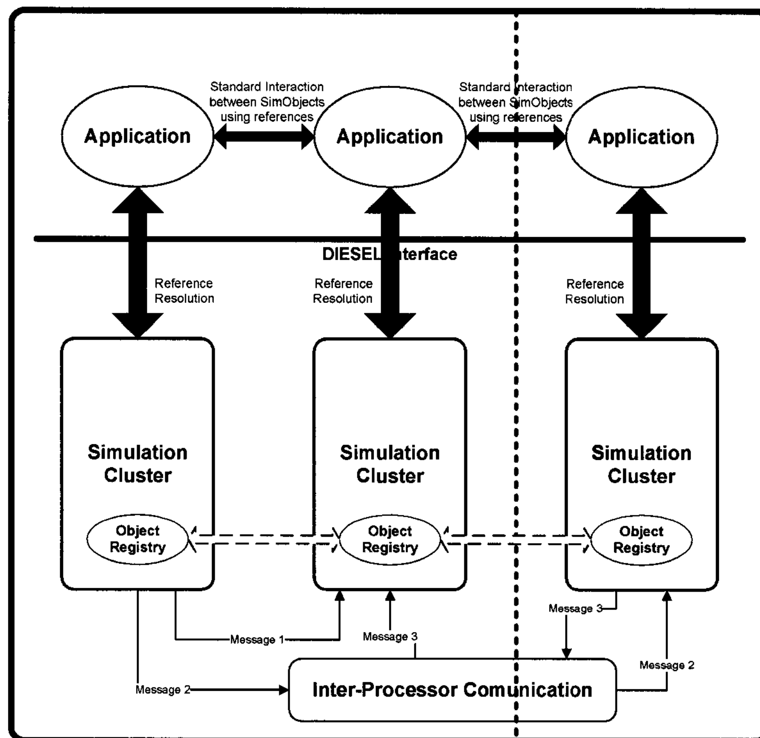


Figure 46. Communication between *SimulationClusters*

SimulationCluster exists on another processor, then the simulation executive asks the communication mechanism to transmit the message to the appropriate processor. The simulation executive on the destination processor then relays the message to the appropriate *SimulationCluster*. This process is shown in Figure 46.

A reliable communication mechanism is assumed between processors such that no messages passed between processors are lost. However, no assumption is made in the order of delivery of the messages, i.e. that the messages are received in the order that they were sent. Messages can also be bunched together depending on the communication protocol. Different communication mechanisms exist in literature from as simple as socket communication [48] to more advanced protocols such as Message Passing Interface (MPI) [49], Distributed Component Object Model (DCOM) from Microsoft©

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Corporation [50, 52], Common Object Request Broker Architecture (CORBA) [51, 52] from Object Management Group (OMG), and Java/Remote Method Invocation (Java/RMI) from JavaSoft [52].

4.2 Extensions to the DIESEL Interface

All interfaces defined in Chapter III can be used with no modifications unless a modified interface has been defined in this section. Most interfaces require minimal or no changes to be used within the distributed model. The distributed model extends rather than make radical changes to the sequential interface. The *SimulationEngineComponent* interface described in Section 3.2.3 requires a new parameter, specifying the *SimulationCluster* it belongs to while being initialized. This is shown in Figure 47.

A new interface needs to be defined to support the distributed model. The *SimulationCluster* interface, shown in Figure 48, initializes a new *SimulationCluster* and manages all objects that are created within the *SimulationCluster*. An application uses the "ScheduleEvent" method to schedule an event to be executed on another *SimObject*.

4.3 DIESEL Distributed Simulation Executive

The *DistributedSimulationExecutive* interface shown in Figure 49 has various methods to initialize a distributed simulation and to create and register *SimulationClusters*. The interface uses the "EventsExist" and "ExecuteDistributedSimulation" to execute the simulation. The "EventsExist" method determines if any *SimulationCluster* has at least one event to be executed to continue simulation execution, while the "ExecuteDistributedSimulation" method executes the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

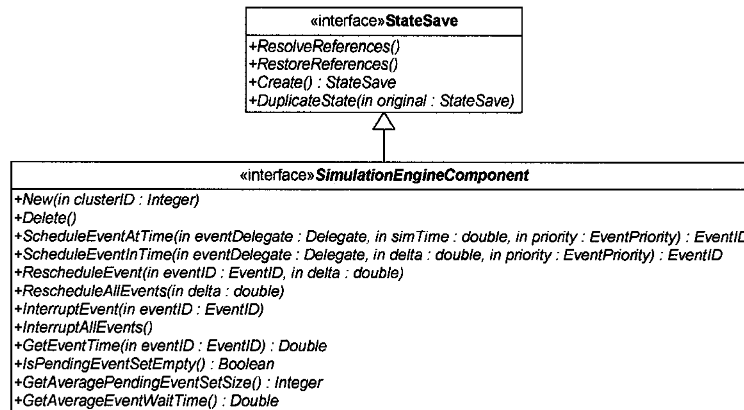


Figure 47. Modified *SimulationEngineComponent* Interface

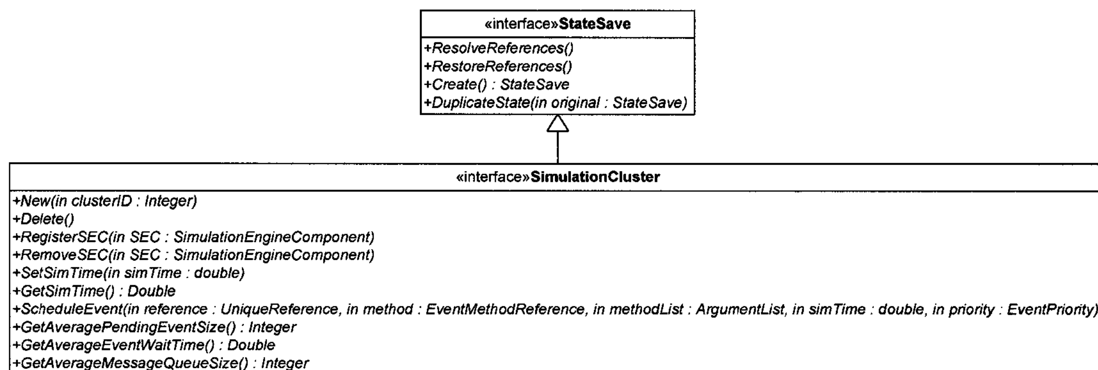


Figure 48. *SimulationCluster* Interface

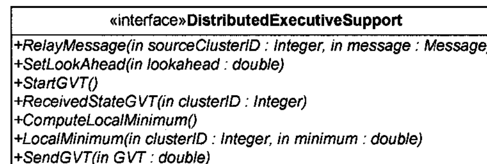


Figure 49. *DistributedSimulationExecutive* Interface

simulation with the specified synchronization algorithm. The application can use the "GetUniqueReference" method to specify a unique reference for an object or *SimObject*. The application uses the "RegisterObject" method to register a new object, its unique

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

reference, and its location with the simulation executive. The "RemoveObject" method is used to migrate the object to a new *SimulationCluster*. The interface can be extended with new methods to support different synchronization algorithms for a distributed simulation. Two synchronization algorithms are included within the Distributed DIESEL interface to manage time within the distributed model. The different methods to manage the execution of the simulation using these algorithms are discussed in the following sub-sections.

4.3.1 Barrier Synchronization Algorithm

The *barrier synchronization* algorithm described in Chapter I is included to demonstrate that the interface can support conservative algorithms within PDES. The first step is to calculate the *lookahead* within the application, which is different for each application [1]. Then the "ExecuteDistributedSimulation" method calculates the next set of safe events to process and executes the simulation as shown by the following algorithm:

```
Procedure
begin
      while (EventsExist()
           minTime = minimum event execution time of all
             SimulationClusters
           for all SimulationClusters
                execute events with timestamp <= (minTime +
                  lookahead)
end
```

4.3.2 *Time Warp* Paradigm

The *Time Warp* paradigm described in Chapter I is implemented to demonstrate how the interface supports optimistic algorithms within PDES. A Time Warp Logical Process (TWLP) can be mapped onto DIESEL in the following ways:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- Each *SimObject* can be treated as a TWLP. In this case, its associated *SimulationEngineComponent* would be part of the TWLP and could be part of multiple TWLPs. This would involve a *SimulationEngineComponent* being distributed across processors, if TWLPs are on different processors.
- Each *SimulationEngineComponent* can be treated as a TWLP. In this case, all *SimObjects* associated with it would be part of the same TWLP.
- A collection of *SimulationEngineComponents* can be treated as a TWLP. In this case, all *SimObjects* associated with them would be part of the same TWLP.
- Each *SimulationCluster* can be treated as a TWLP. In this case, all *SimulationEngineComponents* (and associated *SimObjects*) would be part of the same TWLP.
- A collection of *SimulationClusters* can be treated as a TWLP.

The selected mapping should have an input queue to receive messages from other TWLPs and an output queue to store anti-messages of messages sent to other TWLPs. It should also have the ability to save its state so that it can perform rollback in case a message arrives with a timestamp in its past.

Avril and Tropper [53] describe another technique in which TWLPs are grouped together into clusters (different from a *SimulationCluster*). The technique then identifies each TWLP by its name and the cluster it belongs to and assigns an input queue and output queue to each cluster instead of each TWLP. This grouping reduces some of the overhead required of each TWLP in *Time Warp* by associating the overhead of managing the input and output queues with the cluster rather than each individual TWLP. This technique can be mapped onto DIESEL by defining an interface for a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

SimulationClusterGroup (collection of related *SimulationClusters*) and then assigning the input and output queues to the *SimulationClusterGroup.*

4.4 Support Classes for the DIESEL Distributed Interface

This section provides definitions for support classes within the distributed DIESEL simulation executive to support a distributed simulation and the two synchronization algorithms described in the previous section. The application interfaces defined in Sections 4.2 and 4.3 cannot be modified. This section describes the internal support to the interface and has been defined so that the same structure can be used for different communication mechanisms. The classes defined here can be used within an implementation or new classes can be defined and implemented to support the interface.

4.4.1 *ObjectRegistry* Class

The *ObjectRegistry* class, shown in Figure 50 implements the registry of all application objects within a *SimulationCluster*. It stores information about all objects that have been registered with the simulation executive by the application. An *ObjectRegistry* exists within each *SimulationCluster* and holds unique references for all application objects and a reference to the object if it exists on the *SimulationCluster* that the *ObjectRegistry* is associated with. If the application object exists on another *SimulationCluster*, then the object field has a null value for that object within the current *ObjectRegistry*. An *ObjectRegistry* also holds a reference to the associated *SimulationEngineComponent* if the registered object is a *SimObject.*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

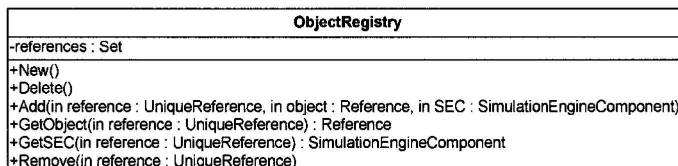


Figure 50. *ObjectRegistry* Class

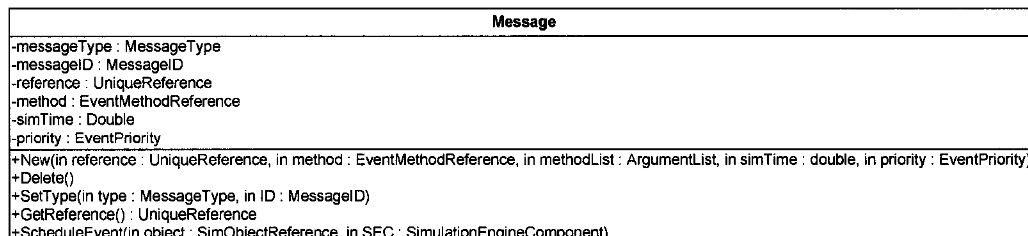


Figure 51. *Message* Class

4.4.2 *Message* Class

The *Message* class is used to communicate between *SimulationClusters* for both synchronization algorithms. The *Message* class shown in Figure 51 encapsulates all information required for objects to communicate with each other in a distributed simulation. It supports one object scheduling an event method to be executed on a *SimObject* residing on a different *SimulationCluster*. Once the message has been relayed to the appropriate *SimulationCluster*, the *SimulationCluster* uses the "GetReference" and "ScheduleEvent" methods to schedule the event for execution.

4.4.3 *MessageQueue* Class

The *MessageQueue* class shown in Figure 52 implements the input and output queues within a *SimulationCluster* for the *Time Warp* implementation. When a message or an anti-message is added to the *MessageQueue*, the *MessageQueue* first ascertains if

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

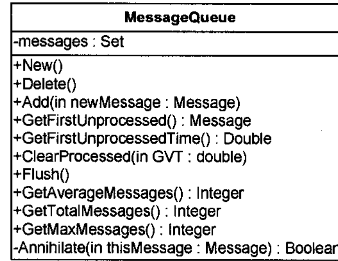


Figure 52. *MessageQueue* Class

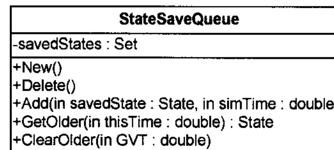


Figure 53. *StateSaveQueue* Class

the message can be annihilated. If not, it is added in timestamp order. The *MessageQueue* also stores all processed messages until a GVT computation by the simulation executive proclaims that it is safe to dispose processed messages older than the GVT. The *MessageQueue* class can also capture various statistics associated with itself.

4.4.4 *StateSaveQueue* Class

The *StateSaveQueue* class implements the ability to store the saved states of a *SimulationCluster* for the *Time Warp* implementation as shown in Figure 53. It retrieves a state older than a specific simulation time when a rollback needs to be performed. It disposes old states older than the GVT, once it is notified of a GVT computation.

4.5 Analytical Support within DIESEL

DIESEL provides considerable amount of support to analyze an application. This

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

support is built in to the *SimulationEngineComponent* and *SimulationCluster* interfaces. The "GetAveragePendingEventSetSize" and "GetAverageEventWaitTime" methods are used to gather various statistics associated with the pending event list within a single *SimulationEngineComponent*. The "GetAveragePendingEventSetSize" method captures the average length of the pending event list within a *SimulationEngineComponent* for the duration of the simulation. This helps to determine if there are too many *SimObjects* associated with the *SimulationEngineComponent*, and whether the *SimObjects* should be associated with new or other existing *SimulationEngineComponents*.

Similarly, the "GetAveragePendingEventSetSize" method within the *SimulationCluster* interface captures the average size of the pending event list within each *SimulationCluster*. This is an average of the size of the pending event lists of all *SimulationEngineComponents* within a *SimulationCluster*. This allows the application developer to analyze the current partitioning of the simulation model by demonstrating if a particular model component is being overloaded with work, while other components are underused within the simulation, thereby helping to develop better model partitions. The "GetAverageMessageQueueSize" method captures the average queue size of the communication buffer. In a distributed simulation, it is desirable to keep communication among processors at a minimum. This involves keeping *SimulationClusters* that interact with each other a lot on the same processor to avoid communication delays.

The interfaces specified in this chapter can be used to build a distributed application. The application can be executed using either a conservative or an optimistic synchronization algorithm. Different synchronization algorithms can be evaluated by implementing these algorithms within the interfaces provided. Various communication

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

mechanisms can also be used to connect the different *SimulationClusters* and networked processors within the distributed simulation.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter V

CASE STUDIES

This chapter describes how an implementation of the DIESEL behavioral model can be used for various purposes, both at the core simulation executive level as well as the application development level. An implementation of the DIESEL behavioral model has been developed in C++ as a proof-of-concept. All case studies described in this chapter have been conducted using the DIESEL C++ implementation.

A comparison study of various event management strategies is described in Section 5.1. A study demonstrating the number of messages passed among *SimulationClusters* on a distributed system is shown in Section 5.2. A comparison of the performance of optimistic and conservative synchronization algorithms within the same application is also discussed. The use of DIESEL in building a high-level application model is described in Section 5.3. Platform-independence of the DIESEL model is discussed in Section 5.4.

5.1 Event Management Study

The insertion of events into a pending event list and the removal of the event(s) with the shortest timestamp for execution should be as fast as possible. The insertion of events in time-sorted order makes the retrieval of the event with the shortest timestamp an $O(1)$ operation. Various techniques for managing the pending event list such as linear lists, binary trees, and priority queues exist in literature [54 – 61]. This section describes the event management structure used in the DIESEL implementation, which has been

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

implemented both as a linear and a non-linear list. Various strategies are described to make the insertion of events into the pending event list as close to an $O(1)$ operation as possible. A comparison study is then performed on the performance of these strategies. The comparison of these strategies on a single platform is made possible since the DIESEL behavioral model does not specify or promote a single method; the standardized interface of DIESEL, in fact, encourages employment and comparison of diverse strategies, as long as they satisfy the expected behavior of the *EventManager* within a *SimulationEngineComponent*.

5.1.1 Implementation of the Event Management Structure

The sequential DIESEL behavioral model described in Chapter III is implemented using the C++ programming language on a Win32 hardware platform. The DIESEL C++ implementation is fully representative of the behavioral model and includes the full functionality described in Chapters II and III. The C++ implementation uses the classes defined in Section 3.10 as the basic building blocks. Functionality has been added to these basic classes and new classes have been defined to satisfy the behavior expected of each component.

The *EventManager* class holds the collection of events to be executed on each *SimulationEngineComponent*. The event management structure for this study has been implemented both as a linear list and a non-linear list. Four different strategies are used to insert an event into a linear list within the *EventManager*. A binary tree structure is used to implement a non-linear list within the *EventManager*.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

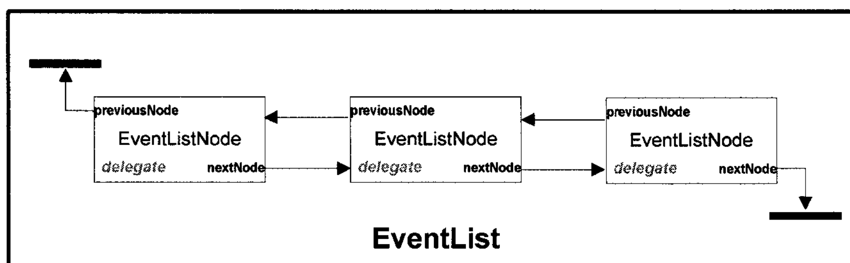


Figure 54. *EventList* Structure

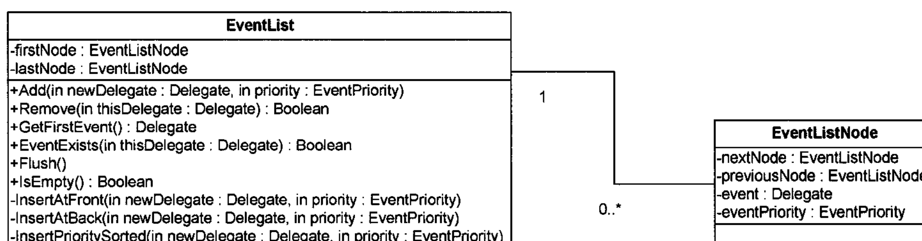


Figure 55. *EventList* and *EventListNode* Classes

EventList

An *EventList* is a linked list of events to be executed at a particular simulation time. The *EventList* is sorted based on relative priority of the events within the list, as shown in Figure 54. An *EventListNode* wraps an interface around each event to be integrated into the *EventList*. Each *EventListNode* is a unique event to be executed. The class declarations for an *EventListNode* and *EventList* are shown in Figures 55.

LinearPendingEventSet

A *LinearPendingEventSet* is a linked list of distinct simulation times, i.e. the scheduled execution times for events to be executed by the *SimObject*. A *LinearPendingEventSet* is shown in Figure 56. The "currentSimTime" and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

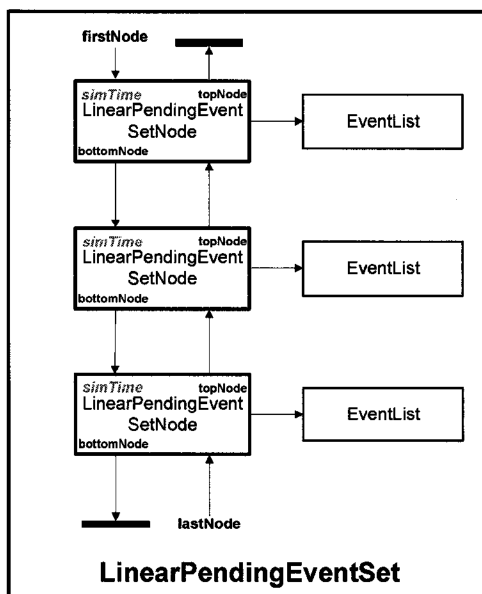


Figure 56. *LinearPendingEvent* Structure

"currentSimTimeList" attributes are used to manage events whose timestamps are the same as the current simulation time. The "ExecuteMinEvents" method within the *EventManager* removes the first *LinearPendingEventSetNode* and assigns its *EventList* to "currentSimTimeList". When an event is encountered with a timestamp equal to "currentSimTime", it is added to "currentSimTimeList", thereby avoiding inserting the event into the main list. When "currentSimTimeList" becomes empty, the *SimObject* has no more events to be executed at the current simulation time and returns the simulation time associated with its first *LinearPendingEventSetNode* to the *SimulationEngine*, to notify the *SimulationEngine* of the next minimum execution time of its remaining events. This information helps the *SimulationEngine* determine which *SimulationEngineComponent* to ask to execute its events next.

A *LinearPendingEventSetNode* wraps an interface around each simulation time to be integrated into the *LinearPendingEventSet*. Each *LinearPendingEventSetNode* is a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

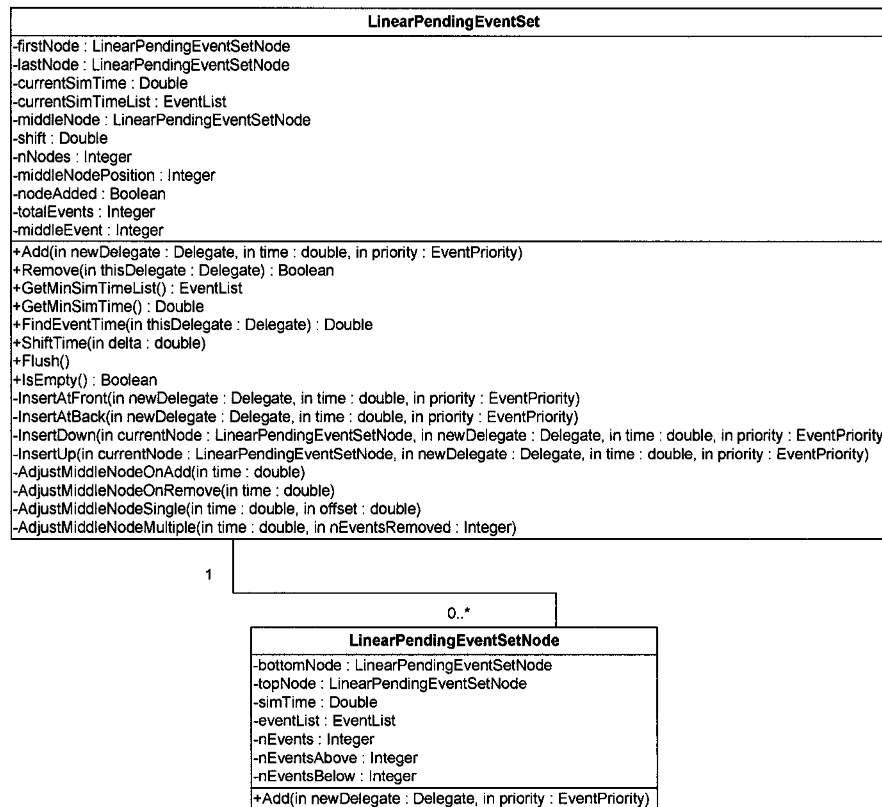


Figure 57. *LinearPendingEventSet* and *LinearPendingEventSetNode* Classes

unique simulation time and has at least one event to be executed at that simulation time by that *SimObject*. The class declarations for a *LinearPendingEventSetNode* and *LinearPendingEventSet* are shown in Figure 57.

NonLinearPendingEventSet

A *NonLinearPendingEventSet* is implemented as a binary tree, where the nodes in the tree are the simulation times of events. Each node, called a *NonLinearPendingEventSetNode*, has a list of events to be executed at that time, similar to a *LinearPendingEventSet* described earlier. The "Rank" method is used to sort the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

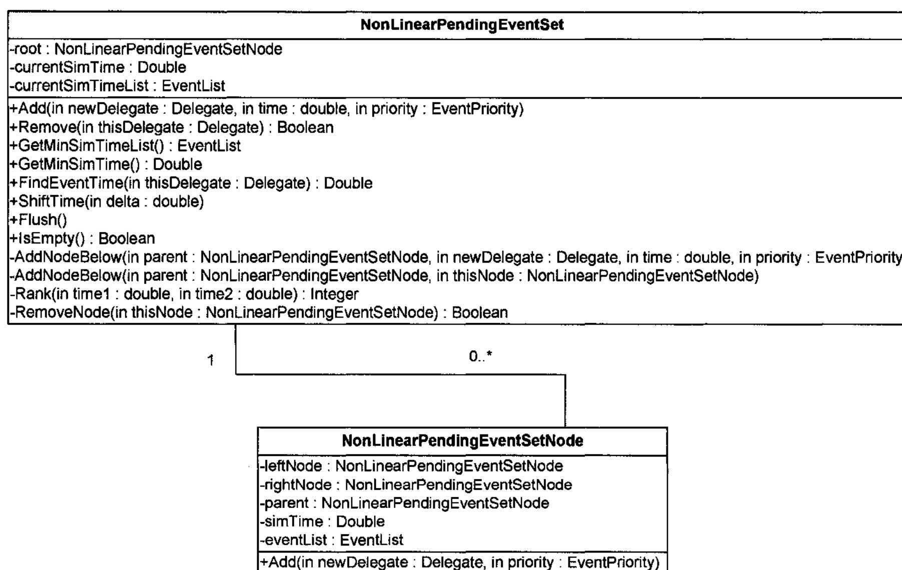


Figure 58. *NonLinearPendingEventSet* and *NonLinearPendingEventSetNode* Classes

nodes based on their simulation times, while inserting an event into the binary tree. The class declarations for a *NonLinearPendingEventSetNode* and *NonLinearPendingEventSet* are shown in Figure 58.

The event management structure here is used to study different strategies for inserting an event into the pending event list, both linear and non-linear. These strategies involve starting the search for an appropriate insertion point at different points in the event list and are described in the next sub-sections.

5.1.2 Event Insertion Strategies for a *LinearPendingEventSet*

Four strategies for inserting an event within the *LinearPendingEventSet* are described below. The first two strategies involve starting the search for an appropriate node at the head and tail of the list respectively, i.e., either from the "firstNode" of the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

LinearPendingEventSet or the "lastNode" of the *LinearPendingEventSet*. The remaining two strategies involve starting the search from some middle node in the list that is maintained within the list.

Insert from front of linear list

The search for an appropriate node to insert a new event starts from the first existing *LinearPendingEventSetNode* in the list. This is the most basic strategy and might be enough for a *LinearPendingEventSet* with a limited number of events. However, with a large number of events, it will take the maximum average time among all the strategies discussed here, since it is a fair assumption that new events will have timestamps closer to the tail of the list. This strategy starts the search at the head of the list and would therefore have to search through most of the nodes to find an appropriate insertion point.

Insert from rear of linear list:

The search for an appropriate node to insert a new event starts from the last existing *LinearPendingEventSetNode* in the list. This strategy is better than the previous strategy because it is a fair assumption that an appropriate insertion point will be at the tail of the list. This strategy is again appropriate for a limited number of events, where the overhead of maintaining and adjusting a middle node might increase the execution time without providing any substantial benefits.

Insert from a midpoint in linear list based on the number of events

The search for an appropriate node to insert a new event starts from a midpoint in

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the list. The midpoint is a node in the list such that approximately half the total number of events is above it and half below it, i.e., it is based on the number of the total events in the list. This strategy is based on the assumption that since previous events have been added to a particular node, there is a very high probability that a new event will be added to the same node. Thus, this strategy searches a sub list that contains half the total number of events in the list.

The "totalEvents", "middleEvent" attributes, and the "AdjustMiddleNodeSingle" and "AdjustMiddleNodeMultiple" methods within the *LinearPendingEventSet* class are used to adjust the midpoint. The "AdjustMiddleNodeSingle" method is used to adjust the middle node while adding or removing a single event from the list, where an offset of +1 indicates adding an event and –1 indicates removing an event. The "AdjustMiddleNodeMultiple" method is used to adjust the middle node when the top node (with multiple events) is removed for executing events at the current simulation time. The "nEvents", "nEventAbove", and "nEventsBelow" attributes within the *LinearPendingEventSetNode* class track the number of events relevant to each node.

The "shift" attribute within the *LinearPendingEventSet* indicates a variable bias (between 0.0 and 1.0) towards the rear end of the sub list. A "shift" of 1.0 starts the search at the head of the sub list, while a value of 0.0 starts the shift at the tail of the sub list. The lower the value of "shift" the more likely that the search for an appropriate node to insert an event will start at the tail end of the sub list and continue upward towards the head of the list. The middle node needs to be adjusted after each new event is added and after the top node is removed while executing events with the minimum execution time. The algorithm for inserting a new event into the list is:

```
Note:
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
  firstTime = simTime associated with first node in list
  lastTime  = simTime associated with last node in list
  midTime   = simTime associated with middle node
  newTime   = simTime of new event to be added
  shift     = bias towards either the front or the rear of the
              sublist.

Preconditions:
   None.
Post-conditions:
   1) Event is added to appropriate node.

Procedure
begin
    if (newTime < firstTime)
        add event to a new node at the head of the list
    else if (newTime = firstTime)
        add event to node at the head of the list
    else if (newTime > lastTime)
        add event to a new node at the tail of the list
    else if (newTime = lastTime)
        add event to  node at the tail of the list
    else
        if (newTime <= midTime)
            if ((time > (((midTime-firstTime)*shift)+firstTime))
                look up from middle node to insert event
            else
                look down from first node to insert event
        else
            if (time > (((lastTime-midTime)*shift)+midTime))
                look up from last node to insert event
            else
                look down from middle node to insert event
    Adjust the middle node
end
```

The algorithm for adjusting the middle node after adding an event is shown below:

```
Preconditions:
   1) nEvents for the node the event is added to has been
      incremented by 1
   2) totalEvents has been incremented by 1
Post-conditions:
   1) middleEvent is adjusted to appropriate value
   2) middleNode is adjusted to appropriate node

Procedure
begin
    newMiddleEvent = totalEvents DIV 2
    if (totalEvents MOD 2 != 0)
        newMiddleEvent++

    if (eventAdded.simTime < middleNode.simTime)
        middleNode.nEventsAbove++
    else
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
        middleNode.nEventsBelow++

    if (newMiddleEvent > middleNode.nEventsAbove) AND
       (newMiddleEvents <= (middleNode.nEventsAbove +
                            middleNode.nEvents))
        [no change in middleNode]
    else
        if (eventAdded.simTime < middleNode.simTime)
            newMiddleNode = middleNode.topNode
            newMiddleNode.nEventsAbove = middleNode.nEventsAbove
                  - newMiddleNode.nEvents
            newMiddleNode.nEventsBelow = middleNode.nEvents +
                  middleNode.nEventsBelow
        else
            newMiddleNode = middleNode.bottomNode
            newMiddleNode.nEventsAbove = middleNode->nEvents +
                  middleNode->nEventsAbove
            newMiddleNode.nEventsBelow = middleNode->nEventsBelow
                  - newMiddleNode->nEvents
        middleNode = newMiddleNode
    middleEvent = newMiddleEvent
end
```

The algorithm for adjusting the middle node after removing events is as shown below:

```
Preconditions:
   1) nEvents for the node the event(s) have been removed from
      has been decremented by appropriate value
   2) totalEvents for the LinearPendingEventSet has been
      decremented by appropriate value
Post-conditions:
   1) middleEvent is adjusted to appropriate value
   2) middleNode is adjusted to appropriate node

Procedure
begin
    newMiddleEvent = totalEvents DIV 2
    if (totalEvents MOD 2 != 0)
        newMiddleEvent++

    if (eventRemoved.simTime < middleNode.simTime)
        middleNode.nEventsAbove = middleNode.nEventsAbove -
            nEventsRemoved
    else
        middleNode.nEventsBelow = middleNode.nEventsBelow -
            nEventsRemoved

    while NOT ((newMiddleEvent > middleNode.nEventsAbove) AND
               (newMiddleEvents <= (middleNode.nEventsAbove +
                                    middleNode.nEvents)))
        if (eventRemoved.simTime <= middleNode.simTime)
            newMiddleNode = middleNode.bottomNode
            newMiddleNode.nEventsAbove = middleNode->nEvents +
                  middleNode->nEventsAbove
            newMiddleNode.nEventsBelow = middleNode->nEventsBelow
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
                    - newMiddleNode->nEvents
            else
                newMiddleNode = middleNode.topNode
                newMiddleNode.nEventsAbove = middleNode.nEventsAbove
                      - newMiddleNode.nEvents
                newMiddleNode.nEventsBelow = middleNode.nEvents +
                      middleNode.nEventsBelow
            middleNode = newMiddleNode
        middleEvent = newMiddleEvent
    end
```

Insert from a midpoint in linear list based on the time of events

The search for an appropriate node to insert a new event starts from a midpoint in the *LinearPendingEventSet*. The midpoint is a node in the list such that exactly half the nodes are above it and half below it, i.e., it is based on the times of the existing nodes in the list. Thus, this strategy searches a sub list that has half the number of nodes compared to the original list.

The "nNodes", "middleNodePosition", "nodeAdded", and "totalEvents" attributes and the "AdjustMiddleNodeOnAdd" method within the *LinearPendingEventSet* class are used to adjust the midpoint of the list after inserting an event. While adding an event to the *LinearPendingEventSet*, the middle node needs to be adjusted only if a new node is created, i.e., "nodeAdded" value is TRUE. The "AdjustMiddleNodeOnRemove" method adjusts the midpoint after event(s) are removed from the list.

The "shift" attribute is used in the same manner as the previous strategy, and the algorithm for inserting a new event into the list based on this strategy is the same as the previous strategy. The algorithm for adjusting the middle node after adding an event is shown below:

```
Preconditions:
   1) new node has been created and event added to it
Post-conditions:
   1) middleNodePosition value is adjusted to appropriate value
   2) middleNode is adjusted to appropriate node
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
Procedure
begin
    newMiddleNodePosition = nNodes DIV 2
    if (nNodes MOD 2 != 0)
        newMiddleNodePosition++

    if (newMiddleNodePosition NOT EQUAL TO middleNodePosition)
        if (time > midTime)
            middleNode = middleNode->bottomNode
        else
            middleNode = middleNode->topNode
    middleNodePosition = newMiddleNodePosition
end
```

The algorithm for adjusting the middle node after removing events is shown below:

```
Preconditions:
   1) new node has been removed
Post-conditions:
   1) middleNodePosition value is adjusted to appropriate value
   2) middleNode is adjusted to appropriate node

Procedure
begin
    newMiddleNodePosition = nNodes DIV 2
    if (nNodes MOD 2 != 0)
        newMiddleNodePosition++

    if (newMiddleNodePosition NOT EQUAL TO middleNodePosition)
        if (time < midTime)
            middleNode = middleNode->bottomNode
        else
            middleNode = middleNode->topNode
    middleNodePosition = newMiddleNodePosition
end
```

The last two strategies described here might perform similarly for most simulations. The strategy based on the number of events might perform better when there are simultaneous events, i.e., when multiple events are scheduled at the same discrete simulation times. This implies that new events are added to the same nodes and, therefore, the insertion procedure does not involve searching through a large number of nodes. On the other hand, the strategy based on the time of events might perform better when events are scheduled at distinct simulation times, with not many simultaneous

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

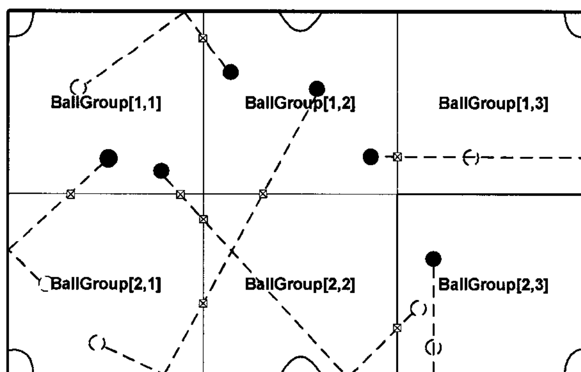


Figure 59. Snooker table divided into BallGroups

events during the simulation.

5.1.3 Timing Results

The event insertion strategies are tested using an example of a snooker table. The snooker table is spatially divided into regions called BallGroups as shown Figure 59, where each BallGroup constitutes a *SimObject.* Each BallGroup is assigned a certain number of balls and the balls are assigned initial velocities and positions within the region. The balls then move between regions or collide with the edges of the table, with both actions generating events within the same *SimObject* (if the ball collides with an edge) or on an adjoining *SimObject* (if the ball crosses into an adjoining BallGroup). The collisions between the balls themselves are ignored for this study.

The insertion strategies are tested by assigning initial values to the X-component and Y-component of the velocities of each ball from the following distributions:

- **Constant**
- **Exponential:** with mean values of 0.3, 0.5, and 0.7.
- **Triangular:** with minimum = 0, maximum = 1, and mid values of 0.25, 0.5, 0.75.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

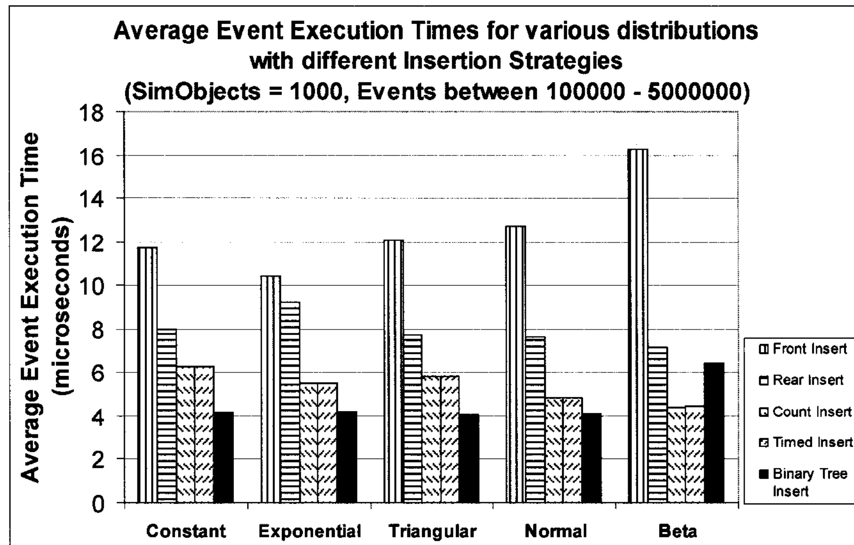
- **Normal:** with $\mu = 0.1, 0.2, 0.3$.
- **Beta: with** $\alpha_1 = 0.8, 1.5, 2, 5$ and $\alpha_2 = 0.8, 1.5, 2, 5$.

Assigning values from these distributions varies the velocities of each ball and in turn varies the time taken by the ball to either cross a BallGroup or collide with an edge. This generates events with timestamps mirroring the parent distribution that need to be added to the list. The aim is to calculate which of the strategies works best for all the distributions.

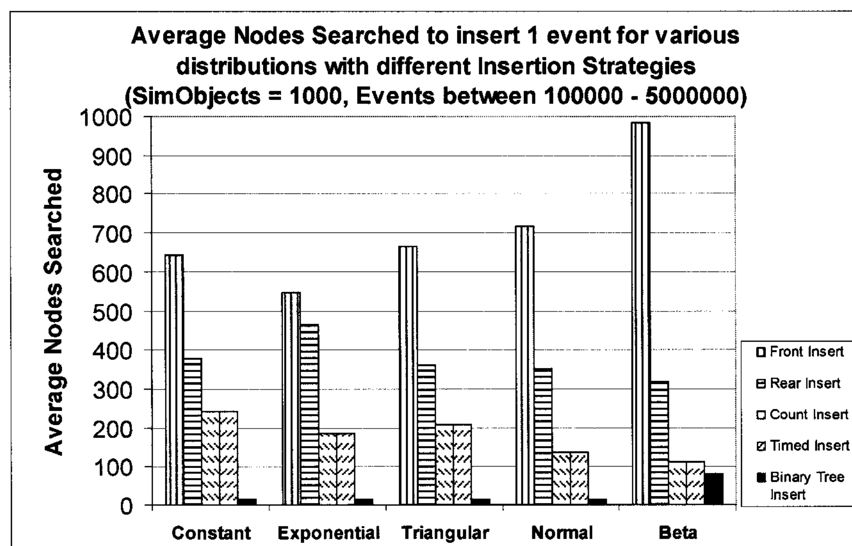
Readings are obtained by assigning 1000 balls each to 6 regions on the snooker table, and then varying the number of events to be executed between 100000 – 5000000. It is assumed that the time to execute the event is the same for all readings and the difference between execution times is due to the insertion times. The readings are normalized for each set of readings and then by calculating the average for each type of distribution. The readings for the insertion strategies based on the time of events and the number of events are taken for different shifts between 0.0 and 1.0. These readings indicate that a shift of 0.2 works best for the maximum number of distributions, implying that the search for the insertion point starts towards the tail end of the sub list. Therefore a shift of 0.2 is used in all subsequent graphs for comparison with the other two strategies.

Graph 2 shows the average execution time for a single event using each strategy described previously. The implementation of the pending event set as a binary tree proves to be the best strategy for all distributions except a Beta distribution. Graph 3 shows the graph of the average number of nodes searched to insert a single event using each strategy. The insertion of an event into a binary tree requires the least number of nodes to be searched. The number of nodes searched for the insertion strategies based on the time

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



Graph 2. Average execution time for different distributions



Graph 3. Average nodes searched for different distributions

of events and the number of events are identical due to the lack of many simultaneous events. In the absence of any simultaneous events, the insertion strategies based on the time and number of events are identical. The insertion strategies from the front and rear of the list perform the worst among all the strategies due to the large number of events in

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the study.

5.2 Distributed Simulation Study

In a distributed simulation, *SimulationClusters* communicate with each other by transmitting messages between each other. It is desirable to reduce network traffic by keeping communication among processors at a minimum. This involves keeping *SimulationClusters* that communicate with each other a lot on the same processor. This section describes an example to study the number of messages transmitted among *SimulationClusters*, when the application model is mapped to different organizations of *SimulationClusters*. The purpose of the study is to identify an organization that results in the least number of messages among *SimulationClusters*.

5.2.1 Distributed Implementation

The distributed DIESEL behavioral model described in Chapter IV is implemented using the C++ programming language on a Win32 hardware platform. The DIESEL C++ implementation is fully representative of the behavioral model and includes the full functionality described in Chapter IV. The sequential implementation is extended to support distributed simulations. In particular, support for the barrier synchronization and *Time Warp* synchronization algorithms has been added to the distributed implementation.

A set of procedures have been defined under the *DistributedExecutiveSupport* interface, shown in Figure 60, to support the distributed simulation executive to execute the simulation using the specified synchronization algorithm. These procedures are

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

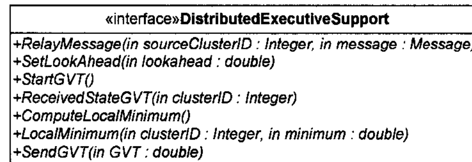


Figure 60. *DistributedExecutiveSupport* Interface

implementation-specific and are transparent to the application. A distributed simulation involves *SimulationClusters* transmitting and receiving messages between each other. If a *SimulationCluster* needs to send a message to another *SimulationCluster*, it calls the "RelayMessage" procedure to send the message to the appropriate *SimulationCluster*. *SimulationClusters* on the same processor are connected by the simulation executive on the processor, while simulation executives on different processors are connected using socket communication between processors. A procedure has been defined to support the *barrier synchronization* algorithm. The "SetLookAhead" procedure sets the estimated lookahead for the simulation.

The *Time Warp* algorithm requires regular computation of GVT, so that each *SimulationCluster* can clear its history information and reclaim memory assigned to it. The following procedures are defined to support the GVT computation.:

- The simulation executive uses the "StartGVT" procedure to notify each *SimulationCluster* to start a GVT computation.
- Each *SimulationCluster* on receiving notification, stops processing events and uses the "ReceivedStartGVT" procedure to notify the simulation executive that it has received notification of a GVT computation.
- After the simulation executive has received notification from all *SimulationClusters*, it uses the "ComputeLocalMinimum" procedure to ask each

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

SimulationCluster to calculate its local minimum time.

- After the simulation executive has received the local minimum from each *SimulationCluster*, it computes the GVT and sends it to each *SimulationCluster*, using the "SendGVT" procedure.
- Each *SimulationCluster* on receiving the GVT, clears all saved history information for timestamps which are in the past of the GVT, and resumes processing events.

5.2.2 Communication Study

The snooker table example described in the previous section is used to study communication among *SimulationClusters* in a distributed simulation. The snooker table is again spatially divided into 6 BallGroups, and these BallGroups transmit messages to each other while scheduling events on each other. The lookahead for the simulation is calculated as follows:

```
Let "length" be the length of the table
Let "width" be the breadth of the table
Let "XDistance" be the length of each BallGroup
Let "YDistance" be the width of each BallGroup
Let "XPosition" be the x-component of a ball's position within a
   BallGroup
Let "YPosition" be the y-component of a ball's position within a
   BallGroup
Let "XVelocity" be the x-component of the velocity assigned to a
   ball within a BallGroup
Let "YVelocity" be the y-component of the velocity assigned to a
   ball within a BallGroup

length    = 3569 mm
width     = 1178 mm
XDistance = length/3 ~ 1189 mm
YDistance = width/2 = 589 mm
XPosition = 1 ≤ XPosition ≤ 1189
YPosition = 1 ≤ YPosition ≤ 589
XVelocity = 10 mm/sec
YVelocity = 10 mm/sec
Lookahead = Minimum ((XPosition/XVelocity), (YPosition/YVelocity))
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

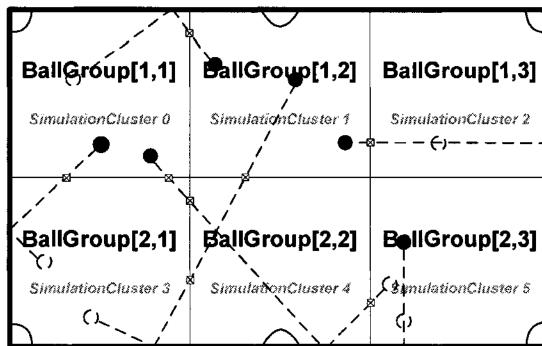


Figure 61a. 6 *SimulationClusters*

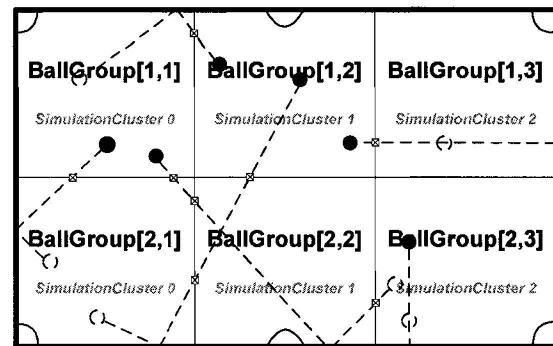


Figure 61b. 3 *SimulationClusters*

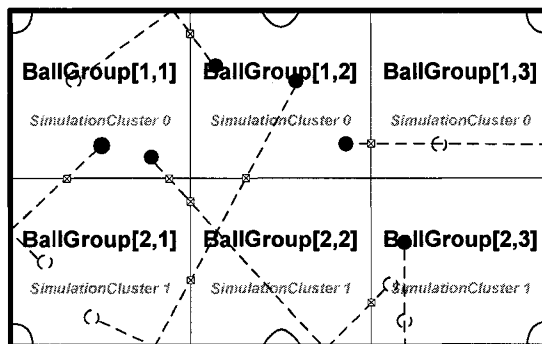


Figure 61c. 2 *SimulationClusters*

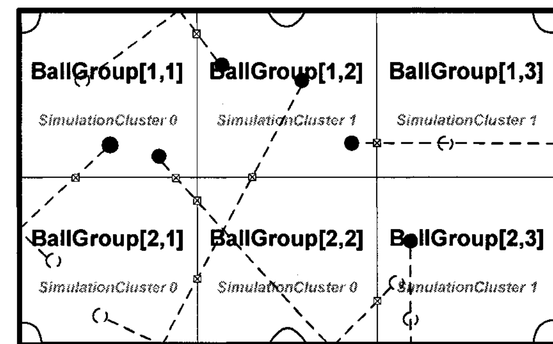


Figure 61d. 2 *SimulationClusters* with L-shaped organization

Figure 61. Mapping of BallGroups to *SimulationClusters*

```
= Minimum ((1/10), (1/10))
= 0.1
```

The BallGroups are assigned to different available *SimulationClusters*, and the messages that they send to each other is noted. The number of available *SimulationClusters* is varied from 2 to 6. The BallGroups are assigned to different *SimulationClusters* as shown in Figure 61. Each BallGroup is assigned a certain number of balls and the balls are assigned initial velocities and positions within the region, similar to the previous section.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Source SimulationCluster	Destination SimulationCluster					
	0	1	2	3	4	5
0	0	2192	0	6139	0	0
1	1972	0	2194	0	6222	0
2	0	2034	0	0	0	6212
3	6308	0	0	0	2042	0
4	0	6289	0	2111	0	2013
5	0	0	6276	0	2049	0

Table 1a. Messages passed

Simulation Cluster	Average Events
0	188
1	122
2	189
3	188
4	122
5	192

Table 1b. Average Events

Table 1. Messages passed and average pending events with 6 *SimulationClusters*

Source SimulationCluster	Destination SimulationCluster		
	0	1	2
0	0	4234	0
1	4083	0	4207
2	0	4083	0

Table 2a. Messages passed

Simulation Cluster	Average Events
0	188
1	122
2	190

Table 2b. Average Events

Table 2. Messages passed and average pending events with 3 *SimulationClusters*

5.2.3 Results

Table 1(a) shows the number of messages passed among different *SimulationClusters* with 6 *SimulationClusters*, while Table 1(b) shows the average size of the pending event set within each *SimulationCluster*. Table 2(a) shows the number of messages passed when the number of *SimulationClusters* is reduced to 3 and the BallGroups are assigned as shown in Figure 61 (b). This partition decreases the total number of messages passed among all *SimulationClusters* by 70%, while the average size of the pending event set remains the same. Table 3 shows the number of messages and the average events with 2 *SimulationClusters* as shown in Figure 61 (c). The average pending event size decreases, but the number of messages doubles as compared to 3

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Source SimulationCluster	Destination SimulationCluster	
	0	1
0	0	18573
1	18873	0

Table 3a. Messages passed

Simulation Cluster	Average Events
0	166
1	167

Table 3b. Average Events

Table 3. Messages passed and average pending events with 2 *SimulationClusters*

Source SimulationCluster	Destination SimulationCluster	
	0	1
0	0	10494
1	10243	0

Table 4a. Messages passed

Simulation Cluster	Average Events
0	165
1	167

Table 4b. Average Events

Table 4. Messages passed and average pending events with 2 *SimulationClusters* with L-shape organization

SimulationClusters. Table 4 shows the statistics when the BallGroups are reorganized within the same 2 *SimulationClusters*. The average size of the pending event size remains the same as the previous organization, but the total number of messages transmitted decreases by half.

The execution time for the simulation using the *Time Warp* algorithm is considerably slower than when using the barrier synchronization algorithm. This is expected, since the state capture ability within DIESEL captures the entire state of the simulation, even though there are few state variables that change a lot within this example. Similarly, a state restore operation due to rollback restores the state of the entire simulation. These are very detailed processes which account for most of the execution time. This state capture ability is specific to this implementation and can be replaced with other state saving techniques [6 – 11] that save the state of variables rather than the whole

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

system. With an application that has a lot of state variables which change often, the state capture ability within DIESEL would be a valuable tool to capture the system state, rather than attaching the overhead to each state variable.

5.3 Port Simulation (PORTSIM)

The Port Simulation (PORTSIM) system [62, 63, 64] is a discrete-event simulation that facilitates the analysis of movements of military unit equipment through worldwide seaports and allows for detailed infrastructure analysis. PORTSIM simulates the use of seaports within the defense cargo transportation system and assists planners in comparing and selecting ports. It determines port throughput capability given explicit assumptions on assets, resources, and scenarios. It tracks utilization of critical resources as well as potential bottlenecks and limiting resources to movement through the seaport.

The Programmable Process Flow Network (PPFN) is a process flow language developed at Old Dominion University to describe the process activities at a cargo terminal [65]. Previous versions of PORTSIM had processes statically coded into the simulation, preventing the analyst from modifying them when a specific scenario required it. By having the processes programmable, the analyst has full control of the simulation processes. PPFN is partitioned into a high level language where the analysts develop their processes and an assembly language that corresponds to data structures internal to the simulation. The assembly-level instructions are interpreted by a virtual machine called the *Process Engine*. This approach enables implementation of the simulation given an appropriate set of process instructions. Meanwhile, the high level language's grammar and semantics are further defined to suit the needs of the analyst

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

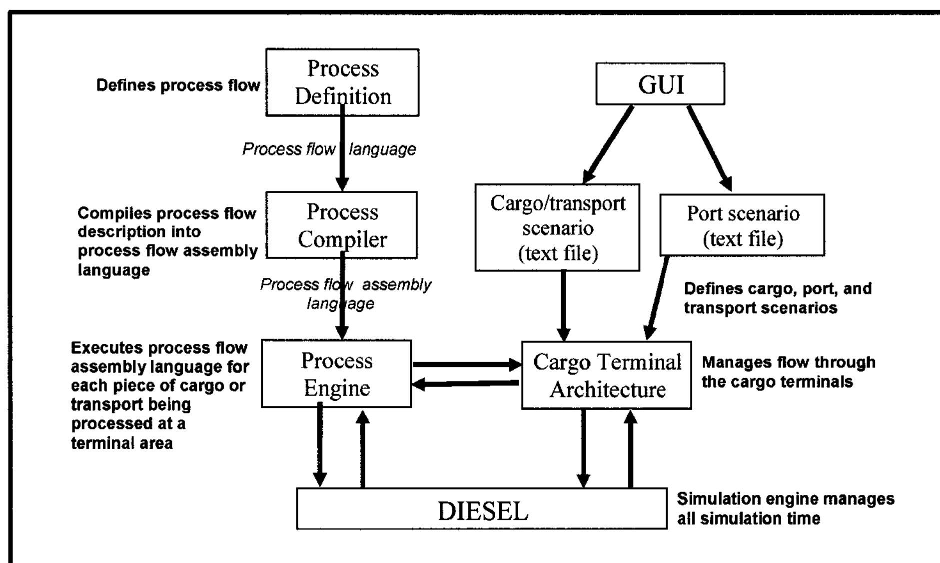


Figure 62. PORTSIM Architecture

without affecting simulation development.

Previous versions of PORTSIM were implemented in MODSIM, originally in MODSIM II and later in MODSIM III [17, 66, 67]. MODSIM is an object-oriented simulation language developed by CACI. However, CACI sold the rights to the language and the language has since been unsupported. This necessitated the movement of the simulation to another language. The Army decided on a general purpose language, requiring adopting a third party simulation executive (deemed undesirable for similar reasons to the MODSIM issues) or development of a simulation executive to support the project. This gave the first justification and opportunity for the development of DIESEL.

A functional prototype has been developed for the new PORTSIM simulation model with PPFN support, utilizing the DIESEL C++ implementation as the supporting simulation language with Windows XP OS on a Win32 platform. The current PORTSIM system is shown in Figure 62. The *Process Engine* and the *Cargo Terminal Architecture*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

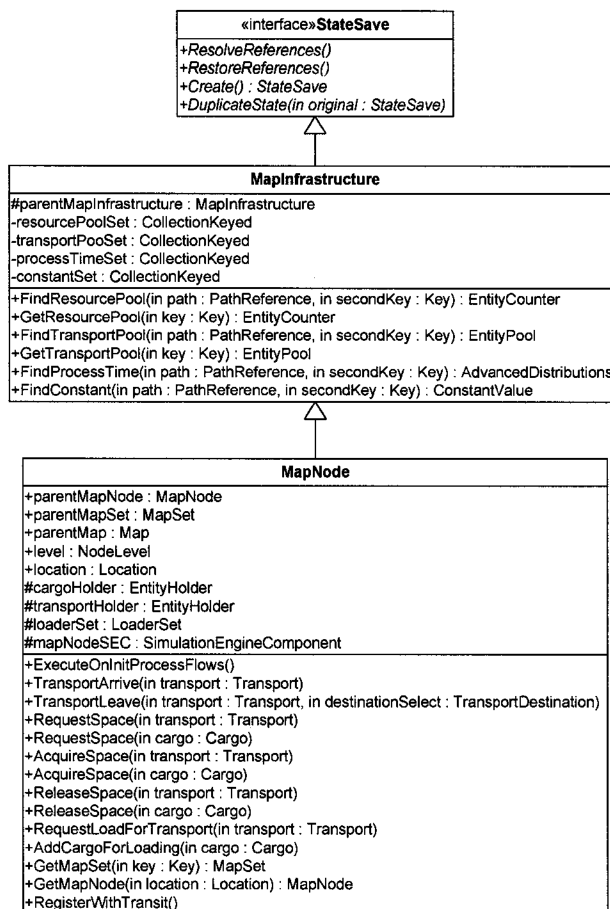


Figure 63. MapNode within PORTSIM Architecture

components within PORTSIM interact with the DIESEL simulation engine, to initialize, control, and execute the simulation. The *Cargo Terminal Architecture* within PORTSIM is defined as a network of nodes, with each node defined by a process or by another network thus allowing a hierarchical structure. A MapNode, shown in Figure 63, represents cargo terminals within the cargo transportation network as well as the individual terminal areas within a single cargo terminal. The architecture also supports the routing of entities (cargo and transports) through the network by defining a starting node, a destination node with a sequence of nodes to be traversed en route. Once a node that can handle a given entity is identified, the entity must request space in the selected

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

node before transiting to the node. This is important to avoid infinite queuing in the interconnection segments and to provide a level of flow control, thereby assisting in the avoidance of deadlock.

A MapNode is one of the many *SimObjects* within PORTSIM with an associated *SimulationEngineComponent*. Events are scheduled to be executed on a MapNode by the arrival of cargo and transport at the MapNode at a particular simulation time, through the routing infrastructure that exists between MapNodes. Requests for space by an entity within a destination MapNode also create events on the source MapNode, once space has been identified for the entity within the destination MapNode. Resources and transports within a MapNode are modeled with the *EntityCounter* and *EntityPool* interfaces in DIESEL, while cargo pools are modeled using other entity management interfaces within DIESEL.

The *Cargo Terminal Architecture* is a shell around the cargo processing model that is unique to each MapNode. A ProcessFlow is defined for each MapNode using PPFN to represent the cargo processing model. A ProcessFlow is also a *SimObject* with an associated *SimulationEngineComponent*. It is defined as a sequence of activities that must be traversed in order by cargo or transports to be processed through a MapNode. An example process flow is shown in Figure 64. Activities such as "ProcessTimeDelay" that involve passing simulation time, cause events to be scheduled on the associated *SimulationEngineComponent*. This causes the activity to be suspended until the specified simulation time, after which it resumes execution. Activities also use the *Trigger*, *TriggerCounter*, *Join*, and *JoinCounter* interfaces to achieve synchronization between themselves within a single ProcessFlow or multiple ProcessFlows.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

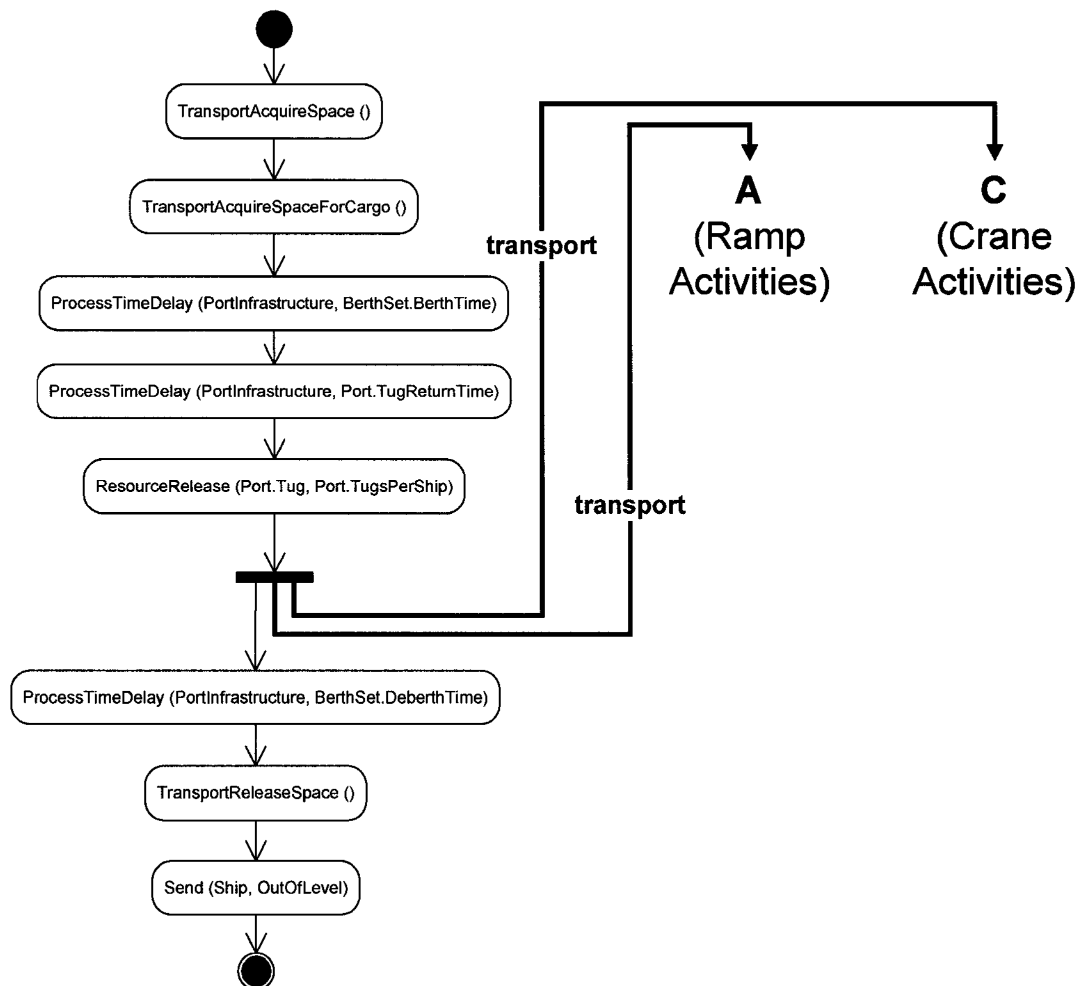


Figure 64. Example PORTSIM ProcessFlow using PPFN

The PORTSIM system uses the DIESEL state capture ability to save the state of the simulation and restore it at a later time. The state capture ability has also been used for debugging purposes, since a state save operation involves resolving all references within the simulation, which is extremely useful to find stranded references.

Results over the past four years have shown the DIESEL behavioral model and its C++ implementation to be more than adequate for supporting the development of real-world applications. An Initial Operational Capability (IOC) release of PORTSIM utilizing the sequential version of DIESEL is scheduled for June 2007.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.4 DIESEL Platform Independence

One of the main requirements of the DIESEL behavioral model is that it should not be focused on a particular programming language or hardware platform. The DIESEL behavioral model described in Chapter II, III and IV is defined at a high level of abstraction. The behavior expected of each component within the model has been captured. However, the implementation of the behavioral model is completely independent; each component can be implemented in a different manner, as long as they satisfy the behavior that is expected of that component. Each new implementation of a component should implement the interface that has been specified for that particular component. This allows each new component to be inserted into the complete model and cause no change to the behavior of the complete model.

Language Independence

The DIESEL behavioral model has been implemented in C++, Java and C#. The DIESEL C++ implementation is fully representative of the behavioral model and includes the full functionality described in Chapters II, III and IV. Java and C# implementations with the core simulation engine functionality have been developed as prototypes by Mr. Saurav Mazumdar from the Department of Electrical and Computer Engineering at Old Dominion University. These implementations prove that the behavioral model is not focused towards any language in particular. Complete Java and C# implementations can be developed depending on user requirements.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Compiler Independence

Various applications including the example timing diagram and the Dining Philosophers' problem described in Chapter III, using the DIESEL C++ implementation, have been tested on different C++ compliers. These compliers include the Visual Studio 2005 SP1 C++ compiler developed by Microsoft© Corporation and the GCC 4.1.1 compiler developed by Free Software Foundation in support of the GNU project using the Anjuta Integrated Development Environment (IDE).

Platform Independence

The example timing diagram and the Dining Philosophers' problem have been tested on the Windows 2000 and Windows XP operating systems from Microsoft© Corporation and the Fedora Core 6 operating system from Red Hat, installed on different hardware platforms, using the DIESEL C++ implementation and the GCC compiler.

The complete source code for the C++ implementation is made available for further research and development through the Department of Electrical and Computer Engineering at Old Dominion University. Source code is extremely valuable while debugging simulations that are developed using DIESEL as the underlying simulation executive. Moreover, one of the goals of this research is to encourage development of new implementations for a small component or for a set of interconnected components. This development is greatly aided if complete source code is available so that the developer can study the behavior of the component that is being replaced in the context of the entire model.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Should an application desire to use DIESEL without concern for the internal implementation, stable libraries that have been built for the complete implementation are made available through ODU too. These implementations can be used for developing applications with no changes or further development required within the underlying DIESEL implementation. This makes DIESEL an attractive option for commercial applications, since the simulation executive along with its implementation is free and easily modified to suit specific application needs.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter VI

CONCLUSION

This chapter concludes the work presented in this dissertation and offers suggestions on future work based on the behavioral model presented here.

6.1 Achievements

DIESEL specifies a behavioral model for a simulation executive and is capable of supporting sequential and distributed application development. The model is defined at a high level of abstraction and the expected behavior of each component within the model is described. The model also describes the interconnection among the model components. However, the model specification implies no implementation details. The implementation of each component or the whole behavioral model itself can be developed independently with no change to system behavior, as long as each component satisfies its expected behavior. The behavioral model is independent of any underlying network communication protocols or synchronization algorithms while executing a distributed application.

A complete and functional implementation of the behavioral model has been developed in C++. The C++ implementation is independent of the compiler that is used to compile the program or the hardware and software platform that it is executed on. Prototypical implementations of the model with core engine functionality have also been developed in Java and C#. Support for the conservative barrier synchronization algorithm

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

and the optimistic *Time Warp* algorithm has been implemented to show that the model can support different synchronization algorithms.

The behavioral model for a simulation executive described in this dissertation can be used for research and development into different aspects of PDES. The simulation executive in its current form can be used to develop discrete-event simulations. The underlying implementation can be changed with no changes required in the application. Different event management strategies, communication mechanisms, and synchronization algorithms, both new and existing, can be compared using the same behavioral model. The model described here and its implementation can be used in classes to teach the basics of a sequential and distributed simulation. It is also extremely stable and can therefore be the underlying simulation executive for commercial applications.

6.2 Enhancements

Several enhancements are possible to the work described in this dissertation:

High Level Architecture (HLA) Compliance

The DIESEL simulation executive could be extended so that any applications developed using DIESEL could be written to be HLA compliant, if needed. HLA compliance could be achieved by specifying the HLA Simulation Object Model (SOM) for DIESEL and its interface to the Runtime Infrastructure (RTI). An application would then need to develop and specify its Federation Object Model (FOM) to achieve compliance. Simulations built using DIESEL (called federates in HLA terminology) would have to use the time management services of the RTI to manage local time in order

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

to be HLA compliant. This would involve the DIESEL time management infrastructure interacting with the RTI to manage and advance the simulation time within an application, which could involve extensions to the DIESEL interface specification.

Support For Continuous-Time Simulations

The DIESEL simulation executive currently supports discrete-event simulations. The behavioral model can be extended to support continuous-time simulations. Simulation time within continuous-time simulations is non-discrete. Therefore, the simulation time would have to be incremented by delta values to advance the simulation, thereby simulating the non-discrete nature of a continuous simulation. The *Delayed State Commitment* that already exists within the current specification can be an important tool. If multiple processes modify the same state variable at the same simulation time, then the order in which the processes act on the variable plays an important role in the final outcome of the simulation. *Delayed State Commitment* makes all processes act on the same value of the state variable and then updates the state variable with its new value after all processes have finished executing at that simulation time.

Support For Process Flow Modeling

The DIESEL model can also be extended to support process flow modeling, which represents a system as a Data Flow Diagram (DFD) [68]. A DFD shows the flow of information through the system using processes where each process transforms a set of inputs to a set of outputs. The DFD can be mapped onto DIESEL by converting the DFD

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

to an event-driven model, with each event representing the transformation from input to output within a process.

Support for Real-Time Simulations and Animation

Another extension to the DIESEL model could be support for real-time simulations and animation within a simulation. The behavioral model currently executes as fast as possible. This execution would have to be slowed down to synchronize it with the wallclock time, such that each event execution would correspond to an activity within real-time. The application model would have to be defined in such a way that a single event computation takes no longer than the time it would take to happen in real-time. If an event is executed faster than real-time, then the simulation would have to be suspended until the wallclock time caught up with the simulation time.

Animation support can be built in as an after-the-fact effort by using DIESEL to keep track of each event that an application object executed. This data could then be used to represent the simulation and the entity interaction within the application.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

REFERENCES

[1] Fujimoto, R.M. "Parallel and Distributed Simulation Systems." *Wiley Series on Parallel and Distributed Computing*, 2000.

[2] Fujimoto, R.M. "Parallel Discrete Event Simulation." *Communications of the ACM* 33, no. 10 (October 1990): 30–53.

[3] Lamport, L. "Time, Clocks, and the Ordering of Events in a Distributed System." *Communications of the ACM* 21, no. 7 (July 1978): 558–565.

[4] Jefferson D. "Virtual Time." *ACM Transactions on Programming Languages* 7, no. 3 (July 1985): 405–425.

[5] Frey, P., R. Radhakrishnan, H.W. Carter, P.A. Wilsey, and P. Alexander. "A Formal Specification and Verification Framework for Time Warp-Based Parallel Simulation." *IEEE Transactions on Software Engineering* 28, no. 1 (January 2002): 58–78.

[6] Ronngren, R., M. Liljenstam, R. Ayani, and J. Montagnat. "Transparent Incremental State Saving in Time Warp Parallel Discrete Event Simulation." In *Proc. 10th Workshop on Parallel and Distributed Simulation*, 70–77, 1996.

[7] Lin, Y.B., B.R. Preiss, W.M. Loucks, and E.D. Lazowska. "Selecting the Checkpoint Interval in Time Warp Simulation." In *Proc. 7th Workshop on Parallel and Distributed Simulation*, 3–10, 1993.

[8] Ronngren, R., and R. Ayani. "Adaptive Checkpointing in Time Warp." In *Proc. 8th Workshop on Parallel and Distributed Simulation*, 110–117, 1994.

[9] Quaglia, F. "Combining Periodic and Probabilistic Checkpointing in Optimistic

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Simulation." In *Proc. 13th Workshop on Parallel and Distributed Simulation*, 109–116, 1999.

[10] Quaglia, F. "Event History Based Sparse State Saving in Time Warp." In *Proc. 12th Workshop on Parallel and Distributed Simulation*, 72–79, 1998.

[11] Carothers, C.D., K.S. Perumalla, and R.M. Fujimoto. "Efficient Optimistic Parallel Simulations using Reverse Computation." *ACM Transactions on Modeling and Computer Simulation* 9, no. 3 (July 1999): 224–253.

[12] Chandy, K.M., and L. Lamport. "Distributed Snapshots: Determining Global States of Distributed Systems." *ACM Transactions on Computer Systems* 3, no. 1 (February 1985): 63–75.

[13] Samadi, B. "Distributed Simulation, Algorithms and Performance Analysis." PhD thesis, Computer Science Department, University of California, Los Angeles, 1985.

[14] Mattern, F. "Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation." *Journal of Parallel and Distributed Computing* 18, no. 4 (August 1993): 423–434.

[15] Zeigler, B.P. "Hierarchical, Modular Discrete-Event Modelling in an Object-Oriented Environment." *Simulation* 49, no. 5 (November 1987): 219–230.

[16] Eldredge, D.L., J.D. McGregor, and M.K. Summers. "Applying the Object-Ooriented Paradigm to Discrete Event Simulations using the C++ language." *Simulation* 54, no. 2 (February 1990): 83–91.

[17] Rich, D. O., and R. E. Michelsen. "An Assessment of the MODSIM/TWOS Parallel Simulation Environment." In *Proc. 1991 Winter Simulation Conference*,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

509–518, 1991.

[18] Dimsdale, B., and H.M. Markowitz. "A Description of the SIMSCRIPT Language." *IBM Systems Journal* 38, no. 2-3 (1999): 151–161.

[19] Price, R. N., and C. R. Harrell. "Simulation Modeling and Optimization using ProModel." In *Proc. 1999 Winter Simulation Conference,* 208–214, 1999.

[20] Bapat, V., and N. Swets. "The Arena Product Family: Enterprise Modeling Solutions." In *Proc. 2000 Winter Simulation Conference*, 163–169, 2000.

[21] Fritzson, P., and V. Engelson. "Modelica - A Unified Object-Oriented Language for System Modelling and Simulation." *Lecture Notes In Computer Science* 1445 (1998): 67–90.

[22] Bagrodia, R.L, and W.T. Liao. "Maisie: A Language for the Design of Efficient Discrete-Event Simulations." *IEEE Transactions on Software Engineering* 20, no. 4 (April 1994): 225–238.

[23] Waldorf, J., and R.L. Bagrodia. "MOOSE: A Concurrent Object-Oriented Language for Simulation." *International Journal in Computer Simulation* 4, no. 2 (1994): 235–257.

[24] Wonnacott, P., and D. Bruce. "The APOSTLE Simulation Language: Granularity Control and Performance Data." In *Proc. 10th Workshop on Parallel and Distributed Simulation*, 114–123, 1996.

[25] Pham, C.D., and R.L. Bagrodia. "Building Parallel Time-constrained HLA Federates: A Case Study with the Parsec Parallel Simulation Language." In *Proc.* 1998 Winter Simulation Conference, 1555–1562, 1998.

[26] Zhang, Y., W. Cai and S.J. Turner. "A Parallel Object-Oriented Manufacturing

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Simulation Language." In *Proc. 15th Workshop on Parallel and Distributed Simulation*, 101–108, 2001.

[27] Kilgore, R.A., and E. Burke. "Object-Oriented Simulation of Distributed Systems using Java® and Silk®." In *Proc.* 2000 Winter Simulation Conference, 1802–1809, 2000.

[28] Lomov, G., and D. Baezner. "A Tutorial Introduction to Object-Oriented Simulation and Sim++." In *Proc.* 1990 Winter Simulation Conference, 149–153, 1990.

[29] Zeigler, B.P. "Multifacetted Modeling and Discrete Event Simulation" *Academic Press*, 1984.

[30] Concepcion, A.I., and B.P. Zeigler. "DEVS Formalism: A Framework for Hierarchical Model Development." *IEEE Transactions on Software Engineering* 14, no. 2 (February 1988): 228–241.

[31] Das, S.R., R.M. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. "GTW: A Time Warp system for Shared Memory Multiprocessors." In *Proc. 1994 Winter Simulation Conference*, 1332–1339, 1994.

[32] Steinman, J.S. "SPEEDES: A Multiple-Synchronization Environment for Parallel Discrete-Event Simulation." *International Journal in Computer Simulation* 2, no. 3 (1992): 251–286.

[33] Calvin, J., A. Dickens, B. Gaines, P. Metzger, D. Miller, and D. Owen. "The SIMNET Virtual World Architecture." In *Proc. IEEE Virtual Reality Annual International Symposium*, 450–455, 1993.

[34] Miller, D.C., and J.A. Thorpe. "SIMNET: The Advent of Simulator Networking."

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Proceedings of the IEEE 83, no. 8 (1995): 1114–1123.

[35] *IEEE Standard for Distributed Interactive Simulation – Application Protocols.* IEEE Standard 1278.1-1995, 1995.

[36] Dahmann, J. S. "High Level Architecture for Simulation." In *Proc. 1st International Workshop on Distributed Interactive Simulation and Real-Time Applications,* 9–14, 1997.

[37] Dahmann, J. S., R. M. Fujimoto and R. M. Weatherly. "The Department of Defense High Level Architecture." In *Proc. 1997 Winter Simulation Conference,* 142–149, 1997.

[38] Fujimoto, R. M., and R. M. Weatherly. "Time Management in the DoD High Level Architecture." In *Proc. 10th workshop on Parallel and Distributed Simulation,* 60–67, 1996.

[39] Christopher, W.A., S.J. Procter, and T.E. Anderson. "The Nachos Instructional Operating System." *Technical Report: CSD-93-739*, University of California at Berkeley, 1993.

[40] Ives, B., S. Hamilton, and G.B. Davis. "A Framework for Research in Computer-Based Management Information Systems." *Management Science* 26, no. 9 (1980): 910–934.

[41] Geotechnical Software Services. *C++ Programming Style Guidelines*. (online at http://geosoft.no/development/cppstyle.html)

[42] *UPC Language Specifications, v1.2*. UPC Consortium, Lawrence Berkeley National Lab Tech Report LBNL-59208, 2005.

[43] Fog, A. "Uniform Random Number Generators." (online at

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

http://www.agner.org/random/randoma.htm)

[44] L'Ecuyer, P. "Fast Combined Multiple Recursive Generators with Multiplies of the form $a=2^q \pm 2^r$." In *Proc. 2000 Winter Simulation Conference*, 683–689, 2000.

[45] Frank, P. "A Generic Object-Oriented Random Variate Model for Discrete Event Simulations: Separating the Data Model from the Process Model." Master's thesis, Department of Electrical and Computer Engineering, Old Dominion University, December 1999.

[46] Gullapalli S. "An Object-Oriented Resource Pool Model in Support of Discrete Event Simulations." Master's thesis, Department of Electrical and Computer Engineering, Old Dominion University, May 2001.

[47] Dijkstra, E. W. "Hierarchical Ordering of Sequential Processes." *Acta Informatica* 1, no. 2 (June 1971): 115–138.

[48] Law, K.L.E., and R. Leung. "A Design and Implementation of Active Network Socket Programming." In *Proc. 11th International Conference on Computer Communications and Networks*, 78–83, 2002.

[49] Karonis, N.T., B. Toonen, and I. Foster. "MPICH-G2: A Grid-enabled Implementation of the Message Passing Interface." *Journal of Parallel and Distributed Computing* 63, no. 5 (May 2003): 551–563.

[50] Wang, Y.M., O.P. Damani and W.J. Lee. "Reliability and Availability Issues in Distributed Component Object Model (DCOM)." In *Proc. 4th International Workshop on Community Networking Proceedings*, 59–63, 1997.

[51] Vinoski, S. "CORBA: Integrating Diverse Applications within Distributed Heterogeneous Environments." *IEEE Communications Magazine* 35, no. 2

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(February 1997): 46–55.

[52] Frantisek P., and M. Stal. "An Architectural View of Distributed Objects and Components in CORBA, Java RMI and COM/DCOM." *Software - Concepts & Tools* 19, no. 1 (June 1998): 14–28.

[53] Avril, H., and C. Tropper. "On Rolling Back and Checkpointing in Time Warp." *IEEE Transactions on Parallel and Distributed Systems* 12, no. 11 (November 2001): 1105–1121.

[54] Franta, W.R., and K. Maly. "An Efficient Data Structure for the Simulation Event Set." *Communications of the ACM* 20, no. 8 (August 1977): 596–602.

[55] Ulrich, E.G. "Event Manipulation for Discrete Simulations Requiring Large Numbers of Events." *Communications of the ACM* 21, no. 9 (September 1978): 777–785.

[56] Franta, W.R., and K. Maly. "A Comparison of Heaps and the TL structure for the Simulation Event Set." *Communications of the ACM* 21, no. 10 (October 1978): 873–875.

[57] McCormack, W.M. "Analysis of Future Event Set Algorithms for Discrete Event Simulation." *Communications of the ACM* 24, no. 12 (December 1981), 801–812.

[58] Jones, W.D. "An Empirical Comparison of Priority-Queue and Event-Set Implementations." *Communications of the ACM* 29, no. 4 (April 1986): 300–311.

[59] Brown, R. "Calendar Queues: A Fast 0(1) Priority Queue implementation for the Simulation Event Set Problem." *Communications of the ACM* 31, no. 10 (October 1988): 1220–1227.

[60] Marin, M. "On the Pending Event Set and Binary Tournaments." *Technical*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Report, Oxford University, 1997.

[61] Dahl, J., M. Chetlur and P.A. Wilsey. "Event List Management In Distributed Simulation." In *Proc. 7th International Euro-Par Conference on Parallel Processing*, 466–475, 2001.

[62] Nevins, M., C. Macal and J. Joines. "PORTSIM: An Object-Oriented Port Simulation." In *Proc. Summer Computer Simulation Conference (SCSC 1995)*, 160–165, 1995.

[63] Leathrum, J.F. and T. Frith. "A Reconfigurable Object Model for Port Operations." In *Proc. Summer Computer Simulation Conference (SCSC 2000)*, 693–698, 2000.

[64] Leathrum, J.F. "PORTSIM: Model Visibility for Training within a Port Simulation." In *Proc. CAORF/JSACC 2000*, L1-1–L1-12, 2000.

[65] Cuckov, F. "The Programmable Process Flow Network (PPFN)." Master's thesis, Department of Electrical and Computer Engineering, Old Dominion University, May 2005.

[66] *MODSIM-III Reference Manual*. CACI Products Company, 2000.

[67] *MODSIM-III Tutorial*. CACI Products Company, 2000.

[68] Arndt, T., and A. Guercio. "Decomposition of Data Flow Diagrams." In *Proc. 4th International Conference on Software Engineering and Knowledge Engineering*, 560–566, 1992.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPENDICES

A. DIESEL Sequential Interface Specification

A.1 State Capture Interfaces

A.1.1 *StateSave* Interface

Interface Diagram:

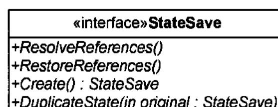


Figure A1. *StateSave* Interface

Methods:

- **ResolveReferences**

  Objective: This method resolves references for dynamically initialized attributes within an entity. This method should be defined for each class or interface that inherits the StateSave interface.

  Parameters: None.

  Return Value: None.

  Exceptions: None.

- **RestoreReferences**

  Objective: This method restores references for dynamically initialized attributes within an entity. This method should be defined for each class or interface that inherits the StateSave interface.

  Parameters: None.

  Return Value: None.

  Exceptions: None.

- **Create**

  Objective: This method creates an empty shell of an entity during the state duplication process of a non-terminating state save operation. This method should be defined for each class or interface that inherits the StateSave interface.

  Parameters: None.

  Return Value:

  Type: StateSave.

  Function: Empty shell of entity.

  Exceptions: None.

- **DuplicateState**

  Objective: This method copies the state of "original" to the current entity during the state duplication process of a non-terminating state save operation. This

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

method should be defined for each class or interface that inherits the StateSave interface.

Parameters:
original
Mode: Input.
Type: StateSave.
Presence: Required.
Function: Specifies entity whose state is to be copied.
Return Value: None.
Exceptions: None.

A.1.2 *StateSaveSupport* Interface

Interface Diagram:

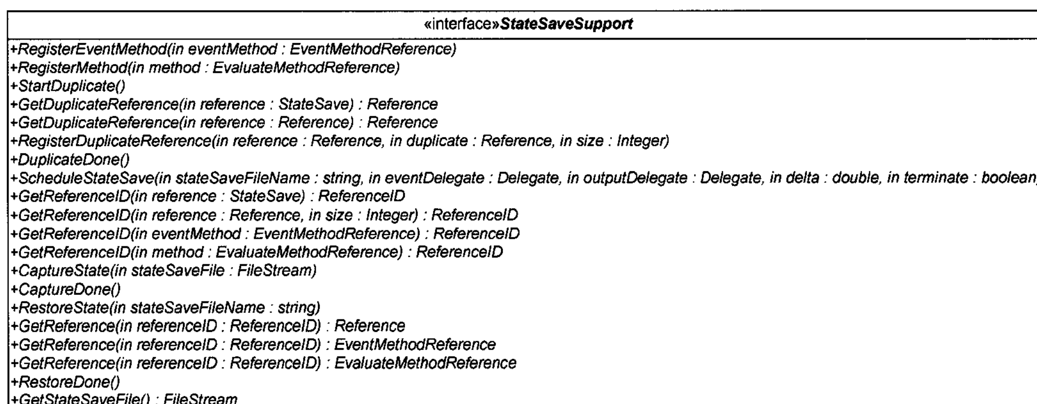


Figure A2. *StateSaveSupport* Interface

Methods:

- **RegisterEventMethod**

Objective: This method registers an event method with the state save support infrastructure.
Parameters:
eventMethod
Mode: Input.
Type: EventMethodReference.
Presence: Required.
Function: Specifies the event method to be registered.
Return Value: None.
Exceptions: None.

- **RegisterMethod**

Objective: This method registers a method that evaluates two elements with the state save support infrastructure.
Parameters:
method
Mode: Input.
Type: EvaluateMethodReference.
Presence: Required.
Function: Specifies the method to be registered.
Return Value: None.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Exceptions: None.

- **StartDuplicate**
  Objective: This method initializes a state duplication process for a non-terminating state save operation.
  Parameters: None.
  Return Value: None.
  Exceptions: None.

- **GetDuplicateReference**
  Objective: This method returns the reference for the duplicate of a StateSave object.
  Parameters:
  reference
  Mode: Input.
  Type: StateSave.
  Presence: Required.
  Function: Specifies the original StateSave object.
  Return Value:
  Type: Reference.
  Function: Reference to the duplicate of the StateSave object.
  Exceptions: None.

- **GetDuplicateReference**
  Objective: This method returns the reference for the duplicate of a non-StateSave object.
  Parameters:
  reference
  Mode: Input.
  Type: Reference.
  Presence: Required.
  Function: Specifies the original non-StateSave object.
  Return Value:
  Type: Reference.
  Function: Reference to the duplicate of the non-StateSave object.
  Exceptions: None.

- **RegisterDuplicateReference**
  Objective: This method registers a non-StateSave object and its duplicate with the state save support infrastructure.
  Parameters:
  reference
  Mode: Input.
  Type: Reference.
  Presence: Required.
  Function: Specifies the original non-StateSave object.
  duplicate
  Mode: Input.
  Type: Reference.
  Presence: Required.
  Function: Specifies the duplicate of the original non-StateSave object.
  size

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
      Mode:      Input.
      Type:      Integer.
      Presence:  Required.
      Function:  the size of the non-StateSave object.
Return Value:    None.
Exceptions:      None.
```

- **DuplicateDone**

```
Objective:       This method ends and cleans up a state duplication
                 process for a non-terminating state save operation.
Parameters:      None.
Return Value:    None.
Exceptions:      None.
```

- **ScheduleStateSave**

```
Objective:       This method schedules a state save operation at an
                 offset of "delta" from the current simulation time.
Parameters:
   stateSaveFileName
      Mode:      Input.
      Type:      String.
      Presence:  Required.
      Function:  Specifies the name of the state save file.
   eventDelegate
      Mode:      Input.
      Type:      Delegate.
      Presence:  Required.
      Function:  Specifies the delegate to be executed to save the
                 state of the simulation.
   dataDelegate
      Mode:      Input.
      Type:      Delegate.
      Presence:  Required.
      Function:  Specifies the delegate to be executed to save the
                 data within the application.
   delta
      Mode:      Input.
      Type:      Double.
      Presence:  Required.
      Function:  Specifies the offset of state save operation from
                 current simulation time.
   terminate
      Mode:      Input.
      Type:      Boolean.
      Presence:  Required.
      Function:  Specifies if the simulation is to be terminated at
                 the end of the state save operation.
Return Value:    None.
Exceptions:      None.
```

- **GetReferenceID**

```
Objective:       This method registers a StateSave object with the
                 state save support infrastructure.
Parameters:
   reference
      Mode:      Input.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
      Type:      StateSave.
      Presence:  Required.
      Function:  Specifies the StateSave object to be registered.
Return Value:
      Type:      ReferenceID.
      Function:  ID of the StateSave object.
Exceptions:      None.
```

- **GetReferenceID**

```
Objective:       This method registers a non-StateSave object with
                 the state save support infrastructure.
Parameters:
   reference
      Mode:      Input.
      Type:      Reference.
      Presence:  Required.
      Function:  Specifies the non-StateSave object to be registered.
   size
      Mode:      Input.
      Type:      Integer.
      Presence:  Required.
      Function:  Specifies the size of the non-StateSave object.
Return Value:
      Type:      ReferenceID.
      Function:  ID of the non-StateSave object.
Exceptions:      None.
```

- **GetReferenceID**

```
Objective:       This method returns the ID for an event method.
Parameters:
   eventMethod
      Mode:      Input.
      Type:      EventMethodReference.
      Presence:  Required.
      Function:  Specifies the event method.
Return Value:
      Type:      ReferenceID.
      Function:  ID of the event method.
Exceptions:      None.
```

- **GetReferenceID**

```
Objective:       This method returns the ID for an evaluate method.
Parameters:
   method
      Mode:      Input.
      Type:      EvaluateMethodReference.
      Presence:  Required.
      Function:  Specifies the method.
Return Value:
      Type:      ReferenceID.
      Function:  ID of the method.
Exceptions:      None.
```

- **CaptureState**

```
Objective:       This method initiates writing the simulation state
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
                 to the state save file for a state save operation.
Parameters:
   stateSaveFile
      Mode:      Input.
      Type:      File Stream.
      Presence:  Required.
      Function:  State Save File.
Return Value:    None.
Exceptions:      None.
```

- **CaptureDone**

```
Objective:       This method ends and cleans up a state save
                 operation.
Parameters:      None.
Return Value:    None.
Exceptions:      None.
```

- **RestoreState**

```
Objective:       This method initializes the support infrastructure
                 for a state restore operation and initiates a state
                 restore operation from the specified state save
                 file.
Parameters:
   stateSaveFileName
      Mode:      Input.
      Type:      String.
      Presence:  Required.
      Function:  Specifies the name of the state save file.
Return Value:    None.
Exceptions:      None.
```

- **GetReference**

```
Objective:       This method restores the reference for an object
                 registered with the state support infrastructure.
Parameters:
   referenceID
      Mode:      Input.
      Type:      ReferenceID.
      Presence:  Required.
      Function:  Specifies the registered object.
Return Value:
      Type:      Reference.
      Function:  Reference to the registered object.
Exceptions:      None.
```

- **GetReference**

```
Objective:       This method restores the reference for an event
                 method registered with the state support
                 infrastructure.
Parameters:
   referenceID
      Mode:      Input.
      Type:      ReferenceID.
      Presence:  Required.
      Function:  Specifies the ID of the registered event method.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
Return Value:
      Type:     EventMethodReference.
      Function: Reference to the registered event method.
Exceptions:     None.
```

- **GetReference**

```
Objective:      This method restores the reference for an evaluate
                method registered with the state support
                infrastructure.
Parameters:
   referenceID
      Mode:     Input.
      Type:     ReferenceID.
      Presence: Required.
      Function: Specifies the ID of the registered method.
Return Value:
      Type:     EvaluateMethodReference.
      Function: Reference to the registered method.
Exceptions:     None.
```

- **RestoreDone**

```
Objective:      This method ends and cleans up a state restore
                operation.
Parameters:     None.
Return Value:   None.
Exceptions:     None.
```

- **GetStateSaveFile**

```
Objective:      This method returns the state save file to the
                simulation application, to save or retrieve
                information.
Parameters:     None.
Return Value:   None.
      Type:     File Stream.
      Function: State Save File.
Exceptions:     None.
```

A.2 Application Support Interfaces

A.2.1 *Replicable* Interface

Interface Diagram:

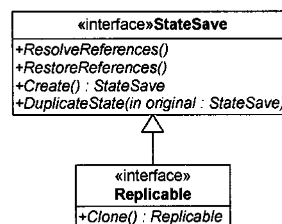


Figure A3. *Replicable* Interface

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Methods:

- **Clone**

  Objective: This method creates a clone of the entity. This method should be defined for each class or interface that inherits the Replicable interface.
  Parameters: None.
  Return Value:
  Type: Replicable.
  Function: Clone of the object.
  Exceptions: None.

A.2.2 *ProgrammaticEvent* Interface

Interface Diagram:

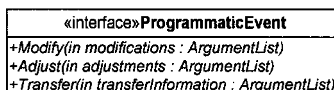


Figure A4. *ProgrammaticEvent* Interface

Methods:

- **Modify**

  Objective: This method modifies the attributes of an entity. This method should be defined for each class or interface that inherits the ProgrammaticEvent interface and desires the ability to programmatically modify its attributes.
  Parameters:
  modifications
  Mode: Input.
  Type: ArgumentList.
  Presence: Required.
  Function: Specifies the information required for modifying properties of an entity.
  Return Value: None.
  Exceptions: None.

- **Adjust**

  Objective: This method adjusts the attributes of an entity. This method should be defined for each class or interface that inherits the ProgrammaticEvent interface and desires the ability to programmatically adjust its attributes.
  Parameters:
  adjustments
  Mode: Input.
  Type: ArgumentList.
  Presence: Required.
  Function: Specifies the information required for adjusting properties of an entity.
  Return Value: None.
  Exceptions: None.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- **Transfer**

Objective: This method transfers properties of an entity to another entity. This method should be defined for each class or interface that inherits the ProgrammaticEvent interface and desires the ability to programmatically transfer its properties.

Parameters:

transferInformation

Mode: Input.
Type: ArgumentList.
Presence: Required.
Function: Specifies the information required for transferring properties an entity.

Return Value: None.
Exceptions: None.

A.2.3 *Routine* Interface

Interface Diagram:

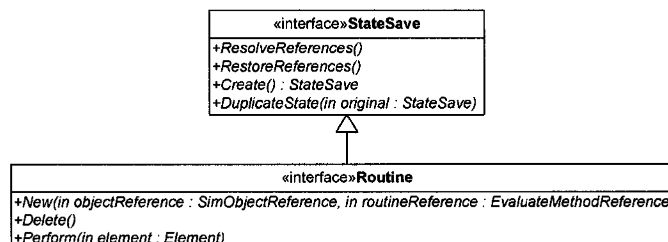


Figure A5. *Routine* Interface

Methods:

- **New**

Objective: This method initializes a Routine.

Parameters:

objectReference

Mode: Input.
Type: Reference to a SimObject.
Presence: Required.
Function: Specifies the reference to a SimObject on which the routine method should be executed.

routineReference

Mode: Input.
Type: EvaluateMethodReference.
Presence: Required.
Function: Specifies the routine method.

Return Value: None.
Exceptions: None.

- **Delete**

Objective: This method disposes a Routine.
Parameters: None.
Return Value: None.
Exceptions: None.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- **Perform**

  Objective: This method executes the specified routine method with an element as its input.

  Parameters:

  element

  Mode: Input.
  Type: Element.
  Presence: Required.
  Function: Specifies the element which is to be used as input for the routine method.

  Return Value: None.
  Exceptions: None.

A.2.4 *Condition* Interface

Interface Diagram:

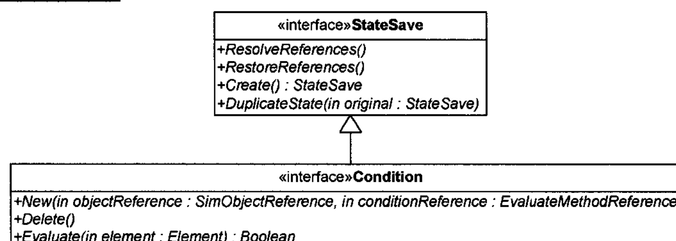


Figure A6. *Condition* Interface

Methods:

- **New**

  Objective: This method initializes a Condition.

  Parameters:

  objectReference

  Mode: Input.
  Type: Reference to a SimObject.
  Presence: Required.
  Function: Specifies the reference to a SimObject on which the routine method should be executed.

  conditionReference

  Mode: Input.
  Type: EvaluateMethodReference.
  Presence: Required.
  Function: Specifies the condition method.

  Return Value: None.
  Exceptions: None.

- **Delete**

  Objective: This method disposes a Condition.
  Parameters: None.
  Return Value: None.
  Exceptions: None.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- **Evaluate**

  Objective: This method executes the specified condition method with an element as its input.

  Parameters:

  element

  Mode: Input.

  Type: Element.

  Presence: Required.

  Function: Specifies the element which is to be used as input for the condition method.

  Return Value:

  Type: Boolean.

  Function: Result of the evaluated condition.

  Exceptions: None.

A.3 Application Interfaces

A.3.1 *SequentialSimulationExecutive* Interface

Interface Diagram:

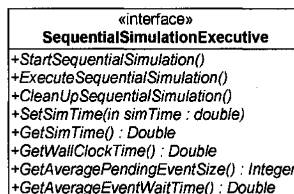


Figure A7. *SequentialSimulationExecutive* Interface

Methods:

- **StartSequentialSimulation**

  Objective: This method initializes DIESEL for a sequential simulation.

  Parameters: None.

  Return Value: None.

  Exceptions: None.

- **ExecuteSequentialSimulation**

  Objective: This method executes a sequential simulation.

  Parameters: None.

  Return Value: None.

  Exceptions: None.

- **CleanUpSequentialSimulation**

  Objective: This method cleans up DIESEL after executing a sequential simulation.

  Parameters: None.

  Return Value: None.

  Exceptions: None.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- **SetSimTime**

```
Objective:       This method sets the current simulation time for a
                 sequential simulation.
Parameters:
   time
      Mode:      Input.
      Type:      Double.
      Presence:  Required.
      Function:  Specifies the current simulation time.
Return Value:    None.
Exceptions:      None.
```

- **GetSimTime**

```
Objective:       This method returns the current simulation time
                 within a sequential simulation.
Parameters:      None.
Return Value:
      Type:      Double.
      Function:  Current simulation time.
Exceptions:      None.
```

- **GetWallClockTime**

```
Objective:       This method returns the wall clock time within a
                 sequential simulation.
Parameters:      None.
Return Value:
      Type:      Double.
      Function:  Current wall clock time.
Exceptions:      None.
```

- **GetAveragePendingEventSize**

```
Objective:       This method returns the average size of the pending
                 event set for all SimulationEngineComponents within
                 a sequential simulation.
Parameters:      None.
Return Value:
      Type:      Integer.
      Function:  Average size of pending event set.
Exceptions:      None.
```

- **GetAverageEventWaitTime**

```
Objective:       This method returns the average waiting time for an
                 event before it is executed within all
                 SimulationEngineComponents in a sequential
                 simulation.
Parameters:      None.
Return Value:
      Type:      Double.
      Function:  Average event wait time.
Exceptions:      None.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A.3.2 *ArgumentList* Interface

Interface Diagram:

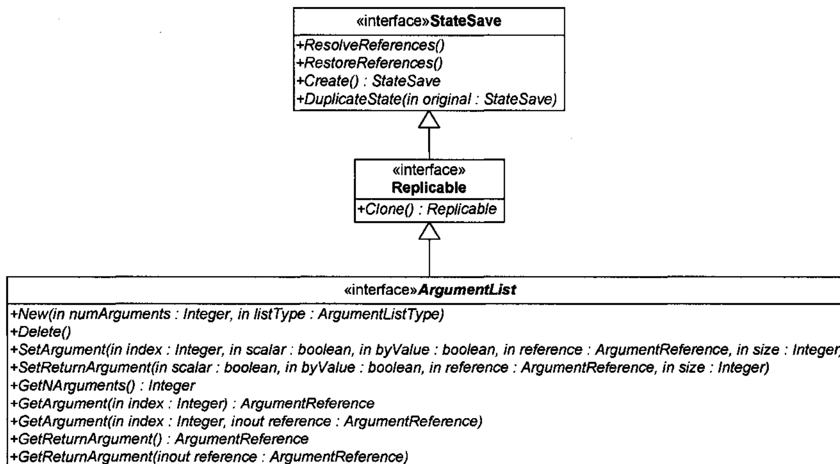


Figure A8. *ArgumentList* Interface

Methods:

- **ResolveReferences**

  Objective: This method resolves references for dynamically initialized attributes within an ArgumentList.
  Parameters: None.
  Return Value: None.
  Exceptions: None.

- **RestoreReferences**

  Objective: This method restores references for dynamically initialized attributes within an ArgumentList.
  Parameters: None.
  Return Value: None.
  Exceptions: None.

- **Create**

  Objective: This method creates an empty shell of an ArgumentList during the state duplication process of a non-terminating state save operation.
  Parameters: None.
  Return Value:
  Type: StateSave.
  Function: Empty shell of ArgumentList.
  Exceptions: None.

- **DuplicateState**

  Objective: This method copies the state of "original" to the current ArgumentList during the state duplication process of a non-terminating state save operation.
  Parameters:
  original
  Mode: Input.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
        Type:     StateSave.
        Presence: Required.
        Function: Specifies ArgumentList whose state is to be copied.
    Return Value: None.
    Exceptions:   None.
```

- **Clone**

```
    Objective:    This method creates a clone of an ArgumentList.
    Parameters:   None.
    Return Value:
        Type:     Replicable.
        Function: Clone of an ArgumentList.
    Exceptions:   None.
```

- **New**

```
    Objective:    This method initializes an ArgumentList.
    Parameters:
      numArguments
        Mode:     Input.
        Type:     Integer.
        Presence: Required.
        Function: Specifies the number of arguments in the
                  ArgumentList.
      listType
        Mode:     Input.
        Type:     ArgumentListType.
        Presence: Required.
        Function: Specifies the type of ArgumentList to be created.
    Return Value: None.
    Exceptions:
      "Invalid number of arguments"
        Cause:    "arguments" <= 0.
        Effect:   Simulation terminates.
```

- **Delete**

```
    Objective:    This method disposes an ArgumentList.
    Parameters:   None.
    Return Value: None.
    Exceptions:   None.
```

- **SetArgument**

```
    Objective:    This method adds an argument to an ArgumentList. An
                  index needs to be specified for an ARGUMENT_INDEXED
                  ArgumentList. If no size is specified, then the
                  reference to the argument is stored within an
                  ArgumentList. In this case, the argument cannot be
                  disposed within the application. If a size is
                  specified, then a copy of the argument is made and
                  stored within the ArgumentList, so that the original
                  argument can be disposed within the application.
    Parameters:
      index
        Mode:     Input.
        Type:     Integer.
        Presence: Optional.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
      Function:  Specifies the index of argument within the
                 ArgumentList.
   scalar
      Mode:      Input.
      Type:      Boolean.
      Presence:  Required.
      Function:  TRUE if argument is a scalar, else FALSE.
   byValue
      Mode:      Input.
      Type:      Boolean.
      Presence:  Required.
      Function:  TRUE if argument is passed by value, else FALSE.
   reference
      Mode:      Input.
      Type:      Reference to an argument.
      Presence:  Required.
      Function:  Specifies the reference to an argument.
   size
      Mode:      Input.
      Type:      Integer.
      Presence:  Required.
      Function:  Specifies the size of argument in bytes.
Return Value:    None.
Exceptions:
   "No index specified for an indexed list"
      Cause:     No index specified while adding an argument to an
                 INDEXED list.
      Effect:    Simulation terminates.
   "Index out of bounds"
      Cause:     (index < 0) OR (index > (nArguments - 1)) for an
                 INDEXED list.
      Effect:    Simulation terminates.
   "Attempt to index a non-indexed list"
      Cause:     An index is specified while adding an argument to a
                 NON-INDEXED list.
      Effect:    Simulation terminates.
```

- **SetReturnArgument**

```
Objective:       This method adds a return argument to an
                 ArgumentList. If no size is specified, then the
                 reference to the return argument is stored within
                 the ArgumentList. In this case, the return argument
                 cannot be disposed within the application. If a size
                 is specified, then a copy of the return argument is
                 made and stored within the ArgumentList, so that the
                 original return argument can be disposed within the
                 application.
Parameters:
   scalar
      Mode:      Input.
      Type:      Boolean.
      Presence:  Required.
      Function:  TRUE if return argument is a scalar, else FALSE.
   byValue
      Mode:      Input.
      Type:      Boolean.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
      Presence:  Required.
      Function:  TRUE if return argument is passed by value, else
                 FALSE.
   reference
      Mode:      Input.
      Type:      Reference to an argument.
      Presence:  Required.
      Function:  Specifies the reference to return argument.
   size
      Mode:      Input.
      Type:      Integer.
      Presence:  Required.
      Function:  Specifies the size of return argument in bytes.
Return Value:    None.
Exceptions:      None.
```

- **GetNArguments**

```
Objective:       This method is used to get the number of arguments
                 within an ArgumentList.
Parameters:      None.
Return Value:
      Type:      Integer.
      Function:  Number of arguments.
Exceptions:      None.
```

- **GetArgument**

```
Objective:       This method is used to return the reference to an
                 argument stored within an ArgumentList.
Parameters:
   index
      Mode:      Input.
      Type:      Integer.
      Presence:  Optional.
      Function:  Specifies the index of argument within the
                 ArgumentList.
Return Value:
      Type:      Reference to an argument.
      Function:  Reference to the argument.
Exceptions:
   "Accessing a non-existent argument"
      Cause:     Accessing an argument that was not stored.
      Effect:    Simulation terminates.
   "Accessing an argument by reference that was passed by value"
      Cause:     Accessing an argument by reference that was passed
                 by value.
      Effect:    Simulation terminates.
   "No index specified for an indexed list"
      Cause:     No index specified while retrieving an argument from
                 an INDEXED list.
      Effect:    Simulation terminates.
   "Index out of bounds"
      Cause:     (index < 0) OR (index > (nArguments - 1)) for an
                 INDEXED list.
      Effect:    Simulation terminates.
   "Attempt to index a non-indexed list"
      Cause:     An index is specified while retrieving an argument
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
                from a NON-INDEXED list.
     Effect:    Simulation terminates.
```

- **GetArgument**

```
Objective:      This method is used to return the value of an
                argument stored within an ArgumentList. The value of
                the argument is copied into the memory location
                pointed to by "reference".
Parameters:
   index
     Mode:      Input.
     Type:      Integer.
     Presence:  Optional.
     Function:  Specifies the index of argument within the
                ArgumentList.
   reference
     Mode:      Input/Output.
     Type:      Reference to an argument.
     Presence:  Required.
     Function:  Specifies the reference to an argument.
Return Value:   None.
Exceptions:
   "Accessing a non-existent argument"
     Cause:     Accessing an argument that was not stored.
     Effect:    Simulation terminates.
   "Accessing an argument by value that was passed by reference"
     Cause:     Accessing an argument by value that was passed by
                reference.
     Effect:    Simulation terminates.
   "No index specified for an indexed list"
     Cause:     No index specified while retrieving an argument from
                an INDEXED list.
     Effect:    Simulation terminates.
   "Index out of bounds"
     Cause:     (index < 0) OR (index > (nArguments - 1)) for an
                INDEXED list.
     Effect:    Simulation terminates.
   "Attempt to index a non-indexed list"
     Cause:     An index is specified while retrieving an argument
                from a NON-INDEXED list.
     Effect:    Simulation terminates.
```

- **GetReturnArgument**

```
Objective:      This method is used to return the reference to the
                return argument from an ArgumentList.
Parameters:     None.
Return Value:
     Type:      Reference to an argument.
     Function:  Reference to the return argument.
Exceptions:
   "Accessing a non-existent return argument"
     Cause:     Accessing an argument that was not stored.
     Effect:    Simulation terminates.
   "Accessing return argument by reference that was passed by value"
     Cause:     Accessing an argument by value that was passed by
                reference.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Effect: Simulation terminates.

- **GetReturnArgument**

Objective: This method is used to return the value of the return argument stored within an ArgumentList. The value of the return argument is copied into the memory location pointed to by "reference".

Parameters:

reference

Mode: Input/Output.
Type: Reference to an argument.
Presence: Required.
Function: Specifies the reference to return argument.

Return Value: None.

Exceptions:

"Accessing a non-existent return argument"

Cause: Accessing an argument that was not stored.
Effect: Simulation terminates.

"Accessing return argument by value that was passed by reference"

Cause: Accessing an argument by value that was passed by reference.
Effect: Simulation terminates.

A.3.3 *Delegate* Interface

Interface Diagram:

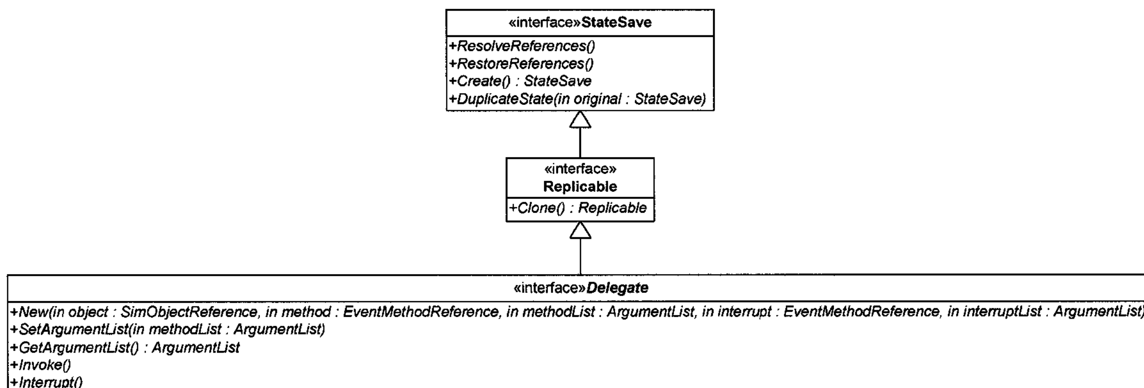


Figure A9. *Delegate* Interface

Methods:

- **ResolveReferences**

Objective: This method resolves references for dynamically initialized attributes within a Delegate.

Parameters: None.
Return Value: None.
Exceptions: None.

- **RestoreReferences**

Objective: This method restores references for dynamically initialized attributes within a Delegate.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
Parameters:      None.
Return Value:    None.
Exceptions:      None.
```

- **Create**

```
Objective:       This method creates an empty shell of a Delegate
                 during the state duplication process of a non-
                 terminating state save operation.
Parameters:      None.
Return Value:
      Type:      StateSave.
      Function:  Empty shell of Delegate.
Exceptions:      None.
```

- **DuplicateState**

```
Objective:       This method copies the state of "original" to the
                 current Delegate during the state duplication
                 process of a non-terminating state save operation.
Parameters:
   original
      Mode:      Input.
      Type:      StateSave.
      Presence:  Required.
      Function:  Specifies Delegate whose state is to be copied.
Return Value:    None.
Exceptions:      None.
```

- **Clone**

```
Objective:       This method creates a clone of a Delegate.
Parameters:      None.
Return Value:
      Type:      Replicable.
      Function:  Clone of a Delegate.
Exceptions:      None.
```

- **New**

```
Objective:       This method initializes a Delegate.
Parameters:
   object
      Mode:      Input.
      Type:      Reference to a SimObject.
      Presence:  Required.
      Function:  Specifies the reference to a SimObject on which the
                 event method should be executed.
   method
      Mode:      Input.
      Type:      EventMethodReference.
      Presence:  Required.
      Function:  Specifies the reference to an event method.
   methodList
      Mode:      Input.
      Type:      ArgumentList.
      Presence:  Required.
      Function:  Specifies the list of parameters to be used when the
                 event method is executed.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

interrupt
Mode: Input.
Type: EventMethodReference.
Presence: Optional.
Function: Specifies the reference to an interrupt method to be executed if the Delegate is interrupted.

interruptList
Mode: Input.
Type: ArgumentList.
Presence: Optional.
Function: Specifies the list of parameters to be used when the interrupt method is executed.

Return Value: None.
Exceptions: None.

- **SetArgumentList**

Objective: This method assigns an ArgumentList to a Delegate for invoking an event method.
Parameters:

methodList
Mode: Input.
Type: ArgumentList.
Presence: Required.
Function: Specifies the list of parameters to be used when the event method is executed.

Return Value: None.
Exceptions: None.

- **GetArgumentList**

Objective: This method returns the ArgumentList to be used for event method invocation from a Delegate.
Parameters: None.
Return Value:
Type: ArgumentList.
Function: Specifies the list of parameters to be used when the event method is executed.
Exceptions: None.

- **Invoke**

Objective: This method executes the event method within a Delegate with the specified ArgumentList.
Parameters: None.
Return Value: None.
Exceptions: None.

- **Interrupt**

Objective: This method cancels the event method invocation within a Delegate. The interrupt method is executed with the specified interrupt ArgumentList.
Parameters: None.
Return Value: None.
Exceptions: None.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A.3.4 *SimulationEngineComponent* Interface

Interface Diagram:

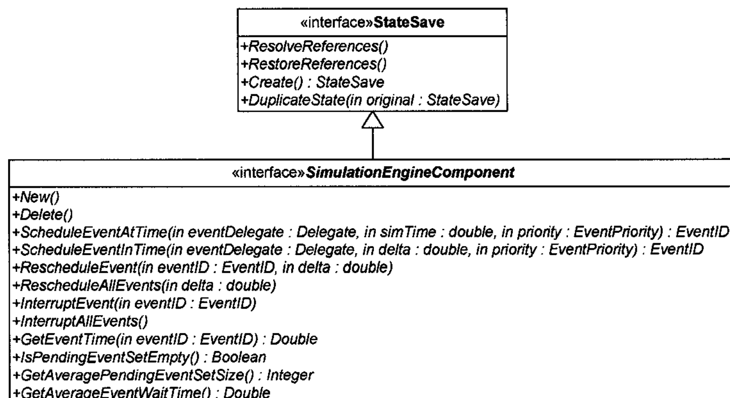


Figure A10. *SimulationEngineComponent* Interface

Methods:

- **ResolveReferences**
  Objective: This method resolves references for dynamically initialized attributes within a SimulationEngineComponent.
  Parameters: None.
  Return Value: None.
  Exceptions: None.

- **RestoreReferences**
  Objective: This method restores references for dynamically initialized attributes within a SimulationEngineComponent.
  Parameters: None.
  Return Value: None.
  Exceptions: None.

- **Create**
  Objective: This method creates an empty shell of a SimulationEngineComponent during the state duplication process of a non-terminating state save operation.
  Parameters: None.
  Return Value:
  Type: StateSave.
  Function: Empty shell of SimulationEngineComponent.
  Exceptions: None.

- **DuplicateState**
  Objective: This method copies the state of "original" to the current SimulationEngineComponent during the state duplication process of a non-terminating state save operation.
  Parameters:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

original

Mode: Input.
Type: StateSave.
Presence: Required.
Function: Specifies SimulationEngineComponent whose state is to be copied.

Return Value: None.
Exceptions: None.

- **New**

Objective: This method initializes a SimulationEngineComponent and registers itself with the SimulationCluster.
Parameters: None.
Return Value: None.
Exceptions: None.

- **Delete**

Objective: This method disposes a SimulationEngineComponent and de-registers itself the SimulationCluster.
Parameters: None.
Return Value: None.
Exceptions: None.

- **ScheduleEventAtTime**

Objective: This method schedules a Delegate to be executed on a SimObject associated with the SimulationEngineComponent at a specific simulation time. A priority can be specified for a Delegate to lower the priority of the Delegate relative to Delegates scheduled at the same simulation time. The Delegate is added to the collection of Delegates scheduled on the SimulationEngineComponent.

Parameters:

eventDelegate

Mode: Input.
Type: Delegate.
Presence: Required.
Function: Specifies the Delegate having reference to the event method.

simTime

Mode: Input.
Type: Double.
Presence: Required.
Function: Specifies the scheduled execution time of the Delegate.

priority

Mode: Event Priority.
Type: Double.
Presence: Optional.
Function: Specifies the priority of the Delegate relative to other delegates scheduled at the same simulation time.

Return Value:

Type: EventID.
Function: Unique ID of the Delegate added to the SimulationEngineComponent. This ID is used to refer

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

to the event for activities such as interrupting the Delegate or changing its scheduled execution time.

Exceptions:

"Execution time for delegate less than current simulation time"

Cause: (simTime < current simulation time).

Effect: Simulation terminates.

"Invalid priority for delegate"

Cause: (priority <= 0).

Effect: Simulation terminates.

- **ScheduleEventInTime**

Objective: This method schedules a Delegate to be executed on a SimObject associated with the SimulationEngineComponent, after a given simulation time has elapsed. This method is similar to the ScheduleEventAtTime method.

Parameters:

eventDelegate

Mode: Input.

Type: Delegate.

Presence: Required.

Function: Specifies the Delegate having reference to the event method.

delta

Mode: Input.

Type: Double.

Presence: Required.

Function: Specifies the offset of Delegate execution from current simulation time.

priority

Mode: Event Priority.

Type: Double.

Presence: Optional.

Function: Specifies the priority of the Delegate relative to other delegates scheduled at the same simulation time.

Return Value:

Type: EventID.

Function: Unique ID of the Delegate added to the SimulationEngineComponent. This ID is used to refer to the event for activities such as interrupting the Delegate or changing its scheduled execution time.

Exceptions:

"Execution time for delegate less than current simulation time"

Cause: (delta < 0).

Effect: Simulation terminates.

"Invalid priority for delegate"

Cause: (priority <= 0).

Effect: Simulation terminates.

- **RescheduleEvent**

Objective: This method reschedules a specific Delegate within a SimulationEngineComponent, by applying a certain simulation time as offset to its current execution time. The offset can be positive or negative. However, to reschedule a Delegate earlier than its

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

original execution time, the offset should be smaller than the difference between the current simulation time and original event execution time.

Parameters:

eventID

Mode: Input.
Type: EventID.
Presence: Required.
Function: Specifies the ID of the Delegate to be rescheduled.

delta

Mode: Input.
Type: Double.
Presence: Required.
Function: Specifies the offset of Delegate execution time from its current execution time.

Return Value: None.

Exceptions:

"Delegate to be rescheduled not found"

Cause: Specified delegate does not exist within the collection of delegates.
Effect: Simulation terminates.

"Attempt to reschedule delegate at time less than current simulation time"

Cause: New scheduled execution time for delegate less than current simulation time.
Effect: Simulation terminates.

- **RescheduleAllEvents**

Objective: This method reschedules all Delegates within a SimulationEngineComponent by applying a certain simulation time as offset to their current execution time. The restrictions on the offset value are identical to that for the offset within the RescheduleEvent method.

Parameters:

delta

Mode: Input.
Type: Double.
Presence: Required.
Function: Specifies the offset of Delegate execution time from its current execution time.

Return Value: None.

Exceptions:

"Attempt to reschedule delegates at time less than current simulation time"

Cause: New scheduled execution time for at least one Delegate less than current simulation time.
Effect: Simulation terminates.

- **InterruptEvent**

Objective: This method cancels the execution of a specific Delegate within a SimulationEngineComponent.

Parameters:

eventID

Mode: Input.
Type: EventID.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
      Presence:  Required.
      Function:  Specifies the ID of the Delegate to be interrupted.
Return Value:    None.
Exceptions:
   "Delegate to be interrupted not found"
      Cause:     Specified Delegate does not exist within the
                 collection of Delegates.
      Effect:    Simulation terminates.
```

- **InterruptAllEvents**

```
Objective:       This method cancels the execution of all Delegates
                 within a SimulationEngineComponent.
Parameters:      None.
Return Value:    None.
Exceptions:      None.
```

- **GetEventTime**

```
Objective:       This method returns the scheduled execution time of
                 a specific Delegate within a
                 SimulationEngineComponent.
Parameters:
   eventID
      Mode:      Input.
      Type:      EventID.
      Presence:  Required.
      Function:  Specifies the ID of the Delegate for which the
                 execution time is required.
Return Value:
      Type:      Double.
      Function:  Scheduled execution time of the Delegate.
Exceptions:
   "Delegate not found"
      Cause:     Specified Delegate does not exist within the
                 collection of events.
      Effect:    Simulation terminates.
```

- **IsPendingEventSetEmpty**

```
Objective:       This method ascertains whether a
                 SimulationEngineComponent has any pending Delegates
                 to be executed.
Parameters:      None.
Return Value:
      Type:      Boolean.
      Function:  TRUE if collection of Delegates is empty, else
                 FALSE.
Exceptions:      None.
```

- **GetAveragePendingEventSetSize**

```
Objective:       This method returns the average size of the pending
                 event set within the SimulationEngineComponent.
Parameters:      None.
Return Value:
      Type:      Integer.
      Function:  Average size of pending event set for the duration
                 of the simulation.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Exceptions: None.

- **GetAverageEventWaitTime**

Objective: This method returns the average wait time of an event to be executed within the SimulationEngineComponent.
Parameters: None.
Return Value:
Type: Double.
Function: Average wait time of an event to be executed.
Exceptions: None.

A.3.5 *DelayedCommit* Interface

Interface Diagram:

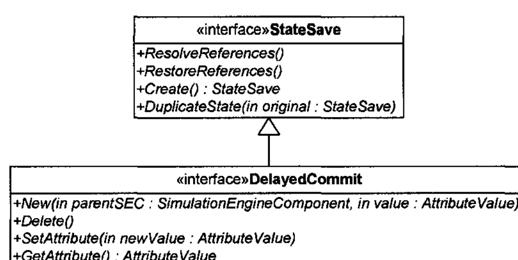


Figure A11. *DelayedCommit* Interface

Methods:

- **ResolveReferences**

Objective: This method resolves references for dynamically initialized attributes within a DelayedCommit.
Parameters: None.
Return Value: None.
Exceptions: None.

- **RestoreReferences**

Objective: This method restores references for dynamically initialized attributes within a DelayedCommit.
Parameters: None.
Return Value: None.
Exceptions: None.

- **Create**

Objective: This method creates an empty shell of a DelayedCommit during the state duplication process of a non-terminating state save operation.
Parameters: None.
Return Value:
Type: StateSave.
Function: Empty shell of DelayedCommit.
Exceptions: None.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- **DuplicateState**

Objective: This method copies the state of "original" to the current DelayedCommit during the state duplication process of a non-terminating state save operation.
Parameters:
  original
    Mode: Input.
    Type: StateSave.
    Presence: Required.
    Function: Specifies DelayedCommit whose state is to be copied.
Return Value: None.
Exceptions: None.

- **New**

Objective: This method initializes a DelayedCommit.
Parameters:
  parentSEC
    Mode: Input.
    Type: SimulationEngineComponent.
    Presence: Required.
    Function: Specifies the associated SimulationEngineComponent.
  value
    Mode: Input.
    Type: Attribute Value.
    Presence: Required.
    Function: Specifies the initial value of the DelayedCommit attribute.
Return Value: None.
Exceptions: None.

- **Delete**

Objective: This method disposes a DelayedCommit.
Parameters: None.
Return Value: None.
Exceptions: None.

- **SetAttribute**

Objective: This method assigns a new value to a DelayedCommit attribute. The value of the attribute is not updated immediately.
Parameters:
  newValue
    Mode: Input.
    Type: Attribute Value.
    Presence: Required.
    Function: Specifies the new value of the DelayedCommit attribute.
Return Value: None.
Exceptions: None.

- **GetAttribute**

Objective: This method returns the value of a DelayedCommit attribute.
Parameters: None.
Return Value:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
     Type:      Attribute Value.
     Function:  The value of the DelayedCommit attribute.
Exceptions:     None.
```

A.4 Synchronization Interfaces

A.4.1 *Trigger* Interface

Interface Diagram:

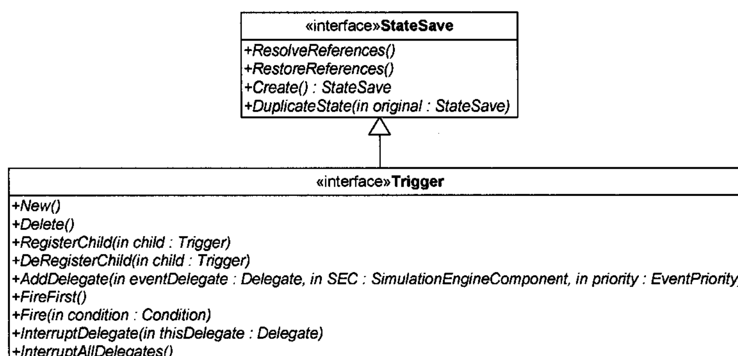


Figure A12. *Trigger* Interface

Methods:

- **ResolveReferences**
  Objective: This method resolves references for dynamically initialized attributes within a Trigger.
  Parameters: None.
  Return Value: None.
  Exceptions: None.

- **RestoreReferences**
  Objective: This method restores references for dynamically initialized attributes within a Trigger.
  Parameters: None.
  Return Value: None.
  Exceptions: None.

- **Create**
  Objective: This method creates an empty shell of a Trigger during the state duplication process of a non-terminating state save operation.
  Parameters: None.
  Return Value:
      Type: StateSave.
      Function: Empty shell of Trigger.
  Exceptions: None.

- **DuplicateState**
  Objective: This method copies the state of "original" to the current Trigger during the state duplication process

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
                    of a non-terminating state save operation.
Parameters:
   original
      Mode:       Input.
      Type:       StateSave.
      Presence:   Required.
      Function:   Specifies Trigger whose state is to be copied.
Return Value:     None.
Exceptions:       None.
```

- **New**

```
Objective:        This method initializes a Trigger.
Parameters:       None.
Return Value:     None.
Exceptions:       None.
```

- **Delete**

```
Objective:        This method disposes a Trigger.
Parameters:       None.
Return Value:     None.
Exceptions:       None.
```

- **RegisterChild**

```
Objective:        This method adds a child Trigger that is to be
                  triggered when the current Trigger is triggered.
Parameters:
   child
      Mode:       Input.
      Type:       Trigger.
      Presence:   Required.
      Function:   Specifies the child Trigger to be added.
Return Value:     None.
Exceptions:       None.
```

- **DeRegisterChild**

```
Objective:        This method removes a child Trigger from the current
                  Trigger.
Parameters:
   child
      Mode:       Input.
      Type:       Trigger.
      Presence:   Required.
      Function:   Specifies the child Trigger to be removed.
Return Value:     None.
Exceptions:       None.
```

- **AddDelegate**

```
Objective:        This method adds a Delegate to be scheduled for
                  execution, when the Trigger is triggered, to a
                  Trigger.
Parameters:
   eventDelegate
      Mode:       Input.
      Type:       Delegate.
      Presence:   Required.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Function: Specifies a Delegate to be executed when the Trigger is triggered.

SEC

Mode: Input.
Type: SimulationEngineComponent.
Presence: Required.
Function: Specifies the SimulationEngineComponent on which the Delegate is to be scheduled.

priority

Mode: Input.
Type: EventPriority.
Presence: Optional.
Function: Specifies a priority for the Delegate relative to other Delegates scheduled at the same simulation time.

Return Value: None.
Exceptions: None.

- **FireFirst**

Objective: This method schedules the immediate execution of the first Delegate scheduled (first-come-first-served) within a Trigger.
Parameters: None.
Return Value: None.
Exceptions: None.

- **Fire**

Objective: This method schedules the immediate execution of all delegates within a Trigger. This method also triggers all child Triggers to schedule the immediate execution of all their delegates, and to trigger any of their child Triggers. If a condition is specified, only delegates satisfying the condition are scheduled for execution within each Trigger.
Parameters:

condition

Mode: Input.
Type: Condition.
Presence: Optional.
Function: Specifies the condition to be satisfied.

Return Value: None.
Exceptions: None.

- **InterruptDelegate**

Objective: This method cancels the execution of a specific delegate within a Trigger.
Parameters:

thisDelegate

Mode: Input.
Type: Delegate.
Presence: Required.
Function: Specifies the Delegate to be interrupted.

Return Value: None.
Exceptions:
"Delegate to be interrupted not found"

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Cause: Specified delegate does not exist within the collection of delegates within the Trigger.
Effect: Simulation terminates.

- **InterruptAllDelegates**
  Objective: This method cancels the execution of all delegates within a Trigger.
  Parameters: None.
  Return Value: None.
  Exceptions: None.

A.4.2 *TriggerCounter* Interface

Interface Diagram:

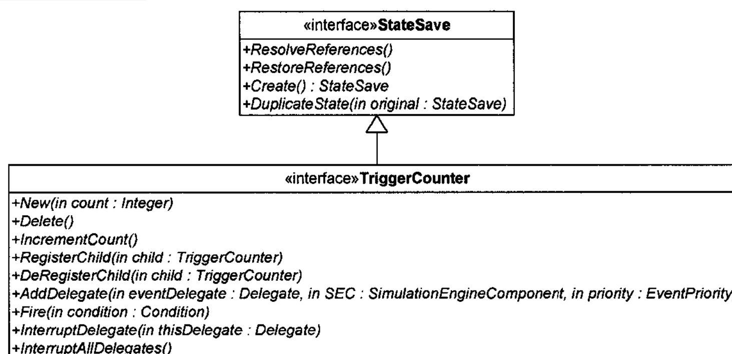


Figure A13. *TriggerCounter* Interface

Methods:

- **ResolveReferences**
  Objective: This method resolves references for dynamically initialized attributes within a TriggerCounter.
  Parameters: None.
  Return Value: None.
  Exceptions: None.

- **RestoreReferences**
  Objective: This method restores references for dynamically initialized attributes within a TriggerCounter.
  Parameters: None.
  Return Value: None.
  Exceptions: None.

- **Create**
  Objective: This method creates an empty shell of a TriggerCounter during the state duplication process of a non-terminating state save operation.
  Parameters: None.
  Return Value:
    Type: StateSave.
    Function: Empty shell of TriggerCounter.
  Exceptions: None.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- **DuplicateState**

```
Objective:      This method copies the state of "original" to the
                current TriggerCounter during the state duplication
                process of a non-terminating state save operation.
Parameters:
   original
      Mode:     Input.
      Type:     StateSave.
      Presence: Required.
      Function: Specifies TriggerCounter whose state is to be
                copied.
Return Value:   None.
Exceptions:     None.
```

- **New**

```
Objective:      This method initializes a TriggerCounter.
Parameters:
   count
      Mode:     Input.
      Type:     Integer.
      Presence: Optional.
      Function: Specifies the number of delegates to be triggered.
Return Value:   None.
Exceptions:     None.
```

- **Destroy**

```
Objective:      This method disposes a TriggerCounter.
Parameters:     None.
Return Value:   None.
Exceptions:     None.
```

- **IncrementCount**

```
Objective:      This method increments the count of Delegates to be
                triggered within the TriggerCounter.
Parameters:     None.
Return Value:   None.
Exceptions:     None.
```

- **RegisterChild**

```
Objective:      This method adds a TriggerCounter that is to be
                triggered when the current TriggerCounter is
                triggered.
Parameters:
   child
      Mode:     Input.
      Type:     TriggerCounter.
      Presence: Required.
      Function: Specifies the child TriggerCounter to be added.
Return Value:   None.
Exceptions:     None.
```

- **DeRegisterChild**

```
Objective:      This method removes a child TriggerCounter from the
                current TriggerCounter.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Parameters:
child
Mode: Input.
Type: TriggerCounter.
Presence: Required.
Function: Specifies the child TriggerCounter to be removed.
Return Value: None.
Exceptions: None.

- **AddDelegate**

Objective: This method adds a Delegate to be scheduled for execution, when a TriggerCounter is triggered, to a TriggerCounter. If the TriggerCounter has been triggered, then the Delegate is immediately scheduled for execution.
Parameters:
eventDelegate
Mode: Input.
Type: Delegate.
Presence: Required.
Function: Specifies a Delegate to be executed when the TriggerCounter is triggered.
SEC
Mode: Input.
Type: SimulationEngineComponent.
Presence: Required.
Function: Specifies the SimulationEngineComponent on which the Delegate is to be scheduled.
priority
Mode: Input.
Type: EventPriority.
Presence: Optional.
Function: Specifies a priority for the Delegate relative to other Delegates scheduled at the same simulation time.
Return Value: None.
Exceptions: None.

- **Fire**

Objective: This method schedules the immediate execution of all delegates within a TriggerCounter. This method also triggers all child TriggerCounters to schedule the immediate execution of all their delegates, and to trigger any of their child TriggerCounters. If a condition is specified, only delegates satisfying the condition are scheduled for execution within each TriggerCounter.
Parameters:
condition
Mode: Input.
Type: Condition.
Presence: Optional.
Function: Specifies the condition to be satisfied.
Return Value: None.
Exceptions: None.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- **InterruptDelegate**

  Objective: This method cancels the execution of a specific delegate within a TriggerCounter.
  Parameters:
  thisDelegate
  Mode: Input.
  Type: Delegate.
  Presence: Required.
  Function: Specifies the Delegate to be interrupted.
  Return Value: None.
  Exceptions:
  "Delegate to be interrupted not found"
  Cause: Specified delegate does not exist within the collection of delegates within the TriggerCounter.
  Effect: Simulation terminates.

- **InterruptAllDelegates**

  Objective: This method cancels the execution of all delegates within a TriggerCounter.
  Parameters: None.
  Return Value: None.
  Exceptions: None.

A.4.3 *Join* Interface

Interface Diagram:

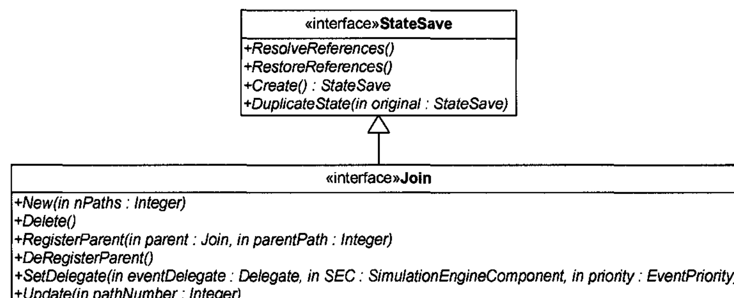


Figure A14. *Join* Interface

Methods:

- **ResolveReferences**

  Objective: This method resolves references for dynamically initialized attributes within a Join.
  Parameters: None.
  Return Value: None.
  Exceptions: None.

- **RestoreReferences**

  Objective: This method restores references for dynamically initialized attributes within a Join.
  Parameters: None.
  Return Value: None.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Exceptions: None.

- **Create**

Objective: This method creates an empty shell of a Join during the state duplication process of a non-terminating state save operation.
Parameters: None.
Return Value:
    Type: StateSave.
    Function: Empty shell of Join.
Exceptions: None.

- **DuplicateState**

Objective: This method copies the state of "original" to the current Join during the state duplication process of a non-terminating state save operation.
Parameters:
  original
    Mode: Input.
    Type: StateSave.
    Presence: Required.
    Function: Specifies Join whose state is to be copied.
Return Value: None.
Exceptions: None.

- **New**

Objective: This method initializes a Join with a specified number of paths, with each path specifying an event to wait for.
Parameters:
  nPaths
    Mode: Input.
    Type: Integer.
    Presence: Required.
    Function: Specifies the number of input paths.
Return Value: None.
Exceptions:
  "Invalid number of paths"
    Cause: (nPaths < 1)
    Effect: Simulation terminates.

- **Delete**

Objective: This method disposes a Join.
Parameters: None.
Return Value: None.
Exceptions: None.

- **RegisterParent**

Objective: This method sets the parent Join for the current Join, and indicates the specific path on which the current Join exists on the parent Join.
Parameters:
  parent
    Mode: Input.
    Type: Join.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Presence: Required.
Function: Specifies the parent Join.

parentPath

Mode: Input.
Type: Integer.
Presence: Required.
Function: Specifies the path on which this Join exists on "parent".

Return Value: None.
Exceptions: None.

- **DeRegisterParent**

Objective: This method removes the parent Join from the current Join.
Parameters: None.
Return Value: None.
Exceptions: None.

- **SetDelegate**

Objective: This method sets the Delegate to be executed after the Join has been satisfied. If all paths have been satisfied at least once,

- The Delegate is scheduled for immediate execution
- If a parent Join exists, it is updated on the specified path.
- The number of events satisfied on each path is decremented by 1.

Parameters:

eventDelegate

Mode: Input.
Type: Delegate.
Presence: Required.
Function: Specifies a Delegate to be executed when the Join is triggered.

SEC

Mode: Input.
Type: SimulationEngineComponent.
Presence: Required.
Function: Specifies the SimulationEngineComponent on which the Delegate is to be scheduled.

priority

Mode: Input.
Type: EventPriority.
Presence: Optional.
Function: Specifies a priority for the Delegate relative to other Delegates scheduled at the same simulation time.

Return Value: None.
Exceptions: None.

- **Update**

Objective: This method indicates that an event has occurred on a specific input path. If all paths have been satisfied at least once, and a Delegate has been specified,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- The Delegate is scheduled for immediate execution
- If a parent Join exists, it is updated on the specified path.
- The number of events satisfied on each path is decremented by 1.

Parameters:
pathNumber
Mode: Input.
Type: Integer.
Presence: Required.
Function: Specifies the number indicating the path that just received an input (numbered starting from 1).
Return Value: None.
Exceptions:
"Path number out of bounds"
Cause: (pathNumber < 0) OR (pathNumber > (nPaths - 1)).
Effect: Simulation terminates.

A.4.4 *JoinCounter* Interface

Interface Diagram:

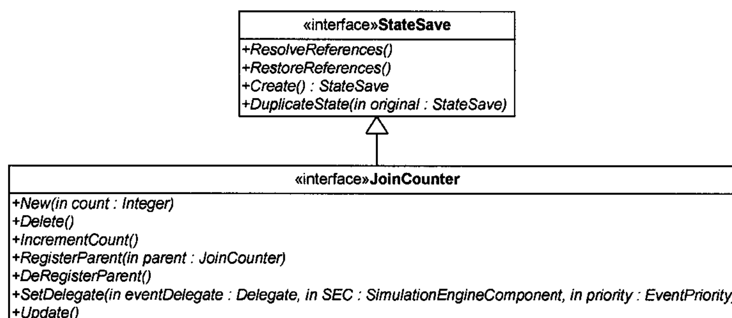


Figure A15. *JoinCounter* Interface

Methods:

- **ResolveReferences**
  Objective: This method resolves references for dynamically initialized attributes within a JoinCounter.
  Parameters: None.
  Return Value: None.
  Exceptions: None.

- **RestoreReferences**
  Objective: This method restores references for dynamically initialized attributes within a JoinCounter.
  Parameters: None.
  Return Value: None.
  Exceptions: None.

- **Create**
  Objective: This method creates an empty shell of a JoinCounter during the state duplication process of a non-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
                  terminating state save operation.
Parameters:       None.
Return Value:
      Type:       StateSave.
      Function:   Empty shell of JoinCounter.
Exceptions:       None.
```

- **DuplicateState**

```
Objective:        This method copies the state of "original" to the
                  current JoinCounter during the state duplication
                  process of a non-terminating state save operation.
Parameters:
   original
      Mode:       Input.
      Type:       StateSave.
      Presence:   Required.
      Function:   Specifies JoinCounter whose state is to be copied.
Return Value:     None.
Exceptions:       None.
```

- **New**

```
Objective:        This method initializes a JoinCounter with a
                  specified number of events to wait for.
Parameters:
   count
      Mode:       Input.
      Type:       Integer.
      Presence:   Optional.
      Function:   Specifies the number of conditions to be satisfied.
Return Value:     None.
Exceptions:       None.
```

- **Delete**

```
Objective:        This method disposes a JoinCounter.
Parameters:       None.
Return Value:     None.
Exceptions:       None.
```

- **IncrementCount**

```
Objective:        This method increments the number of events to
                  occur, before the Delegate within a JoinCounter can
                  be executed.
Parameters:       None.
Return Value:     None.
Exceptions:       None.
```

- **RegisterParent**

```
Objective:        This method sets the parent JoinCounter for the
                  current JoinCounter
Parameters:
   parent
      Mode:       Input.
      Type:       JoinCounter.
      Presence:   Required.
      Function:   Specifies the parent JoinCounter.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Return Value: None.
Exceptions: None.

- **DeRegisterParent**

Objective: This method removes the parent JoinCounter from the current JoinCounter.
Parameters: None.
Return Value: None.
Exceptions: None.

- **SetDelegate**

Objective: This method sets the Delegate to be executed after the JoinCounter has been satisfied. If the specified number of events have occurred,
- The Delegate is scheduled for immediate execution
- If a parent JoinCounter has been registered, it is updated.
- The JoinCounter is reset.

Parameters:

eventDelegate

Mode: Input.
Type: Delegate.
Presence: Required.
Function: Specifies a Delegate to be executed when the JoinCounter is satisfied.

SEC

Mode: Input.
Type: SimulationEngineComponent.
Presence: Required.
Function: Specifies the SimulationEngineComponent on which the Delegate is to be scheduled.

priority

Mode: Input.
Type: EventPriority.
Presence: Optional.
Function: Specifies a priority for the Delegate relative to other Delegates scheduled at the same simulation time.

Return Value: None.
Exceptions: None.

- **Update**

Objective: This method indicates that an event has occurred. If the specified number of events have occurred, and a Delegate has been specified,
- The Delegate is scheduled for immediate execution
- If a parent JoinCounter has been registered, it is updated.
- The JoinCounter is reset.

Parameters: None.
Return Value: None.
Exceptions: None.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A.5 Random Number Generation Interfaces

A.5.1 *RNG* Interface

Interface Diagram:

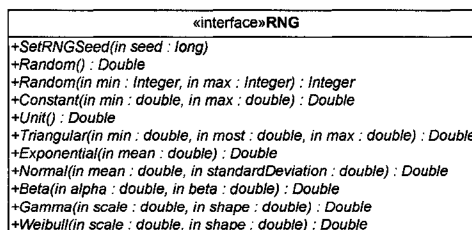


Figure A16. *RNG* Interface

Methods:

- **SetRNGSeed**

  Objective: This method sets the seed for the random number generator.

  Parameters:

  seed

  Mode: Input.
  Type: Long.
  Presence: Required.
  Function: Specifies the seed to be used to generate random numbers.

  Return Value: None.
  Exceptions: None.

- **Random**

  Objective: This method returns a random floating-point number from the random number generator.

  Parameters: None.

  Return Value:

  Type: Double.
  Function: Random floating-point number.

  Exceptions: None.

- **Random**

  Objective: This method returns a random integer between "min" and "max" from the random number generator.

  Parameters:

  min

  Mode: Input.
  Type: Integer.
  Presence: Required.
  Function: Specifies the minimum value of the random integer.

  max

  Mode: Input.
  Type: Integer.
  Presence: Required.
  Function: Specifies the maximum value of the random integer.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
Return Value:
      Type:      Integer.
      Function:  Random integer.
Exceptions:      None.
```

- **Constant**

```
Objective:       This method returns a random floating-point number
                 from a Constant distribution between "min" and
                 "max".
Parameters:
   min
      Mode:      Input.
      Type:      Double.
      Presence:  Required.
      Function:  Specifies the minimum value of the random floating-
                 point number.
   max
      Mode:      Input.
      Type:      Double.
      Presence:  Required.
      Function:  Specifies the maximum value of the random floating-
                 point number.
Return Value:
      Type:      Double.
      Function:  Random floating-point number from a Constant
                 distribution.
Exceptions:      None.
```

- **Unit**

```
Objective:       This method returns a random floating-point number
                 from a Unit distribution.
Parameters:      None.
Return Value:
      Type:      Double.
      Function:  Random floating-point number from a Unit
                 distribution.
Exceptions:      None.
```

- **Triangular**

```
Objective:       This method returns a random floating-point number
                 from a Triangular distribution.
Parameters:
   min
      Mode:      Input.
      Type:      Double.
      Presence:  Required.
      Function:  Specifies the minimum value for the Triangular
                 distribution.
   most
      Mode:      Input.
      Type:      Double.
      Presence:  Required.
      Function:  Specifies the most likely value for the Triangular
                 distribution.
   max
      Mode:      Input.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
        Type:      Double.
        Presence:  Required.
        Function:  Specifies the maximum value for the Triangular
                   distribution.
  Return Value:
        Type:      Double.
        Function:  Random floating-point number from a Triangular
                   distribution.
  Exceptions:      None.
```

- **Exponential**

```
  Objective:       This method returns a random floating-point number
                   from an Exponential distribution.
  Parameters:
     mean
        Mode:      Input.
        Type:      Double.
        Presence:  Required.
        Function:  Specifies the mean value for the Exponential
                   distribution (mean > 0).
  Return Value:
        Type:      Double.
        Function:  Random floating-point number from an Exponential
                   distribution.
  Exceptions:      None.
```

- **Normal**

```
  Objective:       This method returns a random floating-point number
                   from a Normal distribution.
  Parameters:
     mean
        Mode:      Input.
        Type:      Double.
        Presence:  Required.
        Function:  Specifies the mean value for the Normal
                   distribution.
     standardDeviation
        Mode:      Input.
        Type:      Double.
        Presence:  Required.
        Function:  Specifies the standard deviation for the Normal
                   distribution.
  Return Value:
        Type:      Double.
        Function:  Random floating-point number from a Normal
                   distribution.
  Exceptions:      None.
```

- **Beta**

```
  Objective:       This method returns a random floating-point number
                   from a Beta distribution.
  Parameters:
     alpha
        Mode:      Input.
        Type:      Double.
        Presence:  Required.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Function: Specifies the alpha value for the Beta distribution (alpha > 0).

beta

Mode: Input.
Type: Double.
Presence: Required.
Function: Specifies the beta value for the Beta distribution (beta > 0).

Return Value:

Type: Double.
Function: Random floating-point number from a Beta distribution.

Exceptions: None.

- **Gamma**

Objective: This method returns a random floating-point number from a Gamma distribution.

Parameters:

scale

Mode: Input.
Type: Double.
Presence: Required.
Function: Specifies the scale for the Gamma distribution (scale > 0).

shape

Mode: Input.
Type: Double.
Presence: Required.
Function: Specifies the shape for the Gamma distribution (shape > 0).

Return Value:

Type: Double.
Function: Random floating-point number from a Gamma distribution.

Exceptions: None.

- **Weibull**

Objective: This method returns a random floating-point number from a Weibull distribution.

Parameters:

scale

Mode: Input.
Type: Double.
Presence: Required.
Function: Specifies the scale for the Weibull distribution (scale > 0).

shape

Mode: Input.
Type: Double.
Presence: Required.
Function: Specifies the shape for the Weibull distribution (shape > 0).

Return Value:

Type: Double.
Function: Random floating-point number from a Weibull distribution.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Exceptions: None.

A.5.2 *AdvancedDistributions* Interface

Interface Diagram:

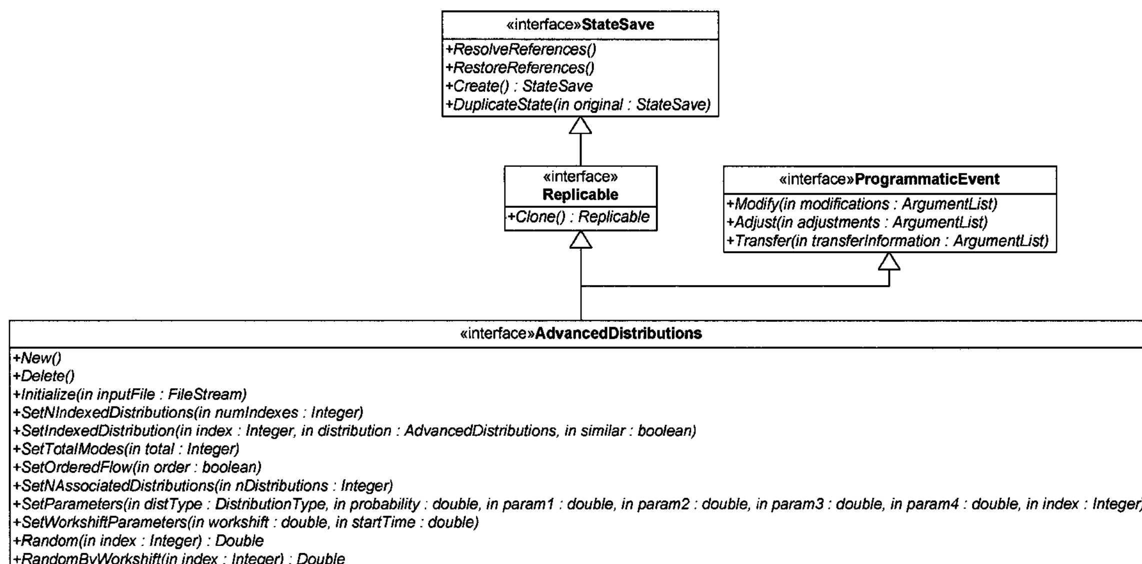


Figure A17. *AdvancedDistributions* Interface

Methods:

- **ResolveReferences**

  Objective: This method resolves references for dynamically initialized attributes within an AdvancedDistributions.
  Parameters: None.
  Return Value: None.
  Exceptions: None.

- **RestoreReferences**

  Objective: This method restores references for dynamically initialized attributes within an AdvancedDistributions.
  Parameters: None.
  Return Value: None.
  Exceptions: None.

- **Create**

  Objective: This method creates an empty shell of an AdvancedDistributions during the state duplication process of a non-terminating state save operation.
  Parameters: None.
  Return Value:
  Type: StateSave.
  Function: Empty shell of AdvancedDistributions.
  Exceptions: None.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- **DuplicateState**

Objective: This method copies the state of "original" to the current AdvancedDistributions during the state duplication process of a non-terminating state save operation.

Parameters:

original

Mode: Input.
Type: StateSave.
Presence: Required.
Function: Specifies AdvancedDistributions whose state is to be copied.

Return Value: None.
Exceptions: None.

- **Clone**

Objective: This method creates a clone of an AdvancedDistributions.
Parameters: None.
Return Value:

Type: Replicable.
Function: Clone of an AdvancedDistributions.

Exceptions: None.

- **Modify**

Objective: This method modifies an AdvancedDistributions by providing a new set of parameters. If any dependencies exist from the current AdvancedDistributions, it is transferred to the new parameters.

Parameters:

modifications

Mode: Input.
Type: ArgumentList.
Presence: Required.
Function: Specifies the new parameters.

Return Value: None.
Exceptions: None.

- **Adjust**

Objective: This method is not defined for an AdvancedDistributions.

Parameters:

adjustments

Mode: Input.
Type: ArgumentList.
Presence: Required.
Function: Specifies the new parameters.

Return Value: None.
Exceptions:

"Adjust method not defined for this interface"

Cause: Method not defined for an AdvancedDistributions.
Effect: Simulation terminates.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- **Transfer**

  Objective: This method is not defined for an AdvancedDistributions.

  Parameters:

  transferInformation

  Mode: Input.

  Type: ArgumentList.

  Presence: Required.

  Function: Specifies the new parameters.

  Return Value: None.

  Exceptions:

  "Transfer method not defined for this interface"

  Cause: Method not defined for an AdvancedDistributions.

  Effect: Simulation terminates.

- **New**

  Objective: This method initializes an AdvancedDistributions.

  Parameters: None.

  Return Value: None.

  Exceptions: None.

- **Delete**

  Objective: This method disposes an AdvancedDistributions.

  Parameters: None.

  Return Value: None.

  Exceptions: None.

- **Initialize**

  Objective: This method initializes an AdvancedDistributions from a file.

  Parameters: None.

  Return Value: None.

  Exceptions: None.

- **SetNIndexedDistributions**

  Objective: This method initializes the number of independent indexed AdvancedDistributions within the current AdvancedDistributions.

  Parameters:

  numIndexes

  Mode: Input.

  Type: Integer.

  Presence: Required.

  Function: Specifies the number of independent indexed distributions.

  Return Value: None.

  Exceptions: None.

- **SetIndexedDistribution**

  Objective: This method initializes the number of independent indexed AdvancedDistributions within the current AdvancedDistributions.

  Parameters:

  index

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Mode: Input.
Type: Integer.
Presence: Required.
Function: Specifies the index of the independent distribution.

distribution
Mode: Input.
Type: AdvancedDistributions.
Presence: Required.
Function: Specifies the independent indexed distribution.

similar
Mode: Input.
Type: Boolean.
Presence: Required.
Function: Specifies if the indexed distribution is the same as any other indexed distribution. TRUE if distribution is similar, else FALSE.

Return Value: None.
Exceptions:
"Attempt to index a non-indexed AdvancedDistributions"
Cause: AdvancedDistributions does not have indexed distributions.
Effect: Simulation terminates.
"Index out of bounds"
Cause: (index < 0) OR (index > (number of indexed types - 1)).
Effect: Simulation terminates.

- **SetTotalModes**

Objective: This method initializes the total number of modes for an AdvancedDistributions.
Parameters:
total
Mode: Input.
Type: Integer.
Presence: Required.
Function: Specifies the total number of modes.
Return Value: None.
Exceptions: None.

- **SetOrderedFlow**

Objective: This method sets an ordered flow of operation within an AdvancedDistributions.
Parameters:
order
Mode: Input.
Type: Boolean.
Presence: Required.
Function: Specifies if the distribution is ordered. TRUE if distribution is ordered, else FALSE.
Return Value: None.
Exceptions: None.

- **SetNAssociatedDistributions**

Objective: This method initializes the number of distributions associated with an AdvancedDistributions. More than 1 distribution may be associated with a single

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

independent distribution to support multimodal, blocked and ordered mode of operations.

Parameters:

nDistributions

Mode: Input.
Type: Integer.
Presence: Required.
Function: Specifies the number of distributions.

Return Value: None.
Exceptions: None.

- **SetParameters**

Objective: This method initializes the parameters for each associated distribution within an AdvancedDistributions.

Parameters:

distType

Mode: Input.
Type: Type of Distribution.
Presence: Required.
Function: Specifies the type of distribution (Constant, Exponential, Normal, Triangular, Beta, Gamma or Weibull).

probability

Mode: Input.
Type: Double.
Presence: Required.
Function: Specifies the probability value that is associated with each distribution. If the process is unimodal, then the probability value associated with that single distribution is always 1. If the process is multimodal, there is probability values associated with every modal distribution with the sum of probabilities of all the modes adding up to 1.

param1, param2, param3, param4

Mode: Input.
Type: Double.
Presence: Required.
Function: Specifies the parameters for the various distributions (specified in the table below).

Distribution	**param1**	**param2**	**param3**	**param4**
Constant	Value	N/A	N/A	N/A
Exponential	Mean	N/A	N/A	N/A
Normal	Mean	Standard Deviation	N/A	N/A
Triangular	Minimum	Maximum	Mode	N/A
Beta	Minimum	Maximum	Alpha	Beta
Gamma	Mean	N/A	Alpha	N/A
Weibull	Mean	N/A	N/A	Beta

index

Mode: Input.
Type: Integer.
Presence: Required.
Function: Specifies the index of the associated distribution.

Return Value: None.
Exceptions:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

"Invalid CONSTANT Distribution"
  Cause: (Value < 0).
  Effect: Simulation terminates.
"Invalid EXPONENTIAL Distribution"
  Cause: (Mean <= 0).
  Effect: Simulation terminates.
"Invalid NORMAL Distribution"
  Cause: (Standard Deviation < 0).
  Effect: Simulation terminates.
"Invalid TRIANGULAR Distribution"
  Cause: (Minimum > Maximum) OR (Minimum >= Mode) OR (Mode >= Maximum).
  Effect: Simulation terminates.
"Invalid BETA Distribution"
  Cause: (Alpha <= 0) OR (((Maximum - Minimum) × Beta) <= 0).
  Effect: Simulation terminates.
"Invalid GAMMA Distribution"
  Cause: (Mean <= 0) OR (Alpha <= 0).
  Effect: Simulation terminates.
"Invalid WEIBULL Distribution"
  Cause: (Mean <= 0) OR (Beta <= 0).
  Effect: Simulation terminates.

- **SetWorkshiftParameters**
  Objective: This method initializes work shift parameters within an AdvancedDistributions.
  Parameters:
    workShift
      Mode: Input.
      Type: Double.
      Presence: Required.
      Function: Specifies the work shift for a day within the simulation (0 < work shift ≤ 24).
    startTime
      Mode: Input.
      Type: Double.
      Presence: Required.
      Function: Specifies the work shift start time during a day within the simulation.
  Return Value: None.
  Exceptions: None.

- **Random**
  Objective: This method returns a random floating-point number from an AdvancedDistributions. If an index is specified, then the specific distribution is used to generate the random number.
  Parameters:
    index
      Mode: Input.
      Type: Integer.
      Presence: Optional.
      Function: Specifies the index of the independent distribution.
  Return Value:
      Type: Double.
      Function: Random floating-point number.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
Exceptions:
   "Attempt to index a non-indexed AdvancedDistributions"
      Cause:     AdvancedDistributions does not have indexed
                 distributions.
      Effect:    Simulation terminates.
   "Index out of bounds"
      Cause:     (index < 0) OR (index > (numIndexedTypes - 1)).
      Effect:    Simulation terminates.
```

- **RandomByWorkShift**

```
Objective:       This method returns a random floating-point number
                 based on work shift from an AdvancedDistributions.
                 If an index is specified, then the specific
                 distribution is used to generate the random number.
Parameters:
   index
      Mode:      Input.
      Type:      Integer.
      Presence:  Optional.
      Function:  Specifies the index of the independent distribution.
Return Value:
      Type:      Double.
      Function:  Random floating-point number.
Exceptions:
   "Attempt to index a non-indexed AdvancedDistributions"
      Cause:     AdvancedDistributions does not have indexed
                 distributions.
      Effect:    Simulation terminates.
   "Index out of bounds"
      Cause:     (index < 0) OR (index > (numIndexedTypes - 1)).
      Effect:    Simulation terminates.
   "Invalid workshift time"
      Cause:     (workShift > 24) OR (workShift not specified.
      Effect:    Simulation terminates.
```

A.6 Entity Management Interfaces

A.6.1 *Set* Interface

Interface Diagram:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

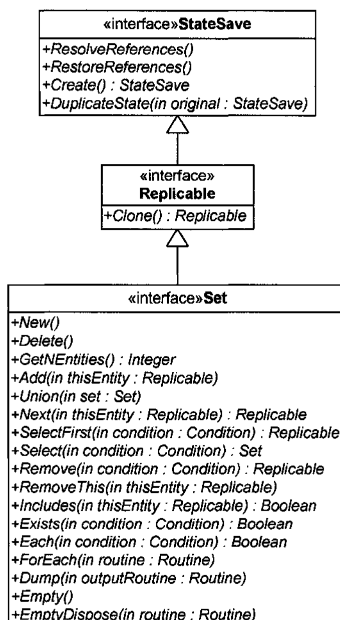


Figure A18. *Set* Interface

Methods:

- **ResolveReferences**

  Objective: This method resolves references for dynamically initialized attributes within a Set.
  Parameters: None.
  Return Value: None.
  Exceptions: None.

- **RestoreReferences**

  Objective: This method restores references for dynamically initialized attributes within a Set.
  Parameters: None.
  Return Value: None.
  Exceptions: None.

- **Create**

  Objective: This method creates an empty shell of a Set during the state duplication process of a non-terminating state save operation.
  Parameters: None.
  Return Value:
  Type: StateSave.
  Function: Empty shell of Set.
  Exceptions: None.

- **DuplicateState**

  Objective: This method copies the state of "original" to the current Set during the state duplication process of a non-terminating state save operation.
  Parameters:
  original
  Mode: Input.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
      Type:      StateSave.
      Presence:  Required.
      Function:  Specifies Set whose state is to be copied.
Return Value:    None.
Exceptions:      None.
```

- **Clone**

```
Objective:       This method creates a clone of a Set with all
                 entites currently within the Set existing in the
                 clone.
Parameters:      None.
Return Value:
      Type:      Replicable.
      Function:  Clone of a Set.
Exceptions:      None.
```

- **New**

```
Objective:       This method initializes a Set.
Parameters:      None.
Return Value:    None.
Exceptions:      None.
```

- **Delete**

```
Objective:       This method disposes a Set.
Parameters:      None.
Return Value:    None.
Exceptions:      None.
```

- **GetNEntities**

```
Objective:       This method returns the current number of entities
                 within a Set.
Parameters:      None.
Return Value:
      Type:      Integer.
      Function:  Current number of entities in the Set.
Exceptions:      None.
```

- **Add**

```
Objective:       This method adds an entity to a Set.
Parameters:
   thisEntity
      Mode:      Input.
      Type:      Entity.
      Presence:  Required.
      Function:  Specifies the entity to be added to the Set.
Return Value:    None.
Exceptions:      None.
```

- **Union**

```
Objective:       This method creates a Union Set of the current Set
                 and "set". Each element of "set" is added to the
                 current Set.
Parameters:
   set
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Mode: Input.
Type: Set.
Presence: Required.
Function: Specifies the Set to be used for Union operation.
Return Value: None.
Exceptions: None.

- **Next**

Objective: This method returns the next entity after "thisEntity" within a Set. If "thisEntity" is NULL, it returns the first entity within the Set. A NULL is returned when there is no more unseen entity in the Set. If this method is continuously called till a NULL value is returned, then a new entity is returned each time.
Parameters:
thisEntity
Mode: Input.
Type: Entity.
Presence: Required.
Function: Specifies the current entity.
Return Value:
Type: Entity.
Function: Next entity after "thisEntity".
Exceptions: None.

- **SelectFirst**

Objective: This method returns the first entity within a Set that satisfies a specified condition.
Parameters:
condition
Mode: Input.
Type: Condition.
Presence: Required.
Function: Specifies the condition to be satisfied.
Return Value:
Type: Entity.
Function: First entity that satisfies "condition".
Exceptions: None.

- **Select**

Objective: This method returns a collection of entities within a Set that satisfies a specified condition.
Parameters:
condition
Mode: Input.
Type: Condition.
Presence: Required.
Function: Specifies the condition to be satisfied.
Return Value:
Type: Set.
Function: Set of entities that satisfy "condition".
Exceptions: None.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- **Remove**

```
Objective:      This method removes and returns the first entity
                within a Set that satisfies a specified condition.
                If a condition is not specified, then it removes and
                returns the first entity within a Set.
Parameters:
   condition
      Mode:     Input.
      Type:     Condition.
      Presence: Optional.
      Function: Specifies the condition to be satisfied.
Return Value:
      Type:     Entity.
      Function: First entity that satisfies "condition".
Exceptions:     None.
```

- **RemoveThis**

```
Objective:      This method removes a particular entity from a Set.
Parameters:
   thisEntity
      Mode:     Input.
      Type:     Entity.
      Presence: Required.
      Function: Specifies the entity to be removed.
Return Value:   None.
Exceptions:
   "Entity to be removed not found"
      Cause:    Specified entity does not exist within the Set.
      Effect:   Simulation terminates.
```

- **Includes**

```
Objective:      This method ascertains if a particular entity exists
                within a Set.
Parameters:
   thisEntity
      Mode:     Input.
      Type:     Entity.
      Presence: Required.
      Function: Specifies the entity to be checked.
Return Value:
      Type:     Boolean.
      Function: TRUE if entity exists, else FALSE.
Exceptions:     None.
```

- **Exists**

```
Objective:      This method ascertains if at least 1 entity exists
                within a Set that satisfies a specified condition.
Parameters:
   condition
      Mode:     Input.
      Type:     Condition.
      Presence: Optional.
      Function: Specifies the condition to be satisfied.
Return Value:
      Type:     Boolean.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
      Function:  TRUE if at least one entity that satisfies
                 "condition" exists within the set, else FALSE.
Exceptions:      None.
```

- **Each**

```
Objective:       This method ascertains if all the entities within a
                 Set satisfy a particular specified condition.
Parameters:
   condition
      Mode:      Input.
      Type:      Condition.
      Presence:  Optional.
      Function:  Specifies the condition to be satisfied.
Return Value:
      Type:      Boolean.
      Function:  TRUE if all entities within the set satisfy
                 "condition", else FALSE.
Exceptions:      None.
```

- **ForEach**

```
Objective:       This method performs a particular operation
                 specified by "routine" on each entity within a Set.
Parameters:
   routine
      Mode:      Input.
      Type:      Routine.
      Presence:  Optional.
      Function:  Specifies a reference to the routine.
Return Value:    None.
Exceptions:      None.
```

- **Dump**

```
Objective:       This method outputs all entities within a Set, using
                 the specified output routine.
Parameters:
   routine
      Mode:      Input.
      Type:      Routine.
      Presence:  Optional.
      Function:  Specifies a reference to the output routine.
Return Value:    None.
Exceptions:      None.
```

- **Empty**

```
Objective:       This method empties a Set without disposing any of
                 the entities within the Set.
Parameters:      None.
Return Value:    None.
Exceptions:      None.
```

- **EmptyDispose**

```
Objective:       This method empties a Set and disposes of its
                 contents using the specified routine.
Parameters:
   routine
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
      Mode:      Input.
      Type:      Routine.
      Presence:  Optional.
      Function:  Specifies a reference to the routine.
Return Value:    None.
Exceptions:      None.
```

A.6.2 *FIFO* Interface

Interface Diagram:

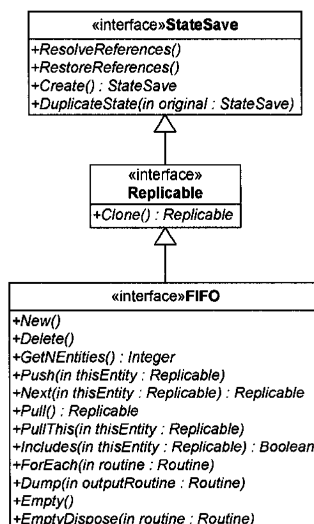


Figure A19. *FIFO* Interface

Methods:

- **ResolveReferences**

```
Objective:       This method resolves references for dynamically
                 initialized attributes within a FIFO.
Parameters:      None.
Return Value:    None.
Exceptions:      None.
```

- **RestoreReferences**

```
Objective:       This method restores references for dynamically
                 initialized attributes within a FIFO.
Parameters:      None.
Return Value:    None.
Exceptions:      None.
```

- **Create**

```
Objective:       This method creates an empty shell of a FIFO during
                 the state duplication process of a non-terminating
                 state save operation.
Parameters:      None.
Return Value:
      Type:      StateSave.
      Function:  Empty shell of FIFO.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
Exceptions:      None.
```

- **DuplicateState**

```
Objective:       This method copies the state of "original" to the
                 current FIFO during the state duplication process of
                 a non-terminating state save operation.
Parameters:
   original
      Mode:      Input.
      Type:      StateSave.
      Presence:  Required.
      Function:  Specifies FIFO whose state is to be copied.
Return Value:    None.
Exceptions:      None.
```

- **Clone**

```
Objective:       This method creates a clone of a FIFO with all
                 entites currently within the FIFO existing in the
                 clone.
Parameters:      None.
Return Value:
      Type:      Replicable.
      Function:  Clone of a FIFO.
Exceptions:      None.
```

- **New**

```
Objective:       This method initializes a FIFO.
Parameters:      None.
Return Value:    None.
Exceptions:      None.
```

- **Delete**

```
Objective:       This method disposes a FIFO.
Parameters:      None.
Return Value:    None.
Exceptions:      None.
```

- **GetNEntities**

```
Objective:       This method returns the current number of entities
                 within a FIFO.
Parameters:      None.
Return Value:
      Type:      Integer.
      Function:  Current number of entities in the FIFO.
Exceptions:      None.
```

- **Push**

```
Objective:       This method adds an entity to a FIFO.
Parameters:
   thisEntity
      Mode:      Input.
      Type:      Entity.
      Presence:  Required.
      Function:  Specifies the entity to be added to the FIFO.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
Return Value:     None.
Exceptions:       None.
```

- **Next**

```
Objective:        This method returns the next entity after
                  "thisEntity" within a FIFO. If "thisEntity" is NULL,
                  it returns the first entity within the FIFO. A NULL
                  is returned when there is no more unseen entity in
                  the FIFO. If this method is continuously called till
                  a NULL value is returned, then a new entity is
                  returned each time.
Parameters:
   thisEntity
      Mode:       Input.
      Type:       Entity.
      Presence:   Required.
      Function:   Specifies the current entity.
Return Value:
      Type:       Entity.
      Function:   Next entity after "thisEntity".
Exceptions:       None.
```

- **Pull**

```
Objective:        This method removes and returns the first entity
                  within a FIFO.
Parameters:       None.
Return Value:
      Type:       Entity.
      Function:   First entity that satisfies "condition".
Exceptions:       None.
```

- **PullThis**

```
Objective:        This method removes a particular entity from a FIFO.
Parameters:
   thisEntity
      Mode:       Input.
      Type:       Entity.
      Presence:   Required.
      Function:   Specifies the entity to be removed.
Return Value:     None.
Exceptions:
   "Entity to be removed not found"
      Cause:      Specified entity does not exist within the FIFO.
      Effect:     Simulation terminates.
```

- **Includes**

```
Objective:        This method ascertains if a particular entity exists
                  within a FIFO.
Parameters:
   thisEntity
      Mode:       Input.
      Type:       Entity.
      Presence:   Required.
      Function:   Specifies the entity to be checked.
Return Value:
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
      Type:       Boolean.
      Function:   TRUE if entity exists, else FALSE.
Exceptions:       None.
```

- **ForEach**

```
Objective:        This method performs a particular operation
                  specified by "routine" on each entity within a FIFO.
Parameters:
   routine
      Mode:       Input.
      Type:       Routine.
      Presence:   Optional.
      Function:   Specifies a reference to the routine.
Return Value:     None.
Exceptions:       None.
```

- **Dump**

```
Objective:        This method outputs all entities within a FIFO,
                  using the specified output routine.
Parameters:
   routine
      Mode:       Input.
      Type:       Routine.
      Presence:   Optional.
      Function:   Specifies a reference to the output routine.
Return Value:     None.
Exceptions:       None.
```

- **Empty**

```
Objective:        This method empties a FIFO without disposing any of
                  the entities within the FIFO.
Parameters:       None.
Return Value:     None.
Exceptions:       None.
```

- **EmptyDispose**

```
Objective:        This method empties a FIFO and disposes of its
                  contents using the specified routine.
Parameters:
   routine
      Mode:       Input.
      Type:       Routine.
      Presence:   Optional.
      Function:   Specifies a reference to the routine.
Return Value:     None.
Exceptions:       None.
```

A.6.3 *LIFO* Interface

Interface Diagram:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

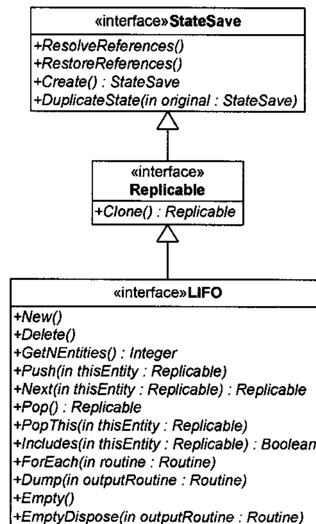


Figure A20. *LIFO* Interface

Methods:

- **ResolveReferences**

  Objective: This method resolves references for dynamically initialized attributes within a LIFO.
  Parameters: None.
  Return Value: None.
  Exceptions: None.

- **RestoreReferences**

  Objective: This method restores references for dynamically initialized attributes within a LIFO.
  Parameters: None.
  Return Value: None.
  Exceptions: None.

- **Create**

  Objective: This method creates an empty shell of a LIFO during the state duplication process of a non-terminating state save operation.
  Parameters: None.
  Return Value:
  Type: StateSave.
  Function: Empty shell of LIFO.
  Exceptions: None.

- **DuplicateState**

  Objective: This method copies the state of "original" to the current LIFO during the state duplication process of a non-terminating state save operation.
  Parameters:
  original
  Mode: Input.
  Type: StateSave.
  Presence: Required.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
      Function:   Specifies LIFO whose state is to be copied.
Return Value:     None.
Exceptions:       None.
```

- **Clone**

```
Objective:        This method creates a clone of a LIFO with all
                  entites currently within the LIFO existing in the
                  clone.
Parameters:       None.
Return Value:
      Type:       Replicable.
      Function:   Clone of a LIFO.
Exceptions:       None.
```

- **New**

```
Objective:        This method initializes a LIFO.
Parameters:       None.
Return Value:     None.
Exceptions:       None.
```

- **Delete**

```
Objective:        This method disposes a LIFO.
Parameters:       None.
Return Value:     None.
Exceptions:       None.
```

- **GetNEntities**

```
Objective:        This method returns the current number of entities
                  within a LIFO.
Parameters:       None.
Return Value:
      Type:       Integer.
      Function:   Current number of entities in the LIFO.
Exceptions:       None.
```

- **Push**

```
Objective:        This method adds an entity to a LIFO.
Parameters:
   thisEntity
      Mode:       Input.
      Type:       Entity.
      Presence:   Required.
      Function:   Specifies the entity to be added to the LIFO.
Return Value:     None.
Exceptions:       None.
```

- **Next**

```
Objective:        This method returns the next entity after
                  "thisEntity" within a LIFO. If "thisEntity" is NULL,
                  it returns the first entity within the LIFO. A NULL
                  is returned when there is no more unseen entity in
                  the LIFO. If this method is continuously called till
                  a NULL value is returned, then a new entity is
                  returned each time.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
Parameters:
   thisEntity
      Mode:      Input.
      Type:      Entity.
      Presence:  Required.
      Function:  Specifies the current entity.
Return Value:
      Type:      Entity.
      Function:  Next entity after "thisEntity".
Exceptions:      None.
```

- **Pop**

```
Objective:       This method removes and returns the last entity
                 within a LIFO.
Parameters:      None.
Return Value:
      Type:      Entity.
      Function:  First entity that satisfies "condition".
Exceptions:      None.
```

- **PopThis**

```
Objective:       This method removes a particular entity from a LIFO.
Parameters:
   thisEntity
      Mode:      Input.
      Type:      Entity.
      Presence:  Required.
      Function:  Specifies the entity to be removed.
Return Value:    None.
Exceptions:
   "Entity to be removed not found"
      Cause:     Specified entity does not exist within the LIFO.
      Effect:    Simulation terminates.
```

- **Includes**

```
Objective:       This method ascertains if a particular entity exists
                 within a LIFO.
Parameters:
   thisEntity
      Mode:      Input.
      Type:      Entity.
      Presence:  Required.
      Function:  Specifies the entity to be checked.
Return Value:
      Type:      Boolean.
      Function:  TRUE if entity exists, else FALSE.
Exceptions:      None.
```

- **ForEach**

```
Objective:       This method performs a particular operation
                 specified by "routine" on each entity within a LIFO.
Parameters:
   routine
      Mode:      Input.
      Type:      Routine.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
      Presence:  Optional.
      Function:  Specifies a reference to the routine.
Return Value:    None.
Exceptions:      None.
```

- **Dump**

```
Objective:       This method outputs all entities within a LIFO,
                 using the specified output routine.
Parameters:
   routine
      Mode:      Input.
      Type:      Routine.
      Presence:  Optional.
      Function:  Specifies a reference to the output routine.
Return Value:    None.
Exceptions:      None.
```

- **Empty**

```
Objective:       This method empties a LIFO without disposing any of
                 the entities within the LIFO.
Parameters:      None.
Return Value:    None.
Exceptions:      None.
```

- **EmptyDispose**

```
Objective:       This method empties a LIFO and disposes of its
                 contents using the specified routine.
Parameters:
   routine
      Mode:      Input.
      Type:      Routine.
      Presence:  Optional.
      Function:  Specifies a reference to the routine.
Return Value:    None.
Exceptions:      None.
```

A.6.4 *BinaryTree* Interface

Interface Diagram:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

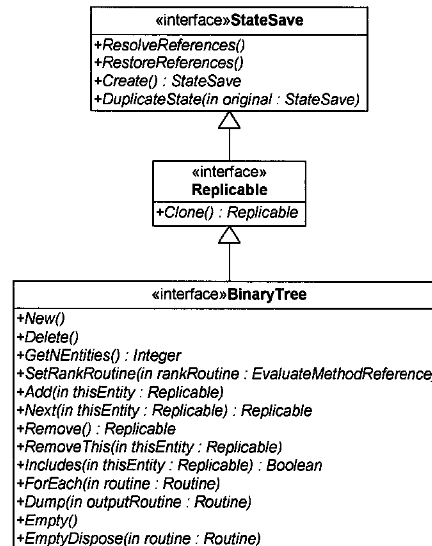


Figure A21. *BinaryTree* Interface

Methods:

- **ResolveReferences**
  Objective: This method resolves references for dynamically initialized attributes within a BinaryTree.
  Parameters: None.
  Return Value: None.
  Exceptions: None.

- **RestoreReferences**
  Objective: This method restores references for dynamically initialized attributes within a BinaryTree.
  Parameters: None.
  Return Value: None.
  Exceptions: None.

- **Create**
  Objective: This method creates an empty shell of a BinaryTree during the state duplication process of a non-terminating state save operation.
  Parameters: None.
  Return Value:
  Type: StateSave.
  Function: Empty shell of BinaryTree.
  Exceptions: None.

- **DuplicateState**
  Objective: This method copies the state of "original" to the current BinaryTree during the state duplication process of a non-terminating state save operation.
  Parameters:
  original
  Mode: Input.
  Type: StateSave.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
        Presence:  Required.
        Function:  Specifies BinaryTree whose state is to be copied.
  Return Value:    None.
  Exceptions:      None.
```

- **Clone**

```
  Objective:       This method creates a clone of a BinaryTree with all
                   entites currently within the BinaryTree existing in
                   the clone.
  Parameters:      None.
  Return Value:
        Type:      Replicable.
        Function:  Clone of a BinaryTree.
  Exceptions:      None.
```

- **New**

```
  Objective:       This method initializes a BinaryTree.
  Parameters:      None.
  Return Value:    None.
  Exceptions:      None.
```

- **Delete**

```
  Objective:       This method disposes a BinaryTree.
  Parameters:      None.
  Return Value:    None.
  Exceptions:      None.
```

- **GetNEntities**

```
  Objective:       This method returns the current number of entities
                   within a BinaryTree.
  Parameters:      None.
  Return Value:
        Type:      Integer.
        Function:  Current number of entities in the BinaryTree.
  Exceptions:      None.
```

- **SetRankRoutine**

```
  Objective:       This method sets a ranking routine for a BinaryTree,
                   to be used when a new entity is added to the
                   BinaryTree.
  Parameters:
     rankRoutine
        Mode:      Input.
        Type:      EvaluateMethodReference.
        Presence:  Required.
        Function:  Specifies a reference to the ranking routine.
  Return Value:    None.
  Exceptions:      None.
```

- **Add**

```
  Objective:       This method adds an entity to a BinaryTree.
  Parameters:
     thisEntity
        Mode:      Input.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
        Type:      Entity.
        Presence:  Required.
        Function:  Specifies the entity to be added to the BinaryTree.
Return Value:      None.
Exceptions:
   "No ranking routine defined"
        Cause:     A routine to rank the entities has not been defined.
        Effect:    Simulation terminates.
```

- **Next**

```
Objective:         This method returns the next entity after
                   "thisEntity" within a BinaryTree. If "thisEntity" is
                   NULL, it returns the first entity within the
                   BinaryTree. A NULL is returned when there is no more
                   unseen entity in the BinaryTree. If this method is
                   continuously called till a NULL value is returned,
                   then a new entity is returned each time.
Parameters:
   thisEntity
        Mode:      Input.
        Type:      Entity.
        Presence:  Required.
        Function:  Specifies the current entity.
Return Value:
        Type:      Entity.
        Function:  Next entity after "thisEntity".
Exceptions:        None.
```

- **Remove**

```
Objective:         This method removes and returns the first entity
                   within a BinaryTree.
Parameters:        None.
Return Value:
        Type:      Entity.
        Function:  First entity that satisfies "condition".
Exceptions:        None.
```

- **RemoveThis**

```
Objective:         This method removes a particular entity from a
                   BinaryTree.
Parameters:
   thisEntity
        Mode:      Input.
        Type:      Entity.
        Presence:  Required.
        Function:  Specifies the entity to be removed.
Return Value:      None.
Exceptions:
   "Entity to be removed not found"
        Cause:     Specified entity does not exist within the
                   BinaryTree.
        Effect:    Simulation terminates.
```

- **Includes**

```
Objective:         This method ascertains if a particular entity exists
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
                    within a BinaryTree.
Parameters:
   thisEntity
      Mode:         Input.
      Type:         Entity.
      Presence:     Required.
      Function:     Specifies the entity to be checked.
Return Value:
      Type:         Boolean.
      Function:     TRUE if entity exists, else FALSE.
Exceptions:         None.
```

- **ForEach**

```
Objective:          This method performs a particular operation
                    specified by "routine" on each entity within a
                    BinaryTree.
Parameters:
   routine
      Mode:         Input.
      Type:         Routine.
      Presence:     Optional.
      Function:     Specifies a reference to the routine.
Return Value:       None.
Exceptions:         None.
```

- **Dump**

```
Objective:          This method outputs all entities within a
                    BinaryTree, using the specified output routine.
Parameters:
   routine
      Mode:         Input.
      Type:         Routine.
      Presence:     Optional.
      Function:     Specifies a reference to the output routine.
Return Value:       None.
Exceptions:         None.
```

- **Empty**

```
Objective:          This method empties a BinaryTree without disposing
                    any of the entities within the BinaryTree.
Parameters:         None.
Return Value:       None.
Exceptions:         None.
```

- **EmptyDispose**

```
Objective:          This method empties a BinaryTree and disposes of its
                    contents using the specified routine.
Parameters:
   routine
      Mode:         Input.
      Type:         Routine.
      Presence:     Optional.
      Function:     Specifies a reference to the routine.
Return Value:       None.
Exceptions:         None.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A.6.5 *PriorityQueue* Interface

Interface Diagram:

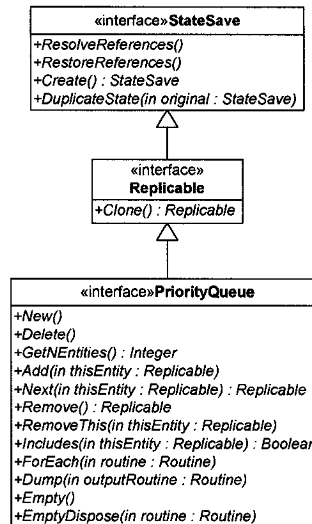


Figure A22. *PriorityQueue* Interface

Methods:

- **ResolveReferences**
  Objective: This method resolves references for dynamically initialized attributes within a PriorityQueue.
  Parameters: None.
  Return Value: None.
  Exceptions: None.

- **RestoreReferences**
  Objective: This method restores references for dynamically initialized attributes within a PriorityQueue.
  Parameters: None.
  Return Value: None.
  Exceptions: None.

- **Create**
  Objective: This method creates an empty shell of a PriorityQueue during the state duplication process of a non-terminating state save operation.
  Parameters: None.
  Return Value:
  Type: StateSave.
  Function: Empty shell of PriorityQueue.
  Exceptions: None.

- **DuplicateState**
  Objective: This method copies the state of "original" to the current PriorityQueue during the state duplication process of a non-terminating state save operation.
  Parameters:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

original
    Mode: Input.
    Type: StateSave.
    Presence: Required.
    Function: Specifies PriorityQueue whose state is to be copied.
Return Value: None.
Exceptions: None.

- **Clone**
  Objective: This method creates a clone of a PriorityQueue with all entites currently within the PriorityQueue existing in the clone.
  Parameters: None.
  Return Value:
      Type: Replicable.
      Function: Clone of a PriorityQueue.
  Exceptions: None.

- **New**
  Objective: This method initializes a PriorityQueue.
  Parameters: None.
  Return Value: None.
  Exceptions: None.

- **Delete**
  Objective: This method disposes a PriorityQueue.
  Parameters: None.
  Return Value: None.
  Exceptions: None.

- **GetNEntities**
  Objective: This method returns the current number of entities within a PriorityQueue.
  Parameters: None.
  Return Value:
      Type: Integer.
      Function: Current number of entities in the PriorityQueue.
  Exceptions: None.

- **Add**
  Objective: This method adds an entity to a PriorityQueue, based on an assigned priority for the entity.
  Parameters:
    thisEntity
      Mode: Input.
      Type: Entity.
      Presence: Required.
      Function: Specifies the entity to be added to the PriorityQueue.
  Return Value: None.
  Exceptions: None.

- **Next**
  Objective: This method returns the next entity after

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

"thisEntity" within a PriorityQueue. If "thisEntity" is NULL, it returns the first entity within the PriorityQueue. A NULL is returned when there is no more unseen entity in the PriorityQueue. If this method is continuously called till a NULL value is returned, then a new entity is returned each time.

Parameters:
thisEntity
Mode: Input.
Type: Entity.
Presence: Required.
Function: Specifies the current entity.
Return Value:
Type: Entity.
Function: Next entity after "thisEntity".
Exceptions: None.

- **Remove**

Objective: This method removes and returns the first entity within a PriorityQueue.
Parameters: None.
Return Value:
Type: Entity.
Function: First entity that satisfies "condition".
Exceptions: None.

- **RemoveThis**

Objective: This method removes a particular entity from a PriorityQueue.
Parameters:
thisEntity
Mode: Input.
Type: Entity.
Presence: Required.
Function: Specifies the entity to be removed.
Return Value: None.
Exceptions:
"Entity to be removed not found"
Cause: Specified entity does not exist within the PriorityQueue.
Effect: Simulation terminates.

- **Includes**

Objective: This method ascertains if a particular entity exists within a PriorityQueue.
Parameters:
thisEntity
Mode: Input.
Type: Entity.
Presence: Required.
Function: Specifies the entity to be checked.
Return Value:
Type: Boolean.
Function: TRUE if entity exists, else FALSE.
Exceptions: None.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- **ForEach**

  Objective: This method performs a particular operation specified by "routine" on each entity within a PriorityQueue.

  Parameters:

  routine

  Mode: Input.
  Type: Routine.
  Presence: Optional.
  Function: Specifies a reference to the routine.

  Return Value: None.
  Exceptions: None.

- **Dump**

  Objective: This method outputs all entities within a PriorityQueue, using the specified output routine.

  Parameters:

  routine

  Mode: Input.
  Type: Routine.
  Presence: Optional.
  Function: Specifies a reference to the output routine.

  Return Value: None.
  Exceptions: None.

- **Empty**

  Objective: This method empties a PriorityQueue without disposing any of the entities within the PriorityQueue.

  Parameters: None.
  Return Value: None.
  Exceptions: None.

- **EmptyDispose**

  Objective: This method empties a PriorityQueue and disposes of its contents using the specified routine.

  Parameters:

  routine

  Mode: Input.
  Type: Routine.
  Presence: Optional.
  Function: Specifies a reference to the routine.

  Return Value: None.
  Exceptions: None.

A.6.6 *QueueWithStatistics* Interface

Interface Diagram:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

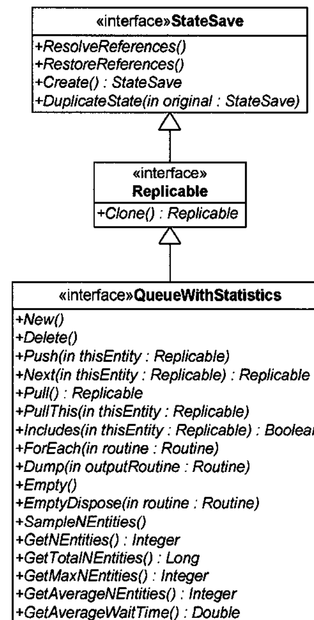


Figure A23. *QueueWithStatistics* Interface

Methods:

- **ResolveReferences**

  Objective: This method resolves references for dynamically initialized attributes within a QueueWithStatistics.
  Parameters: None.
  Return Value: None.
  Exceptions: None.

- **RestoreReferences**

  Objective: This method restores references for dynamically initialized attributes within a QueueWithStatistics.
  Parameters: None.
  Return Value: None.
  Exceptions: None.

- **Create**

  Objective: This method creates an empty shell of a QueueWithStatistics during the state duplication process of a non-terminating state save operation.
  Parameters: None.
  Return Value:
  Type: StateSave.
  Function: Empty shell of QueueWithStatistics.
  Exceptions: None.

- **DuplicateState**

  Objective: This method copies the state of "original" to the current QueueWithStatistics during the state duplication process of a non-terminating state save operation.
  Parameters:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

original

```
      Mode:      Input.
      Type:      StateSave.
      Presence:  Required.
      Function:  Specifies QueueWithStatistics whose state is to be
                 copied.
Return Value:    None.
Exceptions:      None.
```

- **Clone**

```
Objective:       This method creates a clone of a QueueWithStatistics
                 with all entities currently within the
                 QueueWithStatistics existing in the clone.
Parameters:      None.
Return Value:
      Type:      Replicable.
      Function:  Clone of a QueueWithStatistics.
Exceptions:      None.
```

- **New**

```
Objective:       This method initializes a QueueWithStatistics.
Parameters:      None.
Return Value:    None.
Exceptions:      None.
```

- **Delete**

```
Objective:       This method disposes a QueueWithStatistics.
Parameters:      None.
Return Value:    None.
Exceptions:      None.
```

- **Push**

```
Objective:       This method adds an entity to a QueueWithStatistics.
Parameters:
```

thisEntity

```
      Mode:      Input.
      Type:      Entity.
      Presence:  Required.
      Function:  Specifies the entity to be added to the
                 QueueWithStatistics.
Return Value:    None.
Exceptions:      None.
```

- **Next**

```
Objective:       This method returns the next entity after
                 "thisEntity" within a QueueWithStatistics. If
                 "thisEntity" is NULL, it returns the first entity
                 within the QueueWithStatistics. A NULL is returned
                 when there is no more unseen entity in the
                 QueueWithStatistics. If this method is continuously
                 called till a NULL value is returned, then a new
                 entity is returned each time.
Parameters:
```

thisEntity

```
      Mode:      Input.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
        Type:       Entity.
        Presence:   Required.
        Function:   Specifies the current entity.
Return Value:
        Type:       Entity.
        Function:   Next entity after "thisEntity".
Exceptions:         None.
```

- **Pull**

```
Objective:          This method removes and returns the first entity
                    within a QueueWithStatistics.
Parameters:         None.
Return Value:
        Type:       Entity.
        Function:   First entity that satisfies "condition".
Exceptions:         None.
```

- **PullThis**

```
Objective:          This method removes a particular entity from a
                    QueueWithStatistics.
Parameters:
    thisEntity
        Mode:       Input.
        Type:       Entity.
        Presence:   Required.
        Function:   Specifies the entity to be removed.
Return Value:       None.
Exceptions:
    "Entity to be removed not found"
        Cause:      Specified entity does not exist within the
                    QueueWithStatistics.
        Effect:     Simulation terminates.
```

- **Includes**

```
Objective:          This method ascertains if a particular entity exists
                    within a QueueWithStatistics.
Parameters:
    thisEntity
        Mode:       Input.
        Type:       Entity.
        Presence:   Required.
        Function:   Specifies the entity to be checked.
Return Value:
        Type:       Boolean.
        Function:   TRUE if entity exists, else FALSE.
Exceptions:         None.
```

- **ForEach**

```
Objective:          This method performs a particular operation
                    specified by "routine" on each entity within a
                    QueueWithStatistics.
Parameters:
    routine
        Mode:       Input.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
        Type:       Routine.
        Presence:   Optional.
        Function:   Specifies a reference to the routine.
    Return Value:   None.
    Exceptions:     None.
```

- **Dump**

```
    Objective:      This method outputs all entities within a
                    QueueWithStatistics, using the specified output
                    routine.
    Parameters:
      routine
        Mode:       Input.
        Type:       Routine.
        Presence:   Optional.
        Function:   Specifies a reference to the output routine.
    Return Value:   None.
    Exceptions:     None.
```

- **Empty**

```
    Objective:      This method empties a QueueWithStatistics without
                    disposing any of the entities within the
                    QueueWithStatistics.
    Parameters:     None.
    Return Value:   None.
    Exceptions:     None.
```

- **EmptyDispose**

```
    Objective:      This method empties a QueueWithStatistics and
                    disposes of its contents using the specified
                    routine.
    Parameters:
      routine
        Mode:       Input.
        Type:       Routine.
        Presence:   Optional.
        Function:   Specifies a reference to the routine.
    Return Value:   None.
    Exceptions:     None.
```

- **SampleNEntities**

```
    Objective:      This method records the current number of entities
                    within a QueueWithStatistics.
    Parameters:     None.
    Return Value:   None.
    Exceptions:     None.
```

- **GetNEntities**

```
    Objective:      This method returns the current number of entities
                    within a QueueWithStatistics.
    Parameters:     None.
    Return Value:
        Type:       Integer.
        Function:   Current number of entities in the
                    QueueWithStatistics.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Exceptions: None.

- **GetTotalNEntities**

```
Objective:       This method returns the total number of entities to
                 exist within a QueueWithStatistics for the duration
                 of the simulation.
Parameters:      None.
Return Value:
      Type:      Long.
      Function:  Total number of entities in the QueueWithStatistics.
Exceptions:      None.
```

- **GetMaxNEntities**

```
Objective:       This method returns the maximum number of entities
                 to exist within a QueueWithStatistics at any
                 instant, for the duration of the simulation.
Parameters:      None.
Return Value:
      Type:      Integer.
      Function:  Maximum number of entities in the
                 QueueWithStatistics.
Exceptions:      None.
```

- **GetAverageNEntities**

```
Objective:       This method returns the average number of entities
                 to exist within a QueueWithStatistics for the
                 duration of the simulation.
Parameters:      None.
Return Value:
      Type:      Integer.
      Function:  Average number of entities in the
                 QueueWithStatistics.
Exceptions:      None.
```

- **GetAverageWaitTime**

```
Objective:       This method returns the average time that each
                 entity existed within a QueueWithStatistics for the
                 duration of the simulation.
Parameters:      None.
Return Value:
      Type:      Double.
      Function:  Average wait time of each entity in the
                 QueueWithStatistics.
Exceptions:      None.
```

A.4.7 *EntityCounter* Interface

Interface Diagram:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

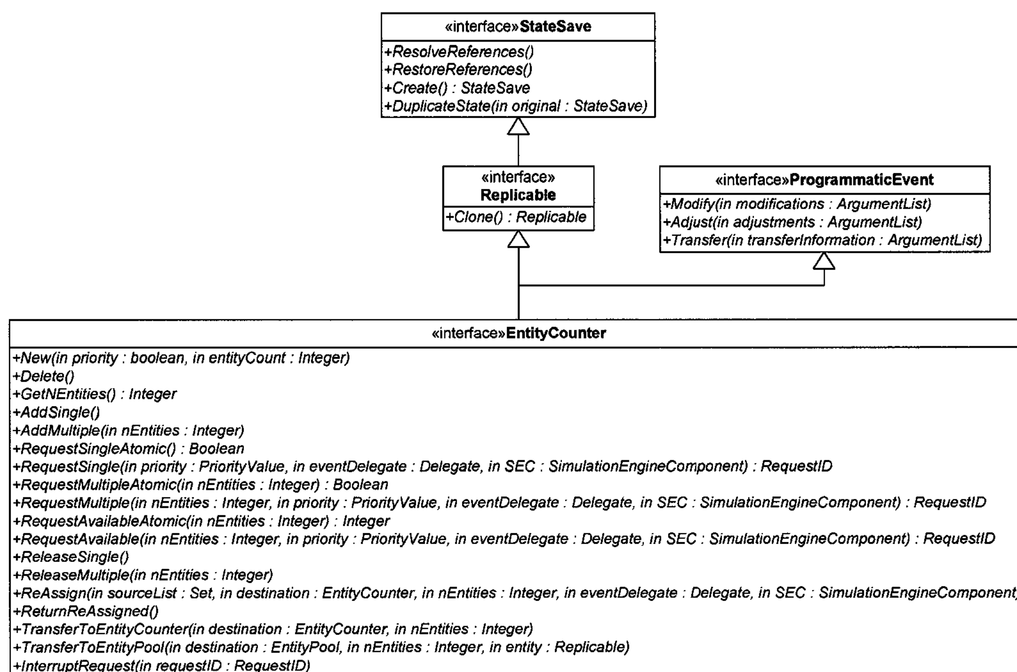


Figure A24. *EntityCounter* Interface

Methods:

- **ResolveReferences**

  Objective: This method resolves references for dynamically initialized attributes within an EntityCounter.
  Parameters: None.
  Return Value: None.
  Exceptions: None.

- **RestoreReferences**

  Objective: This method restores references for dynamically initialized attributes within an EntityCounter.
  Parameters: None.
  Return Value: None.
  Exceptions: None.

- **Create**

  Objective: This method creates an empty shell of an EntityCounter during the state duplication process of a non-terminating state save operation.
  Parameters: None.
  Return Value:
  Type: StateSave.
  Function: Empty shell of EntityCounter.
  Exceptions: None.

- **DuplicateState**

  Objective: This method copies the state of "original" to the current EntityCounter during the state duplication process of a non-terminating state save operation.
  Parameters:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
   original
      Mode:      Input.
      Type:      StateSave.
      Presence:  Required.
      Function:  Specifies EntityCounter whose state is to be copied.
Return Value:    None.
Exceptions:      None.
```

- **Clone**

```
Objective:       This method creates a clone of an EntityCounter with
                 the number of entites in the clone equal to the
                 number of entities currently within the
                 EntityCounter.
Parameters:      None.
Return Value:
      Type:      Replicable.
      Function:  Clone of an EntityCounter.
Exceptions:      None.
```

- **Modify**

```
Objective:       This method is not defined for an EntityCounter.
Parameters:
   modifications
      Mode:      Input.
      Type:      ArgumentList.
      Presence:  Required.
      Function:  Specifies the new parameters.
Return Value:    None.
Exceptions:
   "Modify method not defined for this interface"
      Cause:     Method not defined for an EntityCounter.
      Effect:    Simulation terminates.
```

- **Adjust**

```
Objective:       This method increases or decreases the number of
                 entities within an EntityCounter based on an offset.
                 If insufficient entities currently exist to satisfy
                 a decrease in number of entities, all existing
                 entities are immediately removed, and all entities
                 subsequently released back to the EntityCounter, are
                 removed till the decrease is complete.
Parameters:
   adjustments
      Mode:      Input.
      Type:      ArgumentList.
      Presence:  Required.
      Function:  Specifies the new parameters.
Return Value:    None.
Exceptions:
   "Insufficient arguments for Adjust method"
      Cause:     Not enough arguments within "adjustments".
      Effect:    Simulation terminates.
```

- **Transfer**

```
Objective:       This method invokes TransferToEntityCounter or
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TransferToEntityPool for an EntityCounter, with the appropriate parameters based on "transferInformation".

Parameters:

transferInformation

Mode: Input.
Type: ArgumentList.
Presence: Required.
Function: Specifies the new parameters.

Return Value: None.

Exceptions:

"Insufficient arguments for Transfer method"

Cause: Not enough arguments within "transferInformation".
Effect: Simulation terminates.

- **New**

Objective: This method initializes an EntityCounter.

Parameters:

priority

Mode: Input.
Type: Boolean.
Presence: Required.
Function: Specifies if unsatisfied requests are queued based on priority within the EntityCounter.

entityCount

Mode: Input.
Type: Integer.
Presence: Optional.
Function: Specifies the initial number of entities within the EntityCounter.

Return Value: None.
Exceptions: None.

- **Delete**

Objective: This method disposes an EntityCounter.
Parameters: None.
Return Value: None.
Exceptions: None.

- **GetNEntities**

Objective: This method returns the current number of entities within an EntityCounter.
Parameters: None.
Return Value:

Type: Integer.
Function: Current number of entities in the EntityCounter.

Exceptions: None.

- **AddSingle**

Objective: This method increments the count of entities within an EntityCounter by 1.
Parameters: None.
Return Value: None.
Exceptions: None.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- **AddMultiple**

```
Objective:       This method increments the count of entities within
                 an EntityCounter by "nEntities".
Parameters:
   nEntities
      Mode:      Input.
      Type:      Integer.
      Presence:  Required.
      Function:  Specifies the number of entities to be added.
Return Value:    None.
Exceptions:
   "Invalid number of entities to be added"
      Cause:     "nEntities" < 0.
      Effect:    Simulation terminates.
```

- **RequestSingleAtomic**

```
Objective:       This method makes an atomic request for a single
                 entity to an EntityCounter. If an entity is
                 available, then a TRUE value is returned else a
                 FALSE value is returned.
Parameters:      None.
Return Value:
      Type:      Boolean.
      Function:  TRUE if entity available, else FALSE.
Exceptions:      None.
```

- **RequestSingle**

```
Objective:       This method makes a request for a single entity to
                 an EntityCounter. If an entity is available, then
                 the Delegate is immediately scheduled for execution.
                 If an entity is not available, then the request is
                 queued and serviced in a FCFS manner. If a priority
                 has been specified, then the request is queued with
                 the appropriate priority relative to other pending
                 requests.
Parameters:
   priority
      Mode:      Input.
      Type:      Priority Value.
      Presence:  Optional.
      Function:  Specifies the priority of the request relative to
                 pending requests.
   eventDelegate
      Mode:      Input.
      Type:      Delegate.
      Presence:  Required.
      Function:  Specifies the delegate to be scheduled when request
                 is satisfied.
Return Value:
      Type:      RequestID.
      Function:  -1 if request is satisfied, else ID of pending
                 request.
Exceptions:      None.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- **RequestMultipleAtomic**

Objective: This method makes an atomic request for "nEntities" to an EntityCounter. If the requested entities are available, then a TRUE value is returned else a FALSE value is returned.

Parameters:

nEntities

Mode: Input.
Type: Integer.
Presence: Required.
Function: Specifies the number of requested entities.

Return Value:

Type: Boolean.
Function: TRUE if entities are available, else FALSE.

Exceptions:

"Invalid number of requested entities"

Cause: "nEntities" < 0.
Effect: Simulation terminates.

- **RequestMultiple**

Objective: This method makes a request for "nEntities" to an EntityCounter. If the requested entities are available, then the Delegate is immediately scheduled for execution. If the requested entities are not available, then the request is queued and serviced in a FCFS manner. If a priority has been specified, then the request is queued with the appropriate priority relative to other pending requests.

Parameters:

nEntities

Mode: Input.
Type: Integer.
Presence: Required.
Function: Specifies the number of requested entities.

priority

Mode: Input.
Type: Priority Value.
Presence: Optional.
Function: Specifies the priority of the request relative to pending requests.

eventDelegate

Mode: Input.
Type: Delegate.
Presence: Required.
Function: Specifies the delegate to be scheduled when request is satisfied.

Return Value:

Type: RequestID.
Function: -1 if request is satisfied, else ID of pending request.

Exceptions:

"Invalid number of requested entities"

Cause: "nEntities" < 0.
Effect: Simulation terminates.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- **RequestAvailableAtomic**

Objective: This method makes an atomic request for "nEntities" to an EntityCounter. If no entities are available, then a value of -1 is returned, else the minimum value between "nEntities" and currently available entities in the EntityCounter is returned.

Parameters:

nEntities

Mode: Input.
Type: Integer.
Presence: Required.
Function: Specifies the number of requested entities.

Return Value:

Type: Integer.
Function: minimum value of "nEntities" and available entities, -1 if none available.

Exceptions:

"Invalid number of requested entities"

Cause: "nEntities" < 0.
Effect: Simulation terminates.

- **RequestAvailable**

Objective: This method makes a request for "nEntities" to an EntityCounter. If at least 1 entity is available, then the Delegate is immediately scheduled for execution, with the number of available entities passed as a return argument by value for the event method ArgumentList within the Delegate. If no entities are available or if the number of available entities is less than "nEntities", then the request is queued and serviced in a FCFS manner. If a priority has been specified, then the request is queued with the appropriate priority relative to other pending requests. The Delegate is scheduled for execution every time new entities become available till the request is completely satisfied.

Parameters:

nEntities

Mode: Input.
Type: Integer.
Presence: Required.
Function: Specifies the number of requested entities.

priority

Mode: Input.
Type: Priority Value.
Presence: Optional.
Function: Specifies the priority of the request relative to pending requests.

eventDelegate

Mode: Input.
Type: Delegate.
Presence: Required.
Function: Specifies the delegate to be scheduled when request is satisfied.

Return Value:

Type: RequestID.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Function: -1 if request is satisfied, else ID of pending request.

Exceptions:

"Invalid number of requested entities"

Cause: "nEntities" < 0.

Effect: Simulation terminates.

- **ReleaseSingle**

Objective: This method releases a single entity back to an EntityCounter. It increments the count of entities within the EntityCounter by 1.

Parameters: None.

Return Value: None.

Exceptions: None.

- **ReleaseMultiple**

Objective: This method releases "nEntities" back to an EntityCounter. It increments the count of entities within the EntityCounter by "nEntities".

Parameters:

nEntities

Mode: Input.

Type: Integer.

Presence: Required.

Function: Specifies the number of entities to be released.

Return Value: None.

Exceptions:

"Invalid number of entities to be released"

Cause: "nEntities" < 0.

Effect: Simulation terminates.

- **ReAssign**

Objective: This method transfers "nEntities" from a collection of EntityCounters to the "destination" EntityCounter. This method behaves similarly to the RequestAvailable method and tries to satisfy the entire request, or whatever part of the request it can satisfy. If the current EntityCounter cannot completely satisfy the request, then it queues the request within itself, and then passes along the request to the next EntityCounter in "sourcePoolList". If a transfer request is partially satisfied by any EntityCounter subsequently, then all EntityCounters within "sourcePoolList" update their pending requests accordingly. If a transfer request is completely satisfied by any EntityCounter subsequently, then all EntityCounters within "sourcePoolList" remove the request from their collection of pending requests.

Parameters:

sourcePoolList

Mode: Input.

Type: Set.

Presence: Required.

Function: Specifies the collection of source EntityCounters.

destination

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
      Mode:      Input.
      Type:      EntityCounter.
      Presence:  Required.
      Function:  Specifies the destination EntityCounter.
   nEntities
      Mode:      Input.
      Type:      Integer.
      Presence:  Required.
      Function:  Specifies the number of entities to be reassigned.
   eventDelegate
      Mode:      Input.
      Type:      Delegate.
      Presence:  Required.
      Function:  Specifies the delegate to be scheduled when transfer
                 is complete.
Return Value:    None.
Exceptions:
   "Invalid source EntityCounters"
      Cause:     Invalid list of source EntityCounters.
      Effect:    Simulation terminates.
   "Invalid destination EntityCounter"
      Cause:     Invalid destination EntityCounter.
      Effect:    Simulation terminates.
   "Invalid number of entities to be reassigned"
      Cause:     "nEntities" < 0.
      Effect:    Simulation terminates.
```

- **ReturnReAssigned**

```
Objective:       This method returns the entities back to the source
                 EntityCounters where they were transferred from.
Parameters:      None.
Return Value:    None.
Exceptions:
   "Invalid source EntityCounters"
      Cause:     Invalid list of source EntityCounters.
      Effect:    Simulation terminates.
```

- **TransferToEntityCounter**

```
Objective:       This method permanently transfers "nEntities" from
                 an EntityCounter to the "destination" EntityCounter.
                 If sufficient entities exist to satisfy the
                 transfer, "nEntities" are immediately transferred to
                 "destination". If insufficient entities currently
                 exist, all existing entities are immediately
                 transferred, and all entities subsequently released
                 back to the EntityCounter, are transferred till the
                 transfer is complete.
Parameters:
   destination
      Mode:      Input.
      Type:      EntityCounter.
      Presence:  Required.
      Function:  Specifies the destination EntityCounter.
   nEntities
      Mode:      Input.
      Type:      Integer.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
      Presence:  Required.
      Function:  Specifies the number of entities to be transferred.
Return Value:    None.
Exceptions:      None.
   "Invalid destination EntityCounter"
      Cause:     Invalid destination EntityCounter.
      Effect:    Simulation terminates.
   "Invalid number of entities to be transferred"
      Cause:     "nEntities" < 0.
      Effect:    Simulation terminates.
```

- **TransferToEntityPool**

```
Objective:       This method permanently transfers "nEntities" from
                 an EntityCounter to the "destination" EntityPool. If
                 sufficient entities exist to satisfy the transfer,
                 "entity" is cloned "nEntities" times and immediately
                 transferred to "destination". If insufficient
                 entities currently exist, all existing entities are
                 immediately cloned and transferred, and all entities
                 released back to the EntityCounter are cloned and
                 transferred till the transfer is complete.
Parameters:
   destination
      Mode:      Input.
      Type:      EntityPool.
      Presence:  Required.
      Function:  Specifies the destination EntityPool.
   nEntities
      Mode:      Input.
      Type:      Integer.
      Presence:  Required.
      Function:  Specifies the number of entities to be transferred.
   entity
      Mode:      Input.
      Type:      Replicable.
      Presence:  Required.
      Function:  Specifies the entity to be cloned during transfer.
Return Value:    None.
Exceptions:      None.
   "Invalid destination EntityPool"
      Cause:     Invalid destination EntityPool.
      Effect:    Simulation terminates.
   "Invalid number of entities to be transferred"
      Cause:     "nEntities" < 0.
      Effect:    Simulation terminates.
   "Invalid entity for cloning"
      Cause:     Invalid entity to be cloned during transfer.
      Effect:    Simulation terminates.
```

- **InterruptRequest**

```
Objective:       This method interrupts a pending request within an
                 EntityCounter. The interrupt method within the
                 Delegate, if defined, is executed.
Parameters:
   requestID
      Mode:      Input.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
      Type:     RequestID.
      Presence: Required.
      Function: Specifies the ID of the pending request.
   Return Value:   None.
   Exceptions:
     "Request to be interrupted not found"
      Cause:    Specified request does not exist within collection
                of pending requests.
      Effect:   Simulation terminates.
```

A.4.8 *EntityPool* Interface

Interface Diagram:

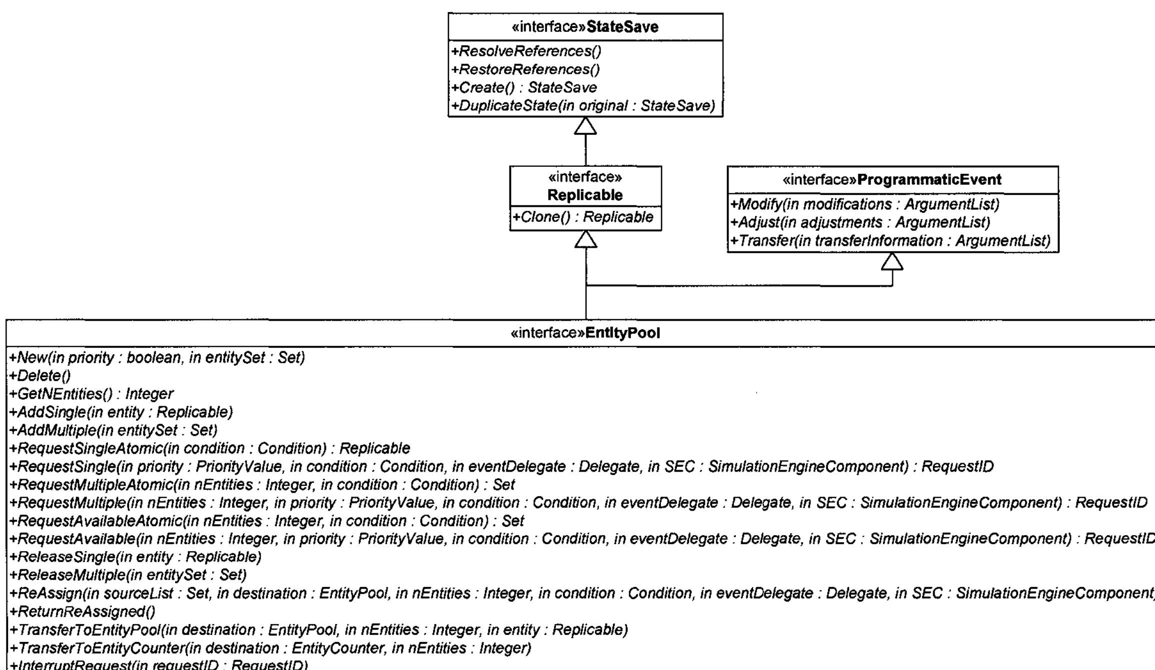


Figure A25. *EntityPool* Interface

Methods:

- **ResolveReferences**

```
Objective:      This method resolves references for dynamically
                initialized attributes within an EntityPool.
Parameters:     None.
Return Value:   None.
Exceptions:     None.
```

- **RestoreReferences**

```
Objective:      This method restores references for dynamically
                initialized attributes within an EntityPool.
Parameters:     None.
Return Value:   None.
Exceptions:     None.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- **Create**

  Objective: This method creates an empty shell of an EntityPool during the state duplication process of a non-terminating state save operation.
  Parameters: None.
  Return Value:
  Type: StateSave.
  Function: Empty shell of EntityPool.
  Exceptions: None.

- **DuplicateState**

  Objective: This method copies the state of "original" to the current EntityPool during the state duplication process of a non-terminating state save operation.
  Parameters:
  original
  Mode: Input.
  Type: StateSave.
  Presence: Required.
  Function: Specifies EntityPool whose state is to be copied.
  Return Value: None.
  Exceptions: None.

- **Clone**

  Objective: This method creates a clone of an EntityPool with the number of entites in the clone equal to the number of entities currently within the EntityPool.
  Parameters: None.
  Return Value:
  Type: Replicable.
  Function: Clone of an EntityPool.
  Exceptions: None.

- **Modify**

  Objective: This method is not defined for an EntityPool.
  Parameters:
  modifications
  Mode: Input.
  Type: ArgumentList.
  Presence: Required.
  Function: Specifies the new parameters.
  Return Value: None.
  Exceptions:
  "Modify method not defined for this interface"
  Cause: Method not defined for an EntityPool.
  Effect: Simulation terminates.

- **Adjust**

  Objective: This method increases or decreases the entities within an EntityPool based on an offset. An entity should be provided when increasing the number of entities, so that it can be cloned and added to the EntityPool. If insufficient entities currently exist to satisfy a decrease in entities, all existing entities are immediately removed, and all entities

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
                 released back to the EntityPool, are removed till
                 the decrease is complete.
Parameters:
   adjustments
      Mode:      Input.
      Type:      ArgumentList.
      Presence:  Required.
      Function:  Specifies the new parameters.
Return Value:    None.
Exceptions:
   "Insufficient arguments for Adjust method"
      Cause:     Not enough arguments within "adjustments".
      Effect:    Simulation terminates.
```

- **Transfer**

```
Objective:       This method invokes TransferToEntityPool or
                 TransferToEntityCounter for an EntityPool, with the
                 appropriate parameters based on
                 "transferInformation".
Parameters:
   transferInformation
      Mode:      Input.
      Type:      ArgumentList.
      Presence:  Required.
      Function:  Specifies the new parameters.
Return Value:    None.
Exceptions:
   "Insufficient arguments for Transfer method"
      Cause:     Not enough arguments within "transferInformation".
      Effect:    Simulation terminates.
```

- **New**

```
Objective:       This method initializes an EntityPool.
Parameters:
   priority
      Mode:      Input.
      Type:      Boolean.
      Presence:  Required.
      Function:  Specifies if unsatisfied requests are queued based
                 on priority within the EntityPool.
   entitySet
      Mode:      Input.
      Type:      Set.
      Presence:  Optional.
      Function:  Specifies the initial set of entities within the
                 EntityPool.
Return Value:    None.
Exceptions:      None.
```

- **Delete**

```
Objective:       This method disposes an EntityPool.
Parameters:      None.
Return Value:    None.
Exceptions:      None.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- **GetNEntities**

```
Objective:       This method returns the current number of entities
                 within an EntityPool.
Parameters:      None.
Return Value:
      Type:      Integer.
      Function:  Current number of entities in the EntityPool.
Exceptions:      None.
```

- **AddSingle**

```
Objective:       This method adds an entity to an EntityPool.
Parameters:
   entity
      Mode:      Input.
      Type:      Entity.
      Presence:  Required.
      Function:  Specifies the entity to be added to the EntityPool.
Return Value:    None.
Exceptions:      None.
```

- **AddMultiple**

```
Objective:       This method adds a collection of entities to an
                 EntityPool.
Parameters:
   entitySet
      Mode:      Input.
      Type:      Set.
      Presence:  Required.
      Function:  Specifies the collection of entities to be added to
                 the EntityPool.
Return Value:    None.
Exceptions:
   "Invalid collection of entities to be added"
      Cause:     Invalid set of entities.
      Effect:    Simulation terminates.
```

- **RequestSingleAtomic**

```
Objective:       This method makes an atomic request for a single
                 entity to an EntityPool. If an entity is available,
                 then it is returned, else a NULL value is returned.
                 If a condition is specified, then an entity that
                 satisfies the condition is returned.
Parameters:
   condition
      Mode:      Input.
      Type:      Condition.
      Presence:  Optional.
      Function:  Specifies the condition to be satisfied.
Return Value:
      Type:      Entity.
      Function:  Entity if available, NULL if not.
Exceptions:      None.
```

- **RequestSingle**

```
Objective:       This method makes a request for a single entity to
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the EntityPool. If an entity is available, then the Delegate is immediately scheduled for execution, with the entity passed as a return argument by reference for the event method ArgumentList within the Delegate. If a condition is specified, then an entity that satisfies the condition is returned. If an entity is not available, then the request is queued and serviced in a FCFS manner. If a priority has been specified, then the request is queued with the appropriate priority relative to other pending requests.

Parameters:

priority

Mode: Input.
Type: Priority Value.
Presence: Optional.
Function: Specifies the priority of the request relative to pending requests.

condition

Mode: Input.
Type: Condition.
Presence: Optional.
Function: Specifies the condition to be satisfied.

eventDelegate

Mode: Input.
Type: Delegate.
Presence: Required.
Function: Specifies the delegate to be scheduled when request is satisfied.

Return Value:

Type: RequestID.
Function: -1 if request is satisfied, else ID of pending request.

Exceptions: None.

- **RequestMultipleAtomic**

Objective: This method makes an atomic request for "nEntities" to an EntityPool. If the requested entities are available, then a Set containing the entities is returned, else a NULL value is returned. If a condition is specified, then entities that satisfy the condition are returned.

Parameters:

nEntities

Mode: Input.
Type: Integer.
Presence: Required.
Function: Specifies the number of requested entities.

condition

Mode: Input.
Type: Condition.
Presence: Optional.
Function: Specifies the condition to be satisfied.

Return Value:

Type: Set.
Function: Set containing the entities if available, NULL if

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
                  not.
Exceptions:
   "Invalid number of requested entities"
      Cause:     "nEntities" < 0.
      Effect:    Simulation terminates.
```

- **RequestMultiple**

```
Objective:        This method makes a request for "nEntities" entities
                  to an EntityPool. If the requested entities are
                  available, then the Delegate is immediately
                  scheduled for execution, with the Set containing the
                  entities passed as a return argument by reference
                  for the event method ArgumentList within the
                  Delegate. If a condition is specified, then entities
                  that satisfy the condition are returned. If the
                  requested entities are not available, then the
                  request is queued and serviced in a FCFS manner. If
                  a priority has been specified, then the request is
                  queued with the appropriate priority relative to
                  other pending requests.
Parameters:
   nEntities
      Mode:      Input.
      Type:      Integer.
      Presence:  Required.
      Function:  Specifies the number of requested entities.
   priority
      Mode:      Input.
      Type:      Priority Value.
      Presence:  Optional.
      Function:  Specifies the priority of the request relative to
                 pending requests.
   condition
      Mode:      Input.
      Type:      Condition.
      Presence:  Optional.
      Function:  Specifies the condition to be satisfied.
   eventDelegate
      Mode:      Input.
      Type:      Delegate.
      Presence:  Required.
      Function:  Specifies the delegate to be scheduled when request
                 is satisfied.
Return Value:
      Type:      RequestID.
      Function:  -1 if request is satisfied, else ID of pending
                 request.
Exceptions:
   "Invalid number of requested entities"
      Cause:     "nEntities" < 0.
      Effect:    Simulation terminates.
```

- **RequestAvailableAtomic**

```
Objective:        This method makes an atomic request for "nEntities"
                  to an EntityPool. If no entities are available, then
                  a NULL value is returned, else a Set containing
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

entities equal to the minimum value of "nEntities" and currently available entities in the EntityPool is returned. If a condition is specified, then entities that satisfy the condition are returned.

Parameters:

nEntities

Mode: Input.
Type: Integer.
Presence: Required.
Function: Specifies the number of requested entities.

condition

Mode: Input.
Type: Condition.
Presence: Optional.
Function: Specifies the condition to be satisfied.

Return Value:

Type: Set.
Function: Set containing the available entities, NULL if none available.

Exceptions:

"Invalid number of requested entities"

Cause: "nEntities" < 0.
Effect: Simulation terminates.

- **RequestAvailable**

Objective: This method makes a request for "nEntities" to an EntityPool. If at least 1 entity is available, then the Delegate is immediately scheduled for execution, with the Set containing the entities passed as a return argument by reference for the event method ArgumentList within the Delegate. If a condition is specified, then entities that satisfy the condition are returned. If no entities are available or if the number of available entities is less than "nEntities", then the request is queued and serviced in a FCFS manner. If a priority has been specified, then the request is queued with the appropriate priority relative to other pending requests. The Delegate is scheduled for execution every time new entities become available till the request is completely satisfied.

Parameters:

nEntities

Mode: Input.
Type: Integer.
Presence: Required.
Function: Specifies the number of requested entities.

priority

Mode: Input.
Type: Priority Value.
Presence: Optional.
Function: Specifies the priority of the request relative to pending requests.

condition

Mode: Input.
Type: Condition.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Presence: Optional.
Function: Specifies the condition to be satisfied.

eventDelegate

Mode: Input.
Type: Delegate.
Presence: Required.
Function: Specifies the delegate to be scheduled when request is satisfied.

Return Value:

Type: RequestID.
Function: -1 if request is satisfied, else ID of pending request.

Exceptions:

"Invalid number of requested entities"

Cause: "nEntities" < 0.
Effect: Simulation terminates.

- **ReleaseSingle**

Objective: This method releases a single entity back to an EntityPool.

Parameters:

entity

Mode: Input.
Type: Entity.
Presence: Required.
Function: Specifies the entity to be released to the EntityPool.

Return Value: None.
Exceptions: None.

- **ReleaseMultiple**

Objective: This method releases a collection of entities to an EntityPool.

Parameters:

entitySet

Mode: Input.
Type: Set.
Presence: Required.
Function: Specifies the collection of entities to be released to the EntityPool.

Return Value: None.
Exceptions: None.

- **ReAssign**

Objective: This method transfers "nEntities" from a collection of EntityPools to the "destinationPool" EntityPool. This method behaves similarly to the RequestAvailable method and tries to satisfy the entire request, or whatever part of the request it can satisfy. If the current EntityPool cannot completely satisfy the request, then it queues the request within itself, and then passes along the request to the next EntityPool in "sourcePoolList". If a transfer request is partially satisfied by any EntityPool subsequently, then all EntityPools within "sourcePoolList" update their pending requests

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

accordingly. If a transfer request is completely satisfied by any EntityPool subsequently, then all EntityPools within "sourcePoolList" remove the request from their collection of pending requests. If a condition is specified, then entities that satisfy the condition are returned.

Parameters:

sourcePoolList

Mode: Input.
Type: Set.
Presence: Required.
Function: Specifies the collection of source EntityPools.

destination

Mode: Input.
Type: EntityPool.
Presence: Required.
Function: Specifies the destination EntityPool.

nEntities

Mode: Input.
Type: Integer.
Presence: Required.
Function: Specifies the number of entities to be transferred.

condition

Mode: Input.
Type: Condition.
Presence: Optional.
Function: Specifies the condition to be satisfied.

eventDelegate

Mode: Input.
Type: Delegate.
Presence: Required.
Function: Specifies the delegate to be scheduled when transfer is complete.

Return Value: None.

Exceptions:

"Invalid source EntityPools"
Cause: Invalid list of source EntityPools.
Effect: Simulation terminates.

"Invalid destination EntityPool"
Cause: Invalid destination EntityPool.
Effect: Simulation terminates.

"Invalid number of entities to be transferred"
Cause: "nEntities" < 0.
Effect: Simulation terminates.

- **ReturnReAssigned**

Objective: This method returns the entities back to the source EntityPools where they were transferred from.
Parameters: None.
Return Value: None.
Exceptions:

"Invalid source EntityPools"
Cause: Invalid list of source EntityPools.
Effect: Simulation terminates.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- **TransferToEntityPool**

Objective: This method permanently transfers "nEntities" from an EntityPool to the "destination" EntityPool. If sufficient entities exist to satisfy the transfer, "entity" is cloned "nEntities" times and immediately transferred to "destination". If insufficient entities currently exist, all existing entities are immediately cloned and transferred, and all entities released back to the EntityPool are cloned and transferred till the transfer is complete.

Parameters:

destination
Mode: Input.
Type: EntityPool.
Presence: Required.
Function: Specifies the destination EntityPool.

nEntities
Mode: Input.
Type: Integer.
Presence: Required.
Function: Specifies the number of entities to be transferred.

entity
Mode: Input.
Type: Replicable.
Presence: Required.
Function: Specifies the entity to be cloned during transfer.

Return Value: None.
Exceptions: None.

"Invalid destination EntityPool"
Cause: Invalid destination EntityPool.
Effect: Simulation terminates.

"Invalid number of entities to be transferred"
Cause: "nEntities" < 0.
Effect: Simulation terminates.

"Invalid entity for cloning"
Cause: Invalid entity to be cloned during transfer.
Effect: Simulation terminates.

- **TransferToEntityCounter**

Objective: This method permanently transfers "nEntities" from an EntityPool to the "destination" EntityCounter. If sufficient entities exist to satisfy the transfer, "nEntities" are immediately transferred to "destination". If insufficient entities currently exist, all existing entities are immediately transferred, and all entities released back to the EntityPool, are transferred till the transfer is complete.

Parameters:

destination
Mode: Input.
Type: EntityCounter.
Presence: Required.
Function: Specifies the destination EntityCounter.

nEntities
Mode: Input.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Type: Integer.
Presence: Required.
Function: Specifies the number of entities to be transferred.
Return Value: None.
Exceptions: None.
"Invalid destination EntityCounter"
Cause: Invalid destination EntityCounter.
Effect: Simulation terminates.
"Invalid number of entities to be transferred"
Cause: "nEntities" < 0.
Effect: Simulation terminates.

- **InterruptRequest**

Objective: This method interrupts a pending request within an EntityPool. The interrupt method within the Delegate, if defined, is executed.
Parameters:
requestID
Mode: Input.
Type: RequestID.
Presence: Required.
Function: Specifies the ID of the pending request.
Return Value: None.
Exceptions:
"Request to be interrupted not found"
Cause: Specified request does not exist within collection of pending requests.
Effect: Simulation terminates.

B. DIESEL Distributed Interface Specification

B.1 *DistributedSimulationExecutive* Interface

Interface Diagram:

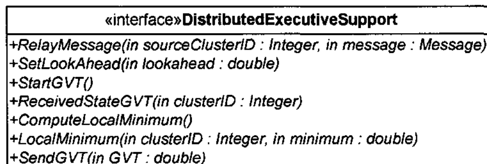


Figure B1. *DistributedSimulationExecutive* Interface

Methods:

- **StartDistributedSimulation**

Objective: This method initializes DIESEL for a distributed simulation with the specified number of SimulationClusters.
Parameters:
nClusters
Mode: Input.
Type: Integer.
Presence: Required.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Function: Specifies the number of SimulationClusters in a distributed simulation.
Return Value: None.
Exceptions: None.

- **GetNSimulationClusters**

Objective: This method returns the number of SimulationClusters in a distributed simulation.
Parameters: None.
Return Value:
Type: Integer.
Function: Number of SimulationClusters in a distributed simulation.
Exceptions: None.

- **CreateSimulationCluster**

Objective: This method creates a SimulationCluster and registers itself with the simulation executive.
Parameters:
clusterID
Mode: Input.
Type: Integer.
Presence: Required.
Function: Specifies the ID of SimulationCluster to be created.
synchronizationMethod
Mode: Input.
Type: Synchronization Algorithm.
Presence: Required.
Function: Specifies the synchronization algorithm used.
Return Value:
Type: SimulationCluster.
Function: Created SimulationCluster.
Exceptions: None.

- **AddSimulationCluster**

Objective: This method registers a SimulationCluster with the simulation executive.
Parameters:
cluster
Mode: Input.
Type: SimulationCluster.
Presence: Required.
Function: Specifies the SimulationCluster to be registered with the simulation executive.
Return Value: None.
Exceptions: None.

- **GetSimulationCluster**

Objective: This method returns access to a SimulationCluster registered with the simulation executive.
Parameters:
clusterID
Mode: Input.
Type: Integer.
Presence: Required.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
      Function:  Specifies the ID of required SimulationCluster.
Return Value:
      Type:      SimulationCluster.
      Function:  Required SimulationCluster.
Exceptions:      None.
```

- **CleanUpDistributedSimulation**

```
Objective:       This method cleans up DIESEL after a distributed
                 simulation.
Parameters:      None.
Return Value:    None.
Exceptions:      None.
```

- **EventsExist**

```
Objective:       This method ascertains if any delegates exist on any
                 SimulationClusters in the distributed simulation and
                 returns the minimum execution time of all delegates
                 if they do exist.
Parameters:
   minimumEventTime
      Mode:      Output.
      Type:      Double.
      Presence:  Required.
      Function:  Specifies the minimum event execution time of
                 delegates within all SimulationClusters.
Return Value:
      Type:      Boolean.
      Function:  TRUE if events exist, ELSE FALSE.
Exceptions:      None.
```

- **ExecuteDistributedSimulation**

```
Objective:       This method executes a distributed simulation with
                 the specified synchronization algorithm.
Parameters:      None.
Return Value:    None.
Exceptions:      None.
```

- **GetUniqueReference**

```
Objective:       This method returns a unique reference for an
                 application object to be used in a distributed
                 simulation.
Parameters:      None.
Return Value:
      Type:      Unique Reference.
      Function:  Unique reference for an application object.
Exceptions:      None.
```

- **RegisterObject**

```
Objective:       This method registers an application object with the
                 simulation executive.
Parameters:
   reference
      Mode:      Input.
      Type:      Unique Reference.
      Presence:  Required.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Function: Specifies the unique reference of the application object to be registered.

clusterID

Mode: Input.
Type: Integer.
Presence: Required.
Function: Specifies the ID of the SimulationCluster that the application object belongs to.

object

Mode: Input.
Type: Object.
Presence: Required.
Function: Specifies the application object to be registered.

SEC

Mode: Input.
Type: SimulationEngineComponent.
Presence: Required.
Function: Specifies associated SimulationEngineComponent if the object is a SimObject.

Return Value: None.
Exceptions: None.

- **RemoveObject**

Objective: This method de-registers an application object with the simulation executive.

Parameters:

reference

Mode: Input.
Type: Unique Reference.
Presence: Required.
Function: Specifies the unique reference of the application object to be de-registered.

Return Value: None.
Exceptions: None.

B.2 Modified *SimulationEngineComponent* Interface

Interface Diagram:

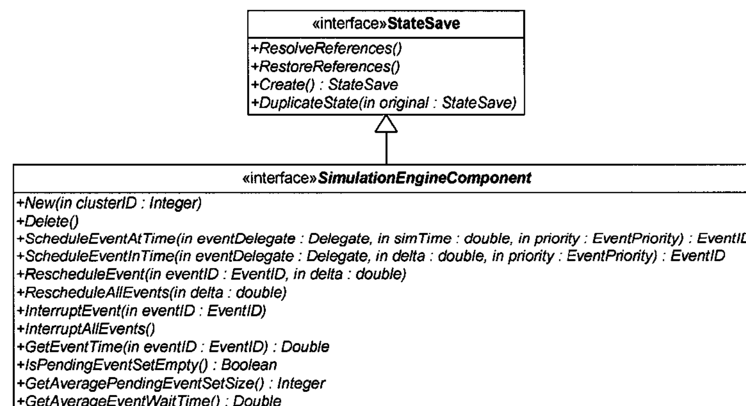


Figure B2. Modified *SimulationEngineComponent* Interface

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Methods:

- **New**

  Objective: This method initializes a SimulationEngineComponent with the ID of the SimulationCluster it belongs to and registers itself with the appropriate SimulationCluster.

  Parameters:

  clusterID

  Mode: Input.

  Type: Integer.

  Presence: Required.

  Function: Specifies the SimulationCluster on which the SimulationEngineComponent has been initialized.

  Return Value: None.

  Exceptions: None.

B.3 *SimulationCluster* Interface

Interface Diagram:

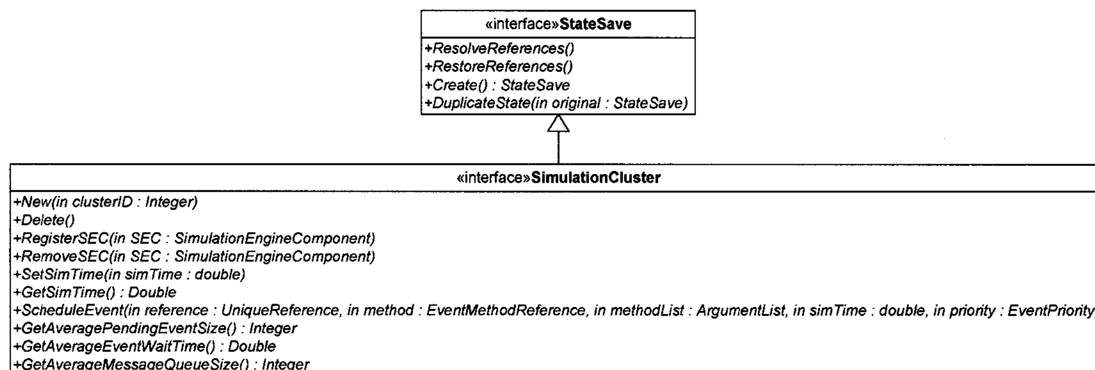


Figure B3. *SimulationCluster* Interface

Methods:

- **ResolveReferences**

  Objective: This method resolves references for dynamically initialized attributes within a SimulationCluster.

  Parameters: None.

  Return Value: None.

  Exceptions: None.

- **RestoreReferences**

  Objective: This method restores references for dynamically initialized attributes within a SimulationCluster.

  Parameters: None.

  Return Value: None.

  Exceptions: None.

- **Create**

  Objective: This method creates an empty shell of a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
                    SimulationCluster during the state duplication
                    process of a non-terminating state save operation.
Parameters:         None.
Return Value:
      Type:         StateSave.
      Function:     Empty shell of SimulationCluster.
Exceptions:         None.
```

- **DuplicateState**

```
Objective:          This method copies the state of "original" to the
                    current SimulationCluster during the state
                    duplication process of a non-terminating state save
                    operation.
Parameters:
   original
      Mode:         Input.
      Type:         StateSave.
      Presence:     Required.
      Function:     Specifies SimulationCluster whose state is to be
                    copied.
Return Value:       None.
Exceptions:         None.
```

- **New**

```
Objective:          This method initializes a SimulationCluster.
Parameters:
   clusterID
      Mode:         Input.
      Type:         Integer.
      Presence:     Required.
      Function:     Specifies ID of SimulationCluster.
Return Value:       None.
Exceptions:         None.
```

- **Delete**

```
Objective:          This method disposes a SimulationCluster.
Parameters:         None.
Return Value:       None.
Exceptions:         None.
```

- **RegisterSEC**

```
Objective:          This method registers a SimulationEngineComponent
                    with a SimulationCluster.
Parameters:
   SEC
      Mode:         Input.
      Type:         SimulationEngineComponent.
      Presence:     Required.
      Function:     Specifies SimulationEngineComponent to be registered
                    with the SimulationCluster.
Return Value:       None.
Exceptions:         None.
```

- **RemoveSEC**

```
Objective:          This method de-registers a SimulationEngineComponent
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

with a SimulationCluster.

Parameters:

SEC

Mode: Input.
Type: SimulationEngineComponent.
Presence: Required.
Function: Specifies SimulationEngineComponent to be de-registered with the SimulationCluster.

Return Value: None.
Exceptions: None.

- **SetSimTime**

Objective: This method sets the current simulation time for a SimulationCluster.

Parameters:

time

Mode: Input.
Type: Double.
Presence: Required.
Function: Specifies the current simulation time.

Return Value: None.
Exceptions: None.

- **GetSimTime**

Objective: This method returns the current simulation time within a SimulationCluster.

Parameters: None.

Return Value:

Type: Double.
Function: Current simulation time.

Exceptions: None.

- **ScheduleEvent**

Objective: This method schedules an event on a SimObject if the SimObject exists within a SimulationCluster. If it does not, then the SimulationCluster asks the simulation executive to relay a message with the required information to the appropriate SimulationCluster.

Parameters:

reference

Mode: Input.
Type: Unique reference to an application object.
Presence: Required.
Function: Specifies the unique reference to an application object.

method

Mode: Input.
Type: EventMethodReference.
Presence: Required.
Function: Specifies the reference to the event method.

methodList

Mode: Input.
Type: ArgumentList.
Presence: Required.
Function: Specifies the list of arguments to be used when the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

event method is executed.

simTime

Mode: Input.
Type: Double.
Presence: Required.
Function: Specifies the scheduled execution time of the event method.

priority

Mode: Input.
Type: Event Priority.
Presence: Optional.
Function: Specifies the priority of the event method relative to other delegates scheduled at the same simulation time.

Return Value: None.
Exceptions: None.

- **GetAveragePendingEventSize**

Objective: This method returns the average size of the pending event set for all SimulationEngineComponents within a SimulationCluster.
Parameters: None.
Return Value:
Type: Integer.
Function: Average size of pending event set.
Exceptions: None.

- **GetAverageEventWaitTime**

Objective: This method returns the average waiting time for an event before it is executed within all SimulationEngineComponents in a SimulationCluster.
Parameters: None.
Return Value:
Type: Double.
Function: Average event wait time.
Exceptions: None.

- **GetAverageMessageQueueSize**

Objective: This method returns the average size of the input queue in a SimulationCluster.
Parameters: None.
Return Value:
Type: Integer.
Function: Average size of input queue.
Exceptions: None.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

VITA

DEGREES:

Doctor of Philosophy (Electrical and Computer Engineering), Old Dominion University, Norfolk, VA, May 2007.

Master of Science (Computer Engineering), Old Dominion University, Norfolk, VA, December 2002.

Bachelor of Engineering (Electronics Engineering), Sardar Patel College of Engineering, Mumbai University, Mumbai, India, June 2000.

PART TIME EMPLOYMENT:

Software Engineering Intern in the Cargo Simulation and Logistics laboratory at **MYMIC LLC**, Portsmouth, VA. (January 2006 – Present)

Research Assistant at the **Virginia Modeling Analysis and Simulation Center (VMASC)**, Suffolk, VA. (August 2000 – December 2005)

PUBLICATIONS:

Books

- Leathrum J.F., R.R. Mielke, S. Mazumdar, R. Mathew, Y. Manepalli, V. Pillai, and R.N. Malladi. "A Simulation Architecture to Support Intratheater Sealift Operations." *Defense Transportation: Algorithms, Models and Applications for the 21st Century*, Elsevier Science Ltd, 2004.

Journals

- Mathew R., J.F. Leathrum, S. Mazumdar, T. Frith, and J. Joines. "An Object-Oriented Architecture for the Simulation of Networks of Cargo Terminal Operations." *Journal of Defense Modeling and Simulation (JDMS)* 2, no. 2 (April 2005): 101–116.
- Leathrum J.F., R.R. Mielke, S. Mazumdar, R. Mathew, Y. Manepalli, V. Pillai, and R.N. Malladi. "A Simulation Architecture to Support Intratheater Sealift Operations." *Mathematical and Computer Modelling* 39, no. 6-8 (May 2004): 817–838.

Conferences

- Mazumdar S., R. Mathew, and J.F. Leathrum. "A Strategy for Distributing Simulations for Statistical Analysis." In *Proc. Summer Computer Simulation Conference (SCSC 2004)*, 294–299, 2004.
- Leathrum J.F., R.R. Mielke, T. Frith, and R. Mathew. "Modeling New Technologies in a Joint Logistics Over The Shore (JLOTS) Operation." In *Proc. Summer Computer Simulation Conference (SCSC 2002)*, 165–169, 2002.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.