# SPEEDES User's Guide

The Synchronous Parallel Environment for Emulation and Discrete-Event Simulation

30 April 2003
Document Number: S024
Revision Number: 4

Prepared by:
Metron, Inc.
512 Via De La Valle Suite 301
Solana Beach, CA 92075-2715
http://www.speedes.com

Prepared For:
The Joint National Integration Center
730 Irwin Avenue
Schriever AFB, CO 80912-7300

# Contents

# List of Tables

# List of Figures

# List of Examples

# Part I

# Overview

# Chapter 1

# Introduction

## 1.1 Purpose

This document provides detailed descriptions on the use of Synchronous Parallel Environment for Emulation and Discrete-Event Simulation (SPEEDES). In general, each new term, function, Application Program Interface (API), etc. is introduced with detailed follow-ups of examples in order to assist the user in the overall understanding of SPEEDES.

## 1.2 System Overview

SPEEDES is a general purpose discrete-event distributed simulation framework that can be used to simulate a wide variety of situations. SPEEDES has been successfully used for missile defense and National Airspace simulations, as well as a number of other models. SPEEDES will support many different hardware architectures ranging from the fastest massively parallel machines utilizing lighting-fast shared memory, to distributed networks of fast workstations.

A discrete-event simulation differs from a time-stepped simulation in that only significant events are simulated. If an airplane flight were simulated from takeoff to landing in a time-stepped simulation, the position of the plane would be updated at every cycle of the simulation. In a discrete-event simulation, there would be an event corresponding to the takeoff of the plane and an event corresponding to the landing of the plane. In between, it is assumed the plane flew its route. This results in much lower Central Processing Unit (CPU) use and much more efficient use of CPU resources. Of course, the plane's position can be queried at any point during the flight through further discrete events, and events can be canceled if other objects affect the timeline of the plane.

SPEEDES runs discrete-event simulations in parallel by distributing the simulation objects to different processors. Each processor is then responsible for executing all events on the simulation objects that it was assigned. Each event may affect objects on other processors and SPEEDES will send the event notice to that processor and add it to the queue of pending events on the appropriate simulation object.

SPEEDES provides several methods of running that include conservative schemes (guarantee no events will be erroneously processed) and optimistic schemes (may need to undo or roll back events). These strategies can be used to aid the modeler in taking the greatest advantage of the hardware that is available. The optimistic strategies include a fully optimistic processing, but risk-free message sending strategy, a strategy with maximum message sending risk, and a strategy with variable message sending risk.

Naturally, the conservative schemes include an optimized sequential algorithm with most of the parallel overhead removed.

In order to support the ability to roll back an event, SPEEDES uses various incremental state saving techniques that are extremely efficient. SPEEDES also supports the ability to roll an event forward, if the state that the event depends upon has not changed, without requiring large amounts of memory overhead. This is known as lazy event re-evaluation.

These techniques have been encapsulated into classes that imitate standard C++ types and function calls. For example, the `RB_int` behaves similar to a normal integer and the `RB_free` function behaves similar to the system call `free`, but all of the operations are done in a rollbackable fashion. What this means is that any change to the simulation's state that is done through a rollbackable variable or function call will be undone if the event that caused the change is rolled back. If it turns out that the rollbackable operations are not extensive enough, it is also a straightforward operation to extend the set of rollbackable types and functions.

SPEEDES provides interfaces for developing external interactions and federations that can run asynchronously from the SPEEDES simulation. The interface is quite powerful, resulting in the ability to develop very intricate interactions and graphical user interfaces to the simulation over low bandwidth and high latency situations.

These interactions allow users to query, monitor, and command simulation objects from outside the simulation while it is running. An interactive simulation in SPEEDES can be synchronized to the wall clock, if desired, to support real-time simulation applications. In addition to these external interactions, SPEEDES also supports hybrid synchronization for external modules through the use of time barriers that are set up internal to SPEEDES. These time barriers prevent the simulation Global Virtual Time (GVT) from getting ahead of expected response times from the external modules. These barriers are automatically removed when external modules exit the simulation (either intentionally or unintentionally). Using the external module approach, it is possible to run geographically distributed simulations or federations quite efficiently.

The internals of SPEEDES are highly optimized for general purpose simulations. A new data structure for managing the list of pending events called the SPEEDES Qheap has been measured to be more than a factor of two faster than traditional Splay trees (which are commonly thought of as being the fastest general purpose data structures for managing event lists). SPEEDES also uses internal hashing mechanisms for supporting event cancellation and for global name service.

SPEEDES has been designed to run independently on various computer architectures. All of the communications used by the SPEEDES executable are standard architecture independent operations. The details of how the communications are supported over various architectures are contained in separate communications libraries. These libraries are specified when linking an application, thus reducing portability concerns.

SPEEDES is not a general purpose programming language with extensions to support simulations such as Modular Simulation (MODSIM) II, Simscript, Simula, and other simulation programming languages. SPEEDES is an object-oriented distributed discrete-event simulation framework written in C++. While SPEEDES can and has been used for non-simulation applications, it really is a framework for synchronizing the scheduling of events (i.e. actions) that occur in a distributed system. SPEEDES provides a rich modeling framework allowing other simulation scripting languages to be written on top of SPEEDES. For example, the Integrated Modeling and Persistent Object Relations Technology (IMPORT) modeling language has already been hosted on top of SPEEDES and takes advantage of all the parallel and external module features of SPEEDES.

Events in SPEEDES are only allowed to act on a single simulation object (this is required for correct and efficient rollback support). An event may alter the state of its corresponding simulation object (i.e. modify variables contained inside the simulation object) and may schedule future events (scheduling events with a zero time delay is allowed) for any object in the simulation.

Under the covers, the scheduling of an event takes place by filling out an event message, sending it to the object on which the event should be taking place, and then inserting an event of the appropriate type on the object's queue of events. The message contains the data for calling the method on the remote object, as well as an optional variable length buffer for data whose size is not known ahead of time.

SPEEDES uses free lists for all fixed sized messages to improve performance and to avoid memory fragmentation. SPEEDES automatically sends the message to the destination node at the appropriate time, depending on the synchronization protocol selected.

When it is time to process that event, SPEEDES removes the event from the event queue and calls the method(s) defined by that event. Users normally provide the bulk of their event processing code (i.e. performing computations, making state changes, and scheduling other events) in this method. All state changes here must use the rollback queue utility supported in SPEEDES, which saves the changes made to the state of the simulation object in a generalized linked list.

SPEEDES uses events to change the state of simulation objects (i.e. simulation models). As events are processed, simulation time advances. The advantage of SPEEDES over sequential modeling frameworks, is that SPEEDES can distribute the model work load over several or many CPUs. If models can be parallelized, then this will lead to more work being done in the same amount of time, or simulations that take much less time to run.

## 1.3 Document Overview

This document provides an in-depth look at SPEEDES and its usage. Each chapter discusses a feature of SPEEDES which builds on the previous chapters. As subjects are discussed, examples are always presented along with the subject manner so that users can "see" how to use the different features of SPEEDES. This document is broken up into seven parts described below in Sections 1.3.1 through 1.3.7. Each addresses different areas of the SPEEDES framework.

### 1.3.1 Overview

#### 1.3.1.1 Introduction

Chapter 1 gives an overview of the document including, its purpose, a system overview of SPEEDES, document conventions, and a document overview.

#### 1.3.1.2 Quick Start

Chapter 2 provides several examples that demonstrate the basics of how to create a SPEEDES based simulation. These examples can be used to get a small simulation up and running quickly.

### 1.3.2 Simulation Objects

#### 1.3.2.1 Simulation Object Overview

Chapter 3 describes simulation objects in-depth, including their construction, decomposition onto nodes, and handles for finding simulation objects.

#### 1.3.2.2 Rollbackable Built-in Types

Chapter 4 provides additional details on building simulation objects in an optimistic Parallel Discrete-Event Simulation (PDES) environment. It presents many rollbackable classes that can be used along with general guidelines that should be followed.

#### 1.3.2.3 Utilities

Chapter 5 addresses additional features that are useful in simulation object construction. These include other rollbackable functions, how to create new rollbackable functions or objects, as well as parameter file parsing for simulation object configuration.

### 1.3.3 Events

#### 1.3.3.1 Point-to-Point Events

Chapter 6 discusses point-to-point events. This is the most basic form of an event, which is used by simulation objects that need to cause an action to occur on one other simulation object.

#### 1.3.3.2 Event Handlers

Chapter 7 describes standard event handlers, interactions, and interface event handlers. Handler events allow for run-time configuration of responses and also allow for one-to-many events, rather than the one-to-one feature of point-to-point events.

#### 1.3.3.3 The Process Model

Chapter 8 describes the process model. Standard point-to-point events do not allow time to pass, but the process model allows for reentrant events. That is, the event can go to sleep and then wake up either through a timeout or through an interrupt.

### 1.3.4 Object Proxies

#### 1.3.4.1 Using Object Proxies

Chapter 9 provides the basic introduction into object proxies, including some simple examples. This chapter should be read in its entirety, as many inter-related topics are presented.

#### 1.3.4.2 Proxy Attributes

Chapter 10 describes all the possible attributes that can be distributed through the object proxy interface. These include static types as well as dynamic types whose value is determined through a time-based function.

#### 1.3.4.3 Data Distribution Management (DDM)

Chapter 11 discusses the Data Distribution Management (DDM) capabilities available in the SPEEDES framework. Object proxies work well for distributing the information that is published, but when simulations get large, filters are needed. DDM allows users to filter this information so that only what is needed is delivered to subscribers.

### 1.3.5 External Interfaces

#### 1.3.5.1 External Modules

Chapter 12 provides a communication path for entities external to the SPEEDES simulation. This allows external programs to inject or extract data into a SPEEDES simulation without adversely affecting the SPEEDES application.

#### 1.3.5.2 Command-Line Utilities

Chapter 13 discusses built-in SPEEDES utilites that provide users with some rudimentary interactions with the simulation.

### 1.3.6 Advanced Topics

#### 1.3.6.1 Simulation Objects

Chapter 14 discusses two advanced topics on simulation objects. The first is dynamic object creation and the second is components.

#### 1.3.6.2 Autonomous Events

Chapter 15 describes to how take full advantage of the SPEEDES event processing framework. The SPEEDES unified API hides the internal event mechanisms from the user. Autonomous events allow users access to the internal event mechanisms, giving users greater control over their events.

#### 1.3.6.3 Checkpoint/Restart: Using Persistence

Chapter 16 discusses the SPEEDES checkpoint/restart capability.

#### 1.3.6.4 Diagnostic Tools

Chapter 17 describes how to examine the framework and the simulation's performance. This can help the user diagnose correct or incorrect behavior of a discrete-event simulation.

### 1.3.7 Appendix

#### 1.3.7.1 Parallel Discrete-Event Simulation Technical Reference

Appendix A discusses PDES operation and additional information on how the SPEEDES framework implements these ideas.

#### 1.3.7.2 SPEEDES Parameter File Configuration

Appendix C discusses the file `speedes.par`, which is responsible for SPEEDES framework configuration.

#### 1.3.7.3 Acronyms and Abbreviations

Appendix D provides a list of acronyms and abbreviations used within this document.

## 1.4 Referenced Documents

The following documents, although not necessarily referenced herein, guided preparation of this document and its contents. Unless otherwise specified, the current revision shall apply.

**Government**
None.

**Prime Contractor**

| | |
|---|---|
| 12146H601S | METRON SOW for Wargame 2000 |

**Contractor**

| | |
|---|---|
| S025 | SPEEDES API Reference Manual |

**Other**

| | |
|---|---|
| 14882 | ANSI standard for C++ |

Subcontractor Data Requirement List (SDRL) and other documenation can be found at:

Metron Incorporated
514 Via De La Valle, Suite 306
Solana Beach, CA 92075-2715

## 1.5 Document Conventions

This document often uses `Courier` font in order to highlight certain ideas or concepts. In general, `Courier` is used whenever a SPEEDES class, C++ name, or a UNIX name is used. However, we avoid using constant SPEEDES lingo when an English term would suffice. This helps to explain how to use SPEEDES without overly using SPEEDES classes as English terms.

For example, when decomposition is discussed, and the actual parameters used as input to the macros are expressed, then these arguments (`BLOCK` or `SCATTER`) are placed in `Courier`. However, when we are discussing decomposition types and how they worked then we used block or scatter in the standard font.

As another example, consider proxy attributes. If we are discussing how to get a `POLY_1_MOTION` item from the `SpFreeDynAttributes` class and adding it to the attribute `DYNAMIC_POSITION_AT-TRIBUTE`, then we used `Courier`. However, if we are discussing pulling dynamic items off of the dynamic free list and adding the item to the dynamic position attribute, then we do not use `Courier`.

Also, we did not use `Courier` in most tables, headings or captions. For example, instead of using INT_ATTRIBUTE we use Integer Attribute which should lead to easier readibility of this document.

Finally, as you read the document you will notice that some examples have are surrounded by a box while others ar not. Any code that is part of a larger example (i.e. several .C and .H files) is enclosed in a box with a caption. This is the way most examples are presented. Occasionally, only short code segments are presented. In these cases if the code starts to approach a half a page in length, then these are also enclosed in a box with an appropriate caption.

# Chapter 2

# Quick Start

## 2.1 Simulation Objects

The SPEEDES framework is a collection of C++ classes and an API, which allows users to build a simulation and run that simulation on one or more processors. The SPEEDES API is a series of built-in C++ macros and functions (some of the functions are automatically generated by calls to macros), which allow users to use the full functionality of SPEEDES. This chapter will lead you through a few simple examples to show you the basic principles of SPEEDES. The examples will also show you how to write, compile, and execute a SPEEDES application.

The fundamental building block of a SPEEDES simulation is a simulation object, which represents an object in your simulation on which events are to be scheduled. A simulation object may be a moving physical object, such as a battleship in a war game simulation, or a less obvious object, such as a bus schedule in a traffic simulation. In any case, the important distinguishing characteristic of a simulation object is that events are scheduled on it. In the case of a battleship, a typical event may be the event of firing one of its guns, or the event of assessing damage sustained after being affected by a “fire” event from an enemy submarine object. In the case of a bus schedule, an event may be a departure or the institution of a delay program.

This section explains some simple concepts of SPEEDES, including an introduction to simulation objects and different types of events. Let us proceed with a simple Hello World application.

In order to simulate a computer printing “Hello World” to its screen (in this trivial case we will be doing, not merely simulating), we are faced with decisions. Do we want a single simulation object and an event on it which does the printing? Perhaps we would rather define two types of simulation objects, one containing an event which prints “Hello” and schedules an event on the second object, which prints “World”. Or perhaps a third choice would be a single type of object, on which two event types are defined. We will explore each case individually.

The first step in building our Hello World example is to create our simulation object. All SPEEDES simulation objects inherit from the base class, `SpSimObj`. This class contains a virtual method called `Init`, which SPEEDES calls once all simulation objects are created. In general, this class should be used to initialize all simulation objects. The easiest Hello World implementation is to have the simulation object’s `Init` method print out the string “Hello World”. This method can be implemented without creating any events. Let us take this path and define a single type of simulation object, called `S_HelloWorld` (by SPEEDES convention, the names of simulation objects start with “`S_`”), which contains no events.

Examples 2.1 and 2.2 show the definition and implementation files, for a simple Hello World program. The simulation object is defined by calling the built-in SPEEDES macro, `DEFINE_SIMOBJ`, which tells SPEEDES how many simulation objects of type, `S_HelloWorld`, to create and how they are to be distributed across the different nodes. Notice that method, `Init`, is printing out "Hello World".

```
// S_HelloWorld.H
#ifndef S_HelloWorld_H
#define S_HelloWorld_H

#include "SpSimObj.H"
#include "SpDefineSimObj.H"

/********************************************************************
 * The S_HelloWorld simulation object is represented by a C++ class
 * of the same name.  Here, the only two methods are the constructor
 * and the virtual method Init.  Note that the class inherits from
 * the SPEEDES class SpSimObj.
 ********************************************************************/
class S_HelloWorld : public SpSimObj {
  public:
    S_HelloWorld() {}
    virtual void Init();
};

/********************************************************************
 * Define the simulation object S_HelloWorld using the
 * DEFINE_SIMOBJ macro, whose arguments are (in order):
 * 1. The name of the simulation object (identical to the class
 *     name)
 * 2. The number of objects of this type to be created
 * 3. The decomposition algorithm (SCATTER or BLOCK)
 * The DEFINE_SIMOBJ macro is defined in the SPEEDES header file
 * SpDefineSimObj.H, included above.
 ********************************************************************/
DEFINE_SIMOBJ(S_HelloWorld, 1, SCATTER);
#endif
```

Example 2.1: Hello World Definition (No Events)

```
// S_HelloWorld.C
#include "S_HelloWorld.H"

/********************************************************************
 * A simulation is typically set in motion by the Init function for
 * one or more of the simulation objects.
 ********************************************************************/
void S_HelloWorld::Init() {
  cout << "Hello World" << endl;
}
```

Example 2.2: Hello World Implementation (No Events)

The last item necessary to make the simulation complete is to create `main`. Built-in macros are used to "plug in" simulation objects and their respective events into the SPEEDES framework. Since there are no events, only the simulation object needs to be plugged in using macro, `PLUG_IN_SIMOBJ`, as shown in Example 2.3. The last item in `main` is always the call to the built-in function, `ExecuteSpeedes`.

```
// Main.C
/******************************************************************
 * The PLUG_IN_SIMOBJ macro, called from main, is defined in the
 * header file, SpMainPlugIn.H.
 ******************************************************************/
#include "SpMainPlugIn.H"
#include "S_HelloWorld.H"

/*******************************************************************
 * The main calling program calls the SPEEDES built-in macro,
 * PLUG_IN_SIMOBJ, once for each simulation object type, to inform
 * the simulation of the existence of each of the simulation
 * objects.  This macro is defined in the SPEEDES header file,
 * SpMainPlugIn.H, included above. The call to the built-in
 * function, ExecuteSpeedes, starts the simulation.
 *******************************************************************/
int main (int argc, char** argv) {
  PLUG_IN_SIMOBJ(S_HelloWorld);
  ExecuteSpeedes(argc, argv);
}
```

Example 2.3: Hello World Main (No Events)

In addition to the mentioned files, you can create a file named `speedes.par`, in which various run-time parameters are set. In the absence of such a file, these parameters default to reasonable values. One such parameter the user might find useful is `tend`, which indicates the simulation end time in seconds. The `speedes.par` file should be placed in the current working directory. Appendix C discusses this file in detail.

Here are explicit instructions for compiling and running the simulation using the GNU's Not Unix (GNU) g++ compiler under Linux.

1. Install SPEEDES. Installation instructions are provided in the file, `README`, which is included in the SPEEDES distribution. All of SPEEDES should reside under a directory called, for example, `$HOME/speedes`.
2. Set environment variables, `SPEEDES_INCLUDES` to `$HOME/speedes/include` and `SPEEDES_LIBS` to `$HOME/speedes/lib/ArchitectureDirs/Linux_i686`. This assumes that SPEEDES has been installed in directory `$HOME/speedes`.
3. In any convenient directory, create files `S_HelloWorld.H`, `S_HelloWorld.C`, and `Main.C`. The examples described within the guide can be downloaded from http://www.speedes.com.
4. Compile, link, and execute the example by typing:

```
% g++ -o HelloWorld -I$SPEEDES_INCLUDES S_HelloWorld.C Main.C \
    $SPEEDES_LIBS/libSpEngine.so                              \
    $SPEEDES_LIBS/libSpShMemTCP.so                            \
    $SPEEDES_LIBS/libSpUtil.so

% ./HelloWorld
```

The Hello World simulation should print several statistics to the screen and finally print “Hello World”.

A powerful feature of SPEEDES is its ability to distribute the processing of simulation objects onto several CPUs, or even several computers, possibly separated by large distances and connected by the

Internet. SPEEDES accomplishes this by forking off the user-specified number of processes for which the simulation is to execute on. Each process is called a node. When processing on one node, SPEEDES automatically runs in sequential mode and eliminates parallel-processing overhead, since no parallelism is involved. The Hello World example contains one object running on one node. Let us change our example to run on multiple nodes with multiple objects.

Edit file `S_HelloWorld.H` and change the line, `DEFINE_SIMOBJ(S_HelloWorld, 1, SCATTER)` to `DEFINE_SIMOBJ(S_HelloWorld, 2, SCATTER)`. After recompilation, the program can be restarted using 2 as a command line argument (this specifies that the program is to execute on two nodes). The result of the simulation will be two occurrences of the string "Hello World" being output to the terminal.

You should now understand how the virtual method `Init` works. It executes after all the simulation objects have been constructed. Let us rewrite the Hello World program such that the string "Hello World" is printed out by an event on the `S_HelloWorld` simulation object. The `S_HelloWorld` simulation object will contain an event called `HelloWorldEvent`, implemented as a method on the `S_HelloWorld` class. The choice of event name is any valid string name. This event is scheduled to execute at at time, $t = 0.0$, by the `Init` method and its only purpose is to print "Hello World".

The code shown in Examples 2.4 and 2.5 show the definition and implementation files for a simple event-driven Hello World simulation.

```
// S_HelloWorld.H
#ifndef S_HelloWorld_H
#define S_HelloWorld_H

#include "SpSimObj.H"
#include "SpDefineSimObj.H"
#include "SpDefineEvent.H"

/********************************************************************
 * Here, a method is declared to represent an event called
 * HelloWorldEvent.
 ********************************************************************/
class S_HelloWorld : public SpSimObj {
  public:
    S_HelloWorld() {}
    virtual void Init();
    void         HelloWorldEvent();
};

DEFINE_SIMOBJ(S_HelloWorld, 1, SCATTER);

/********************************************************************
 * A HelloWorldEvent event, which takes no arguments, is defined on
 * the S_HelloWorld simulation object, and must be declared using
 * the DEFINE_SIMOBJ_EVENT_0_ARG macro (the "0" indicates that
 * the event, represented by a method, takes no arguments). The
 * arguments to DEFINE_SIMOBJ_EVENT_0_ARG are (in order):
 * 1. The name of the event.
 * 2. The name of the simulation object on which the event is
 *    defined.
 * 3. The name of the method which defines the action of the event.
 * The macro, DEFINE_SIMOBJ_EVENT_0_ARG, is defined in
 * SpDefineEvent.H and creates the global function,
 * SCHEDULE_HelloWorldEvent.
 *******************************************************************/
DEFINE_SIMOBJ_EVENT_0_ARG(HelloWorldEvent, S_HelloWorld,
                          HelloWorldEvent);
#endif
```

Example 2.4: Hello World Definition File (With Events)

```
// S_HelloWorld.C
#include "S_HelloWorld.H"

/*******************************************************************
 * A call to the function, SpGetObjHandle, with no arguments
 * instructs SPEEDES to retrieve the object handle, which is a
 * unique identifier, of the current simulation object.  A call to
 * the built-in function, SpGetSimObjKindId, returns SimObjKindId, an
 * integer, indicating which of the two S_HelloWorld objects is
 * currently being initialized (0 for the 1st, 1 for the 2nd).
 *******************************************************************/
void S_HelloWorld::Init() {
  if (SpGetSimObjKindId() == 0) {
    SCHEDULE_HelloWorldEvent(0.0, SpGetObjHandle());
  }
}

void S_HelloWorld::HelloWorldEvent() {
  cout << "Hello World" << endl;
}
```

Example 2.5: Hello World Implementation (With Events)

Note the lines:

```
DEFINE_SIMOBJ_EVENT_0_ARG(HelloWorldEvent, S_HelloWorld,
                          HelloWorldEvent)
```

This macro turns method, `HelloWorldEvent`, into a SPEEDES event called `HelloWorldEvent` and creates global function, `SCHEDULE_HelloWorldEvent`, used to schedule this event. The event is implemented by writing the zero argument method, `HelloWorldEvent`, on the `S_HelloWorld` class.

Function, `SCHEDULE_HelloWorldEvent`, is used to schedule event, `HelloWorldEvent`, at time, $t = 0.0$, on the current `S_HelloWorld` object. The global function, `SpGetObjHandle`, called with no arguments, returns the object handle of the current object.

Example 2.6 shows the code necessary to create the `main` for this Hello World program. This `main` is identical the previous `main` shown in Example 2.3, except that macro, `PLUG_IN_EVENT(HelloWorldEvent)`, has been added. This macro plugs the event into the SPEEDES framework.

```
// Main.C
#include "SpMainPlugIn.H"
#include "S_HelloWorld.H"

/*******************************************************************
 * The built-in macro, PLUG_IN_EVENT, informs SPEEDES as to the
 * existence of an event on a simulation object. Its only argument
 * is the name of the event.  The macro is defined in the SPEEDES
 * header file, SpMainPlugIn.H, included above.
 ******************************************************************/
int main (int argc, char** argv) {
  PLUG_IN_SIMOBJ(S_HelloWorld);
  PLUG_IN_EVENT(HelloWorldEvent);
  ExecuteSpeedes(argc, argv);
}
```

Example 2.6: Hello World Main (With Events)

As a variation of the last version on the Hello World program, we could instantiate two `S_HelloWorld` objects, making the first object's virtual method `Init` schedule a `HelloWorldEvent` event on itself at time, $t = 0.0$, and the second object's `Init` method do nothing. The `HelloWorldEvent` event does the following:

1. If scheduled on the first object, it prints “Hello” and schedules a `HelloWorldEvent` event on the second object for time, $t = 1.0$.
2. If scheduled on the second object, it simply prints “World”.

Examples 2.7 and 2.8 show the code necessary to implement this enhancement to the Hello World simulation.

```
// S_HelloWorld.H
#ifndef S_HelloWorld_H
#define S_HelloWorld_H

#include "SpSimObj.H"
#include "SpDefineSimObj.H"
#include "SpDefineEvent.H"

/******************************************************************
 * Here, a method is declared to represent an event called
 * HelloWorldEvent.
 ******************************************************************/
class S_HelloWorld : public SpSimObj {
  public:
    S_HelloWorld() {}
    virtual void Init();
    void         HelloWorldEvent();
};

/******************************************************************
 * The following call to the macro, DEFINE_SIMOBJ, instructs
 * SPEEDES to define two S_HelloWorld simulation objects, and
 * Scatter (card deal) them to available nodes (CPUs or computers).
 ******************************************************************/
DEFINE_SIMOBJ(S_HelloWorld, 2, SCATTER);

/******************************************************************
 * A HelloWorldEvent event, which takes no arguments, is defined on
 * the S_HelloWorld simulation object, and must be declared using the
 * DEFINE_SIMOBJ_EVENT_0_ARG macro (the "0" indicates that the event,
 * represented by a method, takes no arguments). The arguments to
 * DEFINE_SIMOBJ_EVENT_0_ARG are (in order):
 * 1. The name of the event
 * 2. The name of the simulation object on which the event is defined
 * 3. The name of the method which defines the action of the event.
 * The macro, DEFINE_SIMOBJ_EVENT_0_ARG, is defined in
 * SpDefineEvent.H and creates the global function,
 * SCHEDULE_HelloWorldEvent.
 ********************************************************************/
DEFINE_SIMOBJ_EVENT_0_ARG(HelloWorldEvent, S_HelloWorld,
                          HelloWorldEvent);
#endif
```

Example 2.7: Hello World Definition (With Events #2)

```
// S_HelloWorld.C
#include "S_HelloWorld.H"

/********************************************************************
 * A call to the function, SpGetObjHandle, with no arguments
 * instructs SPEEDES to retrieve the object handle of the current
 * simulation object.
 ********************************************************************/
void S_HelloWorld::Init() {
  if (SpGetSimObjKindId() == 0) {
    SCHEDULE_HelloWorldEvent(0.0, SpGetObjHandle());
  }
}

void S_HelloWorld::HelloWorldEvent() {
  if (SpGetSimObjKindId() == 0) {
    cout << "Hello ";
    SCHEDULE_HelloWorldEvent(1.0,
                             SpGetObjHandle("S_HelloWorld_MGR", 1));
  }
  if (SpGetSimObjKindId() == 1) {
    cout << "World" << endl;
  }
}
```

Example 2.8: Hello World Implementation (With Events #2)

The `main` needed to complete this program is the same as the previous `main` (Example 2.6).

There are many other possible scenerios which could be used in our Hello World example, three of which are mentioned below:

1. One `S_Hello` simulation object containing no events, and one `S_World` simulation object containing a `WorldEvent` event. Method, `Init`, in `S_Hello` prints "Hello" and schedules event `WorldEvent` on `S_World` at $t = 1.0$, which prints "World".

2. One `S_Hello` simulation object and one `S_World` simulation object. The `S_Hello` simulation object contains a `HelloEvent` event, which prints "Hello" and schedules `WorldEvent` event on `S_World` simulation object for one time unit later. The `WorldEvent` event prints "World". The process is started by `S_Hello` simulation object's `Init` scheduling a `HelloEvent` event at time, $t = 0.0$.

3. Two instances of a `S_HelloWorld` object, which contains a `HelloEvent` and a `WorldEvent` event. The `Init` method on the first `S_HelloWorld` simulation object schedules a `HelloEvent` at $t = 0.0$. When this event is executed, the string "Hello" is printed to the display and event, `WorldEvent`, is scheduled for 1 second later on the second simulation object. When event `WorldEvent` executes, the string "World" is printed to the display.

## 2.2 Simulation Object State

State refers to the value of the attributes of a simulation object. A principal difficulty inherent in parallel processing is time management. If simulation objects are distributed onto several nodes, one would like to take advantage of potential parallelism by allowing the processing on a node to run ahead without being hindered by the workload on another node. The case may occur, however, whereby a slow node schedules an event, called a straggler event, on a second node which has run ahead into the future, relative to the first node. If the straggler event is scheduled for a time in the faster node's past, then the state of the faster node must be rolled back to the point in time when the event is scheduled. SPEEDES provides a simple mechanism for accomplishing such a feat, transparent to the user, provided the user declares the state variable to be rollbackable (e.g. by using the built-in class `RB_int` instead of the usual C++ type `int`).

The code shown in Examples 2.9 through 2.11 will illustrate this. The simulation contains two counter objects, one running on a fast node, and the other on a slow node. The slow node can be made "slow" by including a call to the C library function, `sleep`. Each object contains a state variable, which it increments by 2 every two time units. At time, $t = 4.0$, the event, `IncrementEvent`, on the slow node schedules event, `StragglerEvent`, on the fast node for time, $t = 5.0$. The result of this event is to roll the fast node back to $t = 5.0$, add $1000$ to `Counter`, and continue on.

```
// S_Object.H
#ifndef S_Object_H
#define S_Object_H

#include "SpSimObj.H"
#include "SpDefineSimObj.H"
#include "SpDefineEvent.H"

class S_Object : public SpSimObj {
  public:
    S_Object() {}
    virtual void Init();
    void         IncrementEvent();
    void         StragglerEvent();

  private:
    RB_int Counter;
    char*  Indent;
};

DEFINE_SIMOBJ(S_Object, 2, SCATTER);
DEFINE_SIMOBJ_EVENT_0_ARG(IncrementEvent, S_Object, IncrementEvent);
DEFINE_SIMOBJ_EVENT_0_ARG(StragglerEvent, S_Object, StragglerEvent);
#endif
```

Example 2.9: Object State Simulation Definition File

```
// S_Object.C
#include <unistd.h>

#include "S_Object.H"
#include "SpGlobalFunctions.H"

void S_Object::Init() {
  Counter = 0;
  SCHEDULE_IncrementEvent(0.0, SpGetObjHandle());
  if (SpGetSimObjKindId() == 0) {
    Indent = "";
  }
  else {
    Indent = "    ";
  }
}

void S_Object::IncrementEvent() {

  cout << "t=" << SpGetTime().GetTime()
       << ": " << Indent << "Obj" << SpGetSimObjKindId()
       << "'s Counter = " << Counter;
  Counter = Counter + 2;
  cout << ", Incrementing it to " << Counter << endl;
  SCHEDULE_IncrementEvent(SpGetTime() + 2.0, SpGetObjHandle());

  if (SpGetSimObjKindId() == 0) {
    sleep(1);
    if (Counter == 6) {
      SCHEDULE_StragglerEvent(SpGetTime() + 1.0,
                              SpGetObjHandle("S_Object_MGR", 1));
      cout << "                "
           << "Scheduling StragglerEvent for time = "
           << SpGetTime() + 1.0 << endl;
    }
  }
}

void S_Object::StragglerEvent() {
  cout << "t=" << SpGetTime().GetTime()
       << ": " << Indent << "Obj" << SpGetSimObjKindId()
       << "'s Counter = " << Counter;
  Counter = Counter + 1000;
  cout << ", Incrementing it to " << Counter
       << " (Straggler)" << endl;
}
```

Example 2.10: Object State Simulation Implementation File

```
// Main.C
#include "SpMainPlugIn.H"
#include "S_Object.H"

int main (int argc, char** argv) {
  PLUG_IN_SIMOBJ(S_Object);
  PLUG_IN_EVENT(IncrementEvent);
  PLUG_IN_EVENT(StragglerEvent);
  ExecuteSpeedes(argc, argv);
}
```

Example 2.11: Object State Simulation Main

Figure 2.1 shows the output of the simulation for a 10 second run when run on one node.

```
t=0: Obj0's Counter = 0, Incrementing it to 2
t=0:     Obj1's Counter = 0, Incrementing it to 2
t=2: Obj0's Counter = 2, Incrementing it to 4
t=2:     Obj1's Counter = 2, Incrementing it to 4
t=4: Obj0's Counter = 4, Incrementing it to 6
                Scheduling StragglerEvent for time = 5
t=4:     Obj1's Counter = 4, Incrementing it to 6
t=5:     Obj1's Counter = 6, Incrementing it to 1006 (Straggler)
t=6: Obj0's Counter = 6, Incrementing it to 8
t=6:     Obj1's Counter = 1006, Incrementing it to 1008
t=8: Obj0's Counter = 8, Incrementing it to 10
t=8:     Obj1's Counter = 1008, Incrementing it to 1010
t=10: Obj0's Counter = 10, Incrementing it to 12
t=10:     Obj1's Counter = 1010, Incrementing it to 1012
```

Figure 2.1: One Node State Simulation

All of the timestamps in the output are in sequential order. This is due to the fact that, when simulations run on one node, rollbacks do not occur. Next, Figure 2.2 shows output for the same program when executed on two nodes.

```
t=0: Obj0's Counter = 0, Incrementing it to 2
t=0:     Obj1's Counter = 0, Incrementing it to 2
t=2:     Obj1's Counter = 2, Incrementing it to 4
t=4:     Obj1's Counter = 4, Incrementing it to 6
t=6:     Obj1's Counter = 6, Incrementing it to 8
t=8:     Obj1's Counter = 8, Incrementing it to 10
t=10:     Obj1's Counter = 10, Incrementing it to 12
t=2: Obj0's Counter = 2, Incrementing it to 4
t=4: Obj0's Counter = 4, Incrementing it to 6
                Scheduling StragglerEvent for time = 5
t=5:     Obj1's Counter = 6, Incrementing it to 1006 (Straggler)
t=6: Obj0's Counter = 6, Incrementing it to 8
t=6:     Obj1's Counter = 1006, Incrementing it to 1008
t=8:     Obj1's Counter = 1008, Incrementing it to 1010
t=10:     Obj1's Counter = 1010, Incrementing it to 1012
t=8: Obj0's Counter = 8, Incrementing it to 10
t=10: Obj0's Counter = 10, Incrementing it to 12
```

Figure 2.2: Two node State Simulation

The output shown in Figure 2.2 is useful for explaining rollbacks. Notice that Obj1 races forward to 10 seconds while Obj0 remains at 0. This is due to the fact that the node Obj0 is executing on (i.e. node 0) is currently sleeping for one second. After node 0 continues execution, it will eventually schedule event, `StragglerEvent`, at $t = 5$ on simulation object, `S_State`, on node 1. This causes the simulation object, `S_State`, on node 1 to rollback to $t = 5$. At this time, 1000 is added to `Counter`, which was 6 at $t = 4$. Therefore, when event, `IncrementEvent`, is reprocessed forward, the new value at $t = 6$ for Obj1 is 1006 on event entry.

If the `RB_int` type for attribute, `Counter`, is replaced with an `int` and the test is rerun, then the results will be different from the original. Node 1 once again races forward to 10 and `Counter` is set to 12. However, now when node 1 is rolled back, the `Counter` value is not 6, but retains its non-state safe value of 12. Hence, the new value of `Counter` is 1012. This shows that `Counter` is not rollbackable in this example.

In a simulation, users may not not want to have erroneous data values displayed (e.g. `Counter`). This problem can be remedied by replacing the stream output variable, `cout`, with its SPEEDES built-in rollbackable counterpart, `RB_cout`. Figure 2.3 shows the output of this program when executed on two nodes. Notice that the output is in time order.

```
t=0: Obj0's Counter = 0, Incrementing it to 2
t=0:     Obj1's Counter = 0, Incrementing it to 2
t=2: Obj0's Counter = 2, Incrementing it to 4
t=2:     Obj1's Counter = 2, Incrementing it to 4
t=4:     Obj1's Counter = 4, Incrementing it to 6
t=4: Obj0's Counter = 4, Incrementing it to 6
               Scheduling StragglerEvent for time= 5
t=5:     Obj1's Counter = 6, Incrementing it to 1006 (Straggler)
t=6: Obj0's Counter = 6, Incrementing it to 8
t=6:     Obj1's Counter = 1006, Incrementing it to 1008
t=8: Obj0's Counter = 8, Incrementing it to 10
t=8:     Obj1's Counter = 1008, Incrementing it to 1010
t=10: Obj0's Counter = 10, Incrementing it to 12
t=10:     Obj1's Counter = 1010, Incrementing it to 1012
```

Figure 2.3: Two node State Simulation with RB_cout

## 2.3 Event Handlers

SPEEDES provides a mechanism to dynamically add, subscribe, and remove a type of event called event handlers during simulation execution. With traditional point-to-point events, the caller must have knowledge about the event being called, along with what object the event resides on. However, with event handlers this information is unnecessary.

Event handlers have two basic forms, directed and undirected. Directed handlers are handlers for which the scheduler schedules the handler on a specific simulation object via a simulation object handle (i.e. `SpObjHandle`). A scheduler can schedule an undirected event handler by not specifing a simulation object handle. Usually, undirected event handlers are associated with a trigger string.

Examples 2.12 through 2.20 show examples of how to use undirected event handlers. The scheduling simulation object (`S_Scheduler`) does not know which entity receives the handler event, nor which event handlers are active. The receiving entities are made active by subscribing to a particular handler, and may respond differently to the handler event.

The code shown in Examples 2.12 through 2.15 shows how the English and Klingons will respond to a greeting. In both simulation objects, method, `Greet`, has been turned into an event handler by `DEFINE_SIMOBJ_HANDLER`. During run-time, these event handlers are subscribed to in the `Init` method with a trigger string of “Voice”. The effect of this is that any object that schedules an undirected event handler with a trigger of “Voice” will cause the method, `Greet`, to be executed on these two simulation objects.

The code shown in Examples 2.16 and 2.17 shows the Mute simulation object subscribing to an event handler with a trigger string of “Silent”. Therefore, this object will respond to trigger strings of “Silent” but not “Voice”.

Finally, the code shown in Examples 2.18 and 2.19 shows how to schedule an undirected event handler with the trigger of “Voice”. This causes the English and Klingon handlers to execute, while the event handler registered to object `S_Mute` remains silent.

```
// S_English.H
#ifndef S_English_H
#define S_English_H

#include "SpSimObj.H"
#include "SpDefineSimObj.H"
#include "SpDefineHandler.H"

class S_English : public SpSimObj {
  public:
    S_English() {}
    virtual void Init();
    void         Greet();
};

DEFINE_SIMOBJ(S_English, 1, SCATTER);
DEFINE_SIMOBJ_HANDLER(GreetInEnglish, S_English, Greet);
#endif
```

Example 2.12: English Definition File

```
// S_English.C
#include "S_English.H"

void S_English::Init() {
  SubscribeHandler(GreetInEnglish_HDR_ID(), "Voice");
}

void S_English::Greet() {
  cout << "Hello" << endl;
}
```

Example 2.13: English Implementation File

```
// S_Klingon.H
#ifndef S_Klingon_H
#define S_Klingon_H

#include "SpSimObj.H"
#include "SpDefineSimObj.H"
#include "SpDefineHandler.H"

class S_Klingon : public SpSimObj {
  public:
    S_Klingon() {}
    virtual void Init();
    void         Greet();
};

DEFINE_SIMOBJ(S_Klingon, 1, SCATTER);
DEFINE_SIMOBJ_HANDLER(GreetInKlingon, S_Klingon, Greet);
#endif
```

Example 2.14: Klingon Definition File

```
// S_Klingon.C
#include "S_Klingon.H"

void S_Klingon::Init() {
  SubscribeHandler(GreetInKlingon_HDR_ID(), "Voice");
}

void S_Klingon::Greet() {
//See http://www.kli.org/tlh/phrases.html
  cout << "nuqneH" << endl;
}
```

Example 2.15: Klingon Implementation File

```
// S_Mute.H
#ifndef S_Mute_H
#define S_Mute_H

#include "SpSimObj.H"
#include "SpDefineSimObj.H"
#include "SpDefineHandler.H"

class S_Mute : public SpSimObj {
  public:
    S_Mute() {}
    virtual void Init();
    void         Greet();
};

DEFINE_SIMOBJ(S_Mute, 1, SCATTER);
DEFINE_SIMOBJ_HANDLER(GreetWithANod, S_Mute, Greet);
#endif
```

Example 2.16: Mute Definition File

```
// S_Mute.C
#include "S_Mute.H"

void S_Mute::Init() {
  SubscribeHandler(GreetWithANod_HDR_ID(), "Silent");
}

void S_Mute::Greet() {
  cout << "ERROR: S_Mute is mute" << endl;
}
```

Example 2.17: Mute Implementation File

```
// S_Scheduler.H
#ifndef S_Scheduler_H
#define S_Scheduler_H

#include "SpSimObj.H"
#include "SpDefineSimObj.H"

class S_Scheduler : public SpSimObj {
  public:
    S_Scheduler() {}
    virtual void Init();
};

DEFINE_SIMOBJ(S_Scheduler, 1, SCATTER);
#endif
```

Example 2.18: Scheduler Definition File

```
// S_Scheduler.C
#include "S_Scheduler.H"
#include "SpSchedule.H"

void S_Scheduler::Init() {
  SCHEDULE_HANDLER(0.0, "Voice");
}
```

Example 2.19: Scheduler Implementation File

```
// Main.C
#include "SpMainPlugIn.H"
#include "S_English.H"
#include "S_Klingon.H"
#include "S_Mute.H"
#include "S_Scheduler.H"

int main (int argc, char** argv) {
  PLUG_IN_SIMOBJ(S_English);
  PLUG_IN_SIMOBJ(S_Klingon);
  PLUG_IN_SIMOBJ(S_Mute);
  PLUG_IN_SIMOBJ(S_Scheduler);
  ExecuteSpeedes(argc, argv);
}
```

Example 2.20: English and Klingon main File

## 2.4 The Process Model

An event on a simulation object occurs at a particular instance in time. In contrast, a process in SPEEDES refers to a reentrant event, which can take place over a finite interval of time. A process is reentrant in the sense that, when the simulation exits a process, then reenters, it can remember the values of local state variables inside the process. For example, an event could use the macro, `WAIT`, which instructs SPEEDES to exit the process, save the local state variables, and then reenter the process after the specified time interval has expired. The interval of time is specified by an argument to `WAIT`.

For example, the previous State example incremented a simulation object state variable, `Counter`, in an event. This event then schedules itself so that the counter can be subsequently incremented.

Examples 2.21 and 2.22 show how to increment the counter using the process model. Function `main` is not shown here, but is similar to those which have been previously presented (i.e. `main` plugs the simulation object and events into the SPEEDES framework and then calls `ExecuteSpeedes`).

```
// S_Object.H
#ifndef S_Object_H
#define S_Object_H

#include "SpSimObj.H"
#include "SpDefineSimObj.H"
#include "SpDefineEvent.H"

class S_Object : public SpSimObj {
  public:
    S_Object() {}
    virtual void Init();
    void         IncrementProcess();
};

DEFINE_SIMOBJ(S_Object, 1, SCATTER);
DEFINE_SIMOBJ_EVENT_0_ARG(IncrementProcess, S_Object,
                          IncrementProcess);
#endif
```

Example 2.21: Process Model WAIT Definition File

```
// S_Object.C
#include "S_Object.H"
#include "SpGlobalFunctions.H"
#include "SpProc.H"
#include "RB_ostream.H"

void S_Object::Init() {
  SCHEDULE_IncrementProcess(0.0, SpGetObjHandle());
}

void S_Object::IncrementProcess() {
  P_VAR;
  P_LV(int, counter);
  P_BEGIN(1);
  counter = 0;
  while(1) {
    RB_cout << "t=" << SpGetTime().GetTime()
            << ": " << "counter = " << counter;
    counter = counter + 2;
    RB_cout << ", Incrementing it to " << counter << endl;
    WAIT(1, 2.0);
  }
  P_END;
}
```

Example 2.22: Process Model WAIT Implementation File

To use the process model, you must first write a method on an object and use one the SPEEDES define event macros on this method. The event can be turned into a process model event by using the process model macros. There are several macros used to create a process model. The method, `IncrementProcess`, in our example, shows some of these process model constructs. The macro, `P_VAR`, starts the process model off. The user declares local state variables after this declaration (i.e. variables whose values need to be saved should the process model exit and then be reentered). The macro, `P_LV`, is used to declare variable, `counter`, as a local process model state variable. This causes the value for `counter` to be saved when method, `IncrementProcess`, is exited via process model normal processing. The macro call, `P_BEGIN(1)`, indicates where the process model user code starts (i.e. simulation model code). The integer argument to `P_BEGIN` specifies the number of subsequent process model macros used (e.g. `WAIT`, `WAIT_FOR`, `ASK`, etc.), which is 1 in this example. In the `while` loop, the call, `WAIT(1, 2.0)`, sets that particular reentry point to have a label of 1. When this code is executed the first time, the code up to the `WAIT` is executed, then, at the `WAIT` construct, the process model state variables are saved off and this event is exited. The second argument in the construct, `WAIT`, specifies the amount of time which must pass (as measured by GVT) before processing can continue. In this example, the time would be 2 seconds. The process code ends with the call to `P_END`. Although use of the process model constructs may seem difficult, there are many occasions when use of the process model can greatly simplify the code for your simulation models.

As a second example, consider the previous State example which will be modifed to show rollbacks. Examples 2.23 and 2.24 show a process model implementation for this rollback example. Event, `IncrementProcess`, has been modified to use the process model. Notice that the process model local state variable, `counter`, has been moved back to the header file so that event, `StragglerEvent`, can access the variable.

```
// S_Object.H
#ifndef S_Object_H
#define S_Object_H

#include "SpSimObj.H"
#include "SpDefineSimObj.H"
#include "SpDefineEvent.H"

class S_Object : public SpSimObj {
  public:
    S_Object() {}
    virtual void Init();
    void         IncrementProcess();
    void         StragglerEvent();

  private:
    RB_int Counter;
    char*  Indent;
};

DEFINE_SIMOBJ(S_Object, 2, SCATTER);
DEFINE_SIMOBJ_EVENT_0_ARG(IncrementProcess, S_Object,
                          IncrementProcess);
DEFINE_SIMOBJ_EVENT_0_ARG(StragglerEvent, S_Object, StragglerEvent);
#endif
```

Example 2.23: Process Model State Definition File

```
// S_Object.C
#include <unistd.h>

#include "S_Object.H"
#include "SpGlobalFunctions.H"
```

```
#include "SpProc.H"

void S_Object::Init() {
  SCHEDULE_IncrementProcess(0.0,SpGetObjHandle());
  if (SpGetSimObjKindId() == 0) {
    Indent = "";
  }
  else {
    Indent = "    ";
  }
}

void S_Object::IncrementProcess() {
  P_VAR;
  P_BEGIN(1);
  while(1) {
    cout << "t=" << SpGetTime().GetTime()
         << ": " << "Obj" << SpGetSimObjKindId()
         << "’s Counter = " << Counter;
    Counter = Counter + 2;
    cout << ", Incrementing it to " << Counter << endl;

    if (SpGetSimObjKindId() == 0) {
      sleep(1);
      if (Counter == 6) {
        cout << "                "
             << "Scheduling StragglerEvent for time="
             << SpGetTime() + 1.0 << endl;
        SCHEDULE_StragglerEvent(SpGetTime() + 1.0,
                                SpGetObjHandle("S_Object_MGR", 1));
      }
    }
    WAIT(1, 2.0);
  }
  P_END;
}

void S_Object::StragglerEvent() {
  cout << "t=" << SpGetTime().GetTime()
       << ": " << Indent << "Obj" << SpGetSimObjKindId()
       << "’s Counter = " << Counter;
  Counter = Counter + 1000;
  cout << ", Incrementing it to " << Counter
       << " (StragglerEvent)" << endl;
}
```

Example 2.24: Process Model State Definition File

# Part II

# Simulation Objects

# Chapter 3

# Simulation Object Overview

The simulation object represents the most basic concept in SPEEDES. All data representing the state of the object will be contained inside or referenced by an object descending from a simulation object. The fundamental building block for simulation objects is class `SpSimObj` or its child, `S_SpHLA`. All simulation objects must inherit from `SpSimObj` or one of its child classes.

Simulation objects, like other C++ classes, contain a collection of attributes whose types are primitive base types or other classes. If a simulation object's attributes are not state sensitive, then rollbackable types are not necessary. If a simulation object's attribute must maintain its state, then a rollbackable class must be used for the attribute. Section 2.2 shows an example of the use of non-rollbackable and rollbackable attributes in a simulation object. SPEEDES contains many built-in rollbackable classes which users can use to build their simulation objects. These built-in rollbackable classes are discussed in Chapter 4. If the built-in rollbackable classes are not sufficient to meet your needs, then you can always build your own rollbackable types and classes for use with the SPEEDES framework, as shown in Section 5.4.

This chapter will primarily focus on class `SpSimObj`. Class `S_SpHLA` will be discussed when Object Proxies are introduced in Chapter 9.

## 3.1 Simulation Objects

Class `SpSimObj` provides the most basic level of functionality for all simulation objects, such as the ability to schedule events, process event handlers, and respond to a special type of event handler, called interactions. In order to create the simulation object, users must write a class which inherits from `SpSimObj`. A simulation object is just like a normal C++ class in that it contains attributes and methods which define the behavior of the simulation object.

Let us walk though an example which shows the different aspects of a simulation object. Our example will print out the string "Hello from object #", where # is the integer id for that simulation object. The string will be printed out by an event which is executed at 1000 seconds into the simulation. Example 3.1 shows the code for the simulation object definition file.

```
// S_MySimObj.H
#ifndef MySimObj_H
#define MySimObj_H

```

```
#include "SpSimObj.H"
#include "SpDefineSimObj.H"
#include "SpDefineEvent.H"

class S_MySimObj : public SpSimObj {
  public:
    S_MySimObj();
    virtual ~S_MySimObj();
    virtual void Init();
    virtual void Terminate(double simTime);
    static int   GetNumObjs() {return 5;}
    void         MyFirstMethod();

  private:
    char*        StringOutput;
};

DEFINE_SIMOBJ(S_MySimObj, S_MySimObj::GetNumObjs(), SCATTER);
DEFINE_SIMOBJ_EVENT_0_ARG(MyMethod, S_MySimObj, MyFirstMethod);
#endif
```

Example 3.1: Simple Simulation Object Definition

Notice that this class inherits from `SpSimObj` as required to create a SPEEDES simulation object. This example also shows the typical C++ constructor and destructor along with two virtual methods `Init` and `Terminate`. While traditional C++ objects perform their initializations and cleanup in their constructors and destructors, respectively, it is highly recommended that SPEEDES simulation objects do this in the methods `Init` and `Terminate`. If there are tasks which must be initialized in the constructor, then these initializations should be added between the following `if` statement. This is due to the persistence framework, which may create or delete objects at unexpected times.

```
  if (SpPoDataBase::DoNotAllocateMemoryInConstructor == 0) {
    /*
     * Initializations that occur in the constructor are discouraged.
     * Object initializations should occur in the Init Method.
     */
  }
```

Next, notice the macros `DEFINE_SIMOBJ` and `DEFINE_SIMOBJ_EVENT_0_ARG` near the end of Example 3.1. Macro `DEFINE_SIMOBJ_EVENT_0_ARG` will be discussed in further detail in Chapter 6. For now, let it suffice to say that this macro is used to create an event called `MyMethod`, which, when called, will cause method `MyFirstMethod` to execute. Macro, `DEFINE_SIMOBJ`, is used to create a simulation object manager for the simulation object (i.e. creates a simulation object manager for `S_MySimObj`). During simulation initialization, object managers create (i.e. `new`) the user-specified number of simulation objects and call each object's `Init` method. The API for macro `DEFINE_SIMOBJ` is:

```
  DEFINE_SIMOBJ(className,
                numObjects,
                decompMethod)
```

| Parameter | Description |
|---|---|
| className | The name of the simulation object class. |
| numObjects | The number of objects to be created. This argument can be a function call as shown in Example 3.1. |
| decompMethod | The type of decomposition algorithm desired. Currently, SPEEDES contains four built-in types of decomposition algorithms, `BLOCK`, `SCATTER`, `FILE_SCATTER`, and `FILE_BLOCK`. |

Table 3.1: Macro DEFINE_SIMOBJ API

Examples 3.2 and 3.3 shows the code for the implementation of the simulation object and `main`.

```
// S_MySimObj.C
#include "SpGlobalFunctions.H"
#include "RB_ostream.H" // needed for RB_cout
#include "S_MySimObj.H"

S_MySimObj::S_MySimObj() {
  if (SpPoDataBase::DoNotAllocateMemoryInConstructor == 0) {
    /*
     * Initializations that occur in the constructor are discouraged.
     * Object initializations should occur in the Init Method.
     */
  }
}

S_MySimObj::~S_MySimObj() {
/*
 * Simulation object cleanup should be done in the Terminate method.
 */
}

/*
 * Initialize simulation objects here.
 */
void S_MySimObj::Init() {
  StringOutput = new char[strlen("Hello from object ") + 5];
  sprintf(StringOutput, "Hello from object %d", SpGetSimObjKindId());
  SetName(StringOutput);
  SCHEDULE_MyMethod(1000.0, SpGetObjHandle());
}

/*
 * Cleanup simulation objects here.
 */
void S_MySimObj::Terminate(double simTime) {
  cout << "S_MySimObj::Terminate at " << simTime << endl;
  delete [] StringOutput;
}

void S_MySimObj::MyFirstMethod() {
  RB_cout << StringOutput << endl;
}
```

Example 3.2: Simple Simulation Object Implementation

```
// Main.C
#include "SpMainPlugIn.H"
#include "S_MySimObj.H"

int main(int argc, char** argv)
{
  PLUG_IN_SIMOBJ(S_MySimObj);
  PLUG_IN_EVENT(MyMethod);
  ExecuteSpeedes(argc, argv);
}
```

Example 3.3: Simple Simulation main

Example 3.2 shows the implementation for the constructor, destructor, and the two virtual methods `Init` and `Terminate`. Notice that the constructor and destructor are not performing any tasks. Method `Init` allocates memory and assigns attribute `StringOutput` its initial value. It also schedules event `MyMethod` at time, $t = 1000$. In general, method `Init` is used for simulation object attribute initialization, scheduling initial events, and configuring event handlers and interactions. Method `Terminate` cleans up the simulation object on exit by deleting the memory allocated for `StringOutput`.

This example also shows the usage of method `SetName`. By default, SPEEDES names each simulation object by appending the string "`_MGR #`" to the class name used in macro `DEFINE_SIMOBJ`, where # is the object's kind id (see Section 3.2 for a description of kind id). For example, if method `SetName` had not been used, then the first `S_MySimObj` simulation object name would have been "`S_MySimObj_MGR 0`". The name of the object can be important when users need to look up simulation object handles (see Section 3.3). If method `SetName` is used to override the default name, then the new name must be used when looking up object handles.

The last item required to complete this simulation is to "Plug In" the SPEEDES simulation objects and their respective events into the SPEEDES framework. This is shown in Example 3.3. Two macros are provided with the SPEEDES framework which are used for plugging in simulation objects and events called `PLUG_IN_SIMOBJ` and `PLUG_IN_EVENT`, respectively. Macro `PLUG_IN_EVENT` will be discussed in Chapter 6. The API for macro `PLUG_IN_SIMOBJ` is:

```
PLUG_IN_SIMOBJ(className)
```

| Parameter | Description |
|---|---|
| className | The name of the simulation object class. |

Table 3.2: Macro PLUG_IN_SIMOBJ API

## 3.2 Simulation Object Managers

Simulation objects (i.e. objects that inherit from `SpSimObj`) are managed by simulation object managers. When a simulation is started, one simulation object manager for each simulation object type is created on each UNIX process, or node. For example, A simulation might contain 20 aircraft and 10 ships executing on 2 nodes. Each node will have one aircraft and one ship simulation manager objects and the 20 aircraft and 10 ships will be created on the two nodes by the simulation manager objects. Users can determine which node an object is located on by its node id, which is returned by global function `SpGetNodeId` (also, the number of nodes the simulation is executing on is accessable by global function `SpGetNumNodes`).

Simulation object managers are responsible for such items as:

- Initial simulation object creation on their respective nodes.
- Simulation initialization and cleanup processes.
- Dynamic simulation object creation.
- Managing subscriptions to interactions and event handlers.
- Managing external module subscriptions to simulation object data.

As the simulation object manager is creating and initializing simulation objects, several integer ideniti-fiers are assigned to each simulation object instance, including:

- Kind Id: As simulation objects are created by the SPEEDES framework, each simulation object instance (by type) is assigned a unique id starting at $0$. The function `SpGetSimObjKindId` returns the kind id for the current simulation object. As an example, consider a simulation composed of 10 aircraft, 3 airports, and 7 radars. This example contains 3 simulation object types. Within each object type, their kind id would be allocated as $0-9$, $0-2$, and $0-6$ for each instance of aircraft, airport, and radar, respectively. Therefore, in the more general case, kind id for each simulation object will go from 0 to $n-1$ where $n$ is the number of simulation objects of that type. The quantity of each simulation object is specified in the macro `DEFINE_SIMOBJ` as described in Table 3.1.

  As another example, suppose that a simulation contained 10 forts whose positions are fixed along a straight line from the equator to the north pole. When creating the forts, each fort needs to have its latitude and longitude initialized. Assume that all of the forts are on $0°$ latitude and that each fort lies at longitudes of $0°, 10°, 20°, 30°, \ldots$, up to $90°$. The kind id can be used to help avoid having two different nodes initializing one of their forts with identical longitudes. Simply write the code to set the longitude to 10 times kind id. Since SPEEDES assigns unique kind ids to the forts, each longitude will be unique as well. Table 3.3 shows the results of this allocation.

| Longitude | 0 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 |
|---|---|---|---|---|---|---|---|---|---|---|
| Kind Id | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Table 3.3: Fort Example, Longitude vs. Kind Id

- Local Id: This is a sequential list of numbers for each simulation object on the current node. The function `SpGetSimObjLocalId` returns the local id for the current simulation object. As an example, Table 3.4 shows what the local id could be for the simulation executing on 2 nodes. (Note: Decomposition algorithm can change the allocation of simulation objects, hence local ids).

| | Node 0 | | | | | Node 1 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Longitude | 0 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 |
| Kind Id | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Local Id | 0 | 1 | 2 | 3 | 4 | 0 | 1 | 2 | 3 | 4 |

Table 3.4: Fort Example, Longitude vs. Local Id

- Global Id: This is a sequential list of numbers which are unique throughout the entire simulation. Given the previous fort example, the forts would be assigned global ids $0-9$. Suppose the simulation also contain 10 tanks. The tanks would then be assigned global ids of $10-19$ ( their kind ids would be $0-9$ and local ids would be $0-4$ [per node]). Simulation objects may not have the same global ids from run to run if the simulation is executed on a different number of nodes. Function `SpGetSimObjGlobalId` returns the global id for the current simulation object.

Not only are simulation objects assigned identifying numbers, but simulation object managers are as well (each is unique). The identifiers for each simulation object manager are printed to the terminal during simulation initialization. Function `SpGetSimObjMgrId` returns the simulation object manager id for the current imulation object. This id can be looked up for any simulation object manager by using the following function:

```
int SpGetSimObjMgrId(char *name)
```

| Parameter | Description |
|---|---|
| name | The name of the simulation object manager. This is the first parameter used in `DEFINE_SIMOBJ` (i.e. class name) appended with string "`_MGR`". |

Table 3.5: Simulation Object Manager Id API

The creation of this simulation object manager is transparent to users. One of the byproducts of the macro `DEFINE_SIMOBJ` is the creation of the simulation object manager class. This class handles all of the simulation object manager duties. At times, users may need the simulation object manager id or its name for inputs into other functions when looking up objects. The most frequently used function will be `SpGetObjHandle`, which will be used when looking up object handles for simulation objects. Object handles are discussed in Section 3.3.

### 3.2.1 Simulation Object Decomposition

Object decomposition is the act of assigning simulation objects to nodes or UNIX processes. Ideally, each simulation node is allocated to its own processor on a high performance computer in order to maximize simulation run-time performance. For example, suppose our simulation consisted of 100 aircraft and 10 ships. For a 2 node simulation, how should the objects be distributed? Some distribution examples could be that all aircraft are placed on node 0 and ships on node 1, half the aircraft and ships on nodes 0 with the remaining aircraft and ships on node 1, or perhaps, 75 aircraft on node 0 and the remaining aircraft and ships on node 1. The placement of the aircraft and ships, or in the more general case, simulation objects, is left up to the user.

Obviously, the placement of the simulation objects onto their respective nodes can have a huge impact on processor or node load balancing. If all 100 airplanes with high computational expensive models were placed on one node and 10 ships with very simple models were placed on the other node, then the simulation would have very poor run-time performance. Hence, objects which require high CPU usage should be distributed across available CPUs to achieve a balanced CPU load and better run-time performance.

The placement of these objects onto nodes is called object decomposition. SPEEDES provides two built-in automatic object decomposition methods, called block and scatter, which will be discussed in more detail below.

#### 3.2.1.1 Block and Scatter Decomposition (Automatic Object Placement)

SPEEDES provides two automatic decomposition methods called block and scatter. When used, the simulation objects are distributed across available nodes based on the method specified. Block decomposition distributes the simulation objects to nodes evenly. However, as many objects as possible with adjacent kind ids will be placed on the same node. Scatter decomposition distributes the simulation objects such that simulation objects with consecutive kind ids are located on different consecutive nodes. This method is similar to a card dealer dealing out playing cards, hence scatter is often referred to as the card deal method of distribution.

The block decomposition algorithm starts off by calculating the number of simulation objects that can be placed on each node evenly. This is simply taking the integer result of the number of simulation objects divided by the number of nodes. For example, if we have 8 simulation objects and 3 nodes, then each node will receive 2 objects with 2 objects still needing distribution. The algorithm then takes any simulation objects not distributed and assigns them to nodes starting on the node after the node where the last simulation object was deposited. If, in our example, the last simulation object was placed on node 0, then our current 8 simulation objects would be distributed as follows: 3 simulation objects with kind ids of $0 - 2$ would be placed on node 1, the next 3 simulation objects with kind ids of $3 - 5$ would be placed on node 2 and the remaining simulation objects with kind ids of 6 and 7 would be placed on node 0.

The scatter decomposition algorithm uses a simple card deal method for simulation object distribution. The distribution always starts at the node following where the last simulation object was placed. Therefore, the current simulation object whose kind id is 0 is placed on the next available node, followed by the simulation object whose kind id is 1 placed on the next node, etc. This continues until the all simulation objects of a given type are distributed. For example, suppose we once again had 8 simulation objects with 3 nodes and SPEEDES placed the last simulation object on node 0. Then node 1 gets the next object whose kind id is 0, node 2 gets the next object whose kind id is 1, node 0 gets the next object whose kind id is 2, etc.

Let us look at the previous example again. Suppose the forts were to be allocated across two nodes. Table 3.6 shows how simulation objects would be distributed if block decomposition were used.

| Longitude | 0 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 |
|---|---|---|---|---|---|---|---|---|---|---|
| Kind Id | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Node | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |

Table 3.6: Block Decomposition

Suppose the forts in our simulation wanted to share ammunition and each fort can only share ammunition with the adjacent forts. In that case, the above decomposition would be good since in most cases the interaction between simulation objects would be on the same node. Only the infrequent case of the forts whose kind ids are 4 and 5 would require communication across nodes. All other fort communications would be on same node. Since communication with other nodes is more time-consuming, this decomposition would minimize the more expensive communication. This distribution method is called block decomposition.

Now suppose the forts do not share ammunition at all, but instead, their primary purpose is to defend against invasion. Invasions tend to occur in small geographic areas since the enemy likes to concentrate its forces. If the invasion occurred at $45°$ longitude, then only forts that are close to this longitude would be able to defend against the invasion. In this case, only forts with kind ids of 4 and 5 are close enough. Since these two forts are on separate nodes, each node would perform approximately one half of the work load so the simulation would be well balanced. However, suppose the invasion occurred at a longitude of 75°. Now forts 7 and 8 would defend against the invasion and, with block decomposition, all of the work would be done on node 1. In this case, node 1 is overloaded and node 0 is underloaded, creating an unbalanced load.

Therefore, in this case, having adjacent forts on different nodes would maximize run-time performance. That way, any invasion involving two adjacent forts would be simulated on both nodes with roughly equally work loads. The nodes would be balanced. This could be achieved by having the simulation objects with even numbered kind ids be on node 0 and the simulation objects with odd numbered kind ids be on node 1. This type of distribution is call scatter decomposition. Table 3.7 shows the forts distribution pattern using scatter decomposition.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Longitude | 0 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 |
| Kind Id | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Node | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |

Table 3.7: Scatter Decomposition

However, notice that scatter decomposition would be a poor fort distribution method for the first hypothetical problem of ammunition sharing between forts. There are always pluses and minuses when choosing any decomposition method. The decomposition method will affect the run-time performance of a simulation and users should consider this issue when selecting a method. The decomposition method is specified in the third parameter in macro `DEFINE_SIMOBJ` as `BLOCK` and `SCATTER`, as shown in Table 3.1.

Regardless of which type of decomposition method selected, SPEEDES picks up where it left off. That is, if the last object of the previous type of simulation object was placed on node 3, then the first object of the next simulation object type is placed on the next available node (e.g. node 0 on a 4 node simulation).

As a last example, consider a simulation involving the following simulation objects with the specified quantities, as shown in Table 3.8.

| **Object Type** | **Objects Type Quantity** | **Decomposition Type** |
|---|---|---|
| Car | 2 | scatter |
| Ship | 8 | block |
| Plane | 6 | scatter |
| Train | 4 | scatter |

Table 3.8: Decomposition Example

In addition, assume that the simulation is to be run on three nodes with the objects plugged in the order specified above. The simulation objects are then distributed across the different nodes, as shown in Table 3.9.

| **Node 0** | **Node 1** | **Node 2** |
|---|---|---|
| Car0 | Car1 | Ship0 |
| Ship3 | Ship6 | Ship1 |
| Ship4 | Ship7 | Ship2 |
| Ship5 | Plane0 | Plane1 |
| Plane2 | Plane3 | Plane4 |
| Plane5 | Train0 | Train1 |
| Train2 | Train3 | |

Table 3.9: Block and Scatter Object Distribution by Object Type and Kind Id

#### 3.2.1.2 File Driven Decomposition (Manual Object Placement)

Block and scatter decompositions often do fairly well when the number of objects is large and the interactions between objects is somewhat random. However, in many simulations, this is not the case. For simulations that contain simulation objects which are tightly coupled and will tend to roll each other back if they reside on different nodes, then manual placement can achieve better simulation performance results. SPEEDES allows users to specify object placement via an input file specification.

SPEEDES allows for two different file driven object placement schemes called file block and file scatter by specifying `FILE_BLOCK` and `FILE_SCATTER` as the third parameter in macro `DEFINE_SIMOBJ`. When using these types, the user must supply a file called `SimObjPlacement.par`, which specifies the simulation object versus node layout matrix. All simulation objects do not have to be placed on nodes in this specification. All simulation objects not specified in the input file will be placed on nodes based on the input parameter `FILE_BLOCK` and `FILE_SCATTER` (i.e. an object not specified in the input file will be placed on a node based on the block or scatter decomposition algorithm). Figure 3.1 is an example of a placement file for placing objects `S_PlacedObjectA`, `S_PlacedObjectB`, and `S_PlacedObjectC`.

```
int     NumNodes                   3  // Number of nodes
logical ComprehensiveSimObjMgrs    F  // All object MGRs required?
logical ComprehensivePlacement     F  // All instances place?

S_PlacedObjectA_MGR {
  logical ComprehensivePlacement T  // All instance specified
  int     SimObj_0                 0  // Kind id is 0 is on node 0
  int     SimObj_1                 2  // Kind id is 1 is on node 2
  int     SimObj_2                 2  // Kind id is 2 is on node 2
  int     SimObj_3                 1  // Kind id is 3 is on node 1
}

S_PlacedObjectB_MGR {
  int     SimObj_2                 0  // Kind id is 2 is on node 0
}

S_PlacedObjectC_MGR {
  int     SimObj_0                 0  // Kind id is 0 is on node 0
  int     SimObj_1                 2  // Kind id is 1 is on node 2
}
```

Figure 3.1: Manual Object Manager Input File (SimObjPlacement.par)

The first three lines in the input file are optional. Their definitions are as follows:

- `NumNodes`:
  If the number of nodes is specified, then the simulation must be run with the number of nodes specified. If a simulation is started with a different number of nodes, then the simulation prints an error message and exits.

- `ComprehensiveSimObjMgrs`:
  When this is set to true, then all object types which use `FILE_BLOCK` or `FILE_SCATTER` must be specified in this file. For example, suppose that the simulation represented by Figure 3.1 also contains a simulation object `S_PlacedObjectD`, which used `FILE_SCATTER` as part

of its definition. Then, when `ComprehensiveSimObjMgrs` is set to false, a definition for `S_PlacedObjectD_MGR` is not required in file `SimObjPlacement.par`. However, if `ComprehensiveSimObjMgrs` is set to true, then an entry for `S_PlacedObjectD_MGR` must be found in file `SimObjPlacement.par`. The default value is false.

- `ComprehensivePlacement`:
  When this is set to true, then, for all specified object types, all instances of that type must be specified. This value can then be overridden for a specific object type. The default value is false.

All of the lines following the first three are used to specify object instance placement. Object instances are grouped within each object manager (i.e. object managers are derived from the name supplied to the `DEFINE_SIMOBJ` appended with the string "`_MGR`"). Each object instance is identified in the file by using `SimObj_#` where # is the simulation object's kind id.

In Figure 3.1, this input file specifies the number of nodes as 3, not all object managers must be present and not all objects within each manager type must be specified. The specification for object `S_PlacedObjectA` overrides `ComprehensiveSimObjMgrs` by setting its value to true. Therefore, all objects must have their placements specified. If an object is not specified, then an error message will be printed and the simulation will exit. This input file contains four instances of object type `S_PlacedObjectA`, which are placed on nodes 0, 2, 2, and 1. Node placements for object type `S_PlacedObjectB` are specified by the entry `S_PlacedObjectB_MGR`. For this specification file, only the object instance whose kind id is 2 is placed. All other object instances of type `S_PlacedObjectB` will be placed based on the decomposition algorithm specified (i.e. `FILE_BLOCK` or `FILE_SCATTER`). The decomposition algorithm works as follows:

1. SPEEDES reads the placement file `SimObjPlacement.par` and determines the location for all specified objects.
2. Next, objects are placed using the specified decomposition method. The objects are placed in the order in which they are plugged in.
3. As the decomposition algorithm advances, it will eventually attempt to place an object that was specified in the placement file. When this occurs, the object is placed on the node as specified in the input file and the automatic placement location for this object is skipped. For example, suppose an object was supposed to to be placed on node 1 and the placement file specified it should be placed on node 0. The object will be placed on node 0, node 1 will be skipped, and the next object to be placed will be placed on node 2.

This method of manual object placement maintains the placement of all of the auto-placed objects, moving only the objects placed by the specification file.

Consider the previous example shown in Table 3.8. Let us modify the example to use the following placement file specification. The ship and the plane will use file block and file scatter decomposition, respectively.

```
int NumNodes 3

S_Ship_MGR {
  int SimObj_0 1
  int SimObj_6 2
}

S_Plane_MGR {
  int SimObj_1 1
  int SimObj_2 2
  int SimObj_3 2
}
```

Figure 3.2: Car, Ship, Plane, and Train File Placement

Notice that the input file does not contain definitions for the car and train simulation objects. The decomposition method for these objects could be either scatter or file scatter. Since `ComprehensiveSimObjMgrs` is not set to true, either decomposition method specification will work. The result of this placement specification is shown in Table 3.10. Notice that all of the objects not specified in the input file are placed on the same nodes as shown in Table 3.9.

| **Node 0** | **Node 1** | **Node 2** |
|---|---|---|
| Car0 | Car1 | |
| Ship3 | | Ship1 |
| Ship4 | Ship7 | Ship2 |
| Ship5 | Plane0 | |
| | | Plane4 |
| Plane5 | Train0 | Train1 |
| Train2 | Train3 | |
| **Placed Objects** | | |
| | Ship0 | Ship6 |
| | Plane1 | Plane2 |
| | | Plane3 |

Table 3.10: File Driven Placement

## 3.3 Simulation Object “Object Handles”

When a simulation object is created, a unique object handle is assigned. An object handle is an instance of the SPEEDES class `SpObjHandle`, and every simulation object has a unique handle. This class encapsulates the data required to locate any simulation object among all simulation objects within a simulation.

Class `SpObjHandle` consists of three items which will uniquely identify all simulation objects within a SPEEDES simulation. These items include the node that the object resides on, the type of the object (manager id), and the local id of the object. Objects of the same type on a single node are numbered sequentially from 0 to the number of objects - 1. This means that, in some cases, objects of the same type on different nodes will have the same local id. Contrast this with how objects of the same type on different nodes will always have different kind ids. The constructor for an object handle looks as follows:

```
SpObjHandle(int nodeId, int simObjMgrId, int simObjLocalId)
```

As an example, if we had three objects of type 12 running on two nodes with scatter decomposition, we would have object handles of $(0, 12, 0)$, $(1, 12, 0)$, and $(0, 12, 1)$, where object handles have the form of (Node, Object Type, Local Id). This corresponds to the three objects residing on nodes 0, 1, and 0, respectively, whose type is 12 with local ids of 0, 0, and 1, respectively. Also, it is worth mentioning here that on each node there is an object manager managing each of these simulation objects. Each object manager is assigned an object handle whose type is 12 with its local id set to -1. Therefore, the object handles for the object managers would be $(0, 12, -1)$ and $(1, 12, -1)$.

Building object handles manually requires quite a bit of knowledge about how the objects are created and where they are distributed. The SPEEDES framework sets the object handles to their appropriate values during object creation, which alleviates the user from initializing these. However, the user needs to understand object handles, since object handles are a required field when scheduling events. The SPEEDES framework provides a set of global functions for looking up object handles. Table 3.11 shows these global functions which are defined in header file `SpGlobalFunctions.H`.

| **Function** | **Description** |
|---|---|
| `SpObjHandle SpGetObjHandle()` | Returns the object handle for the current simulation object. This is useful for scheduling events on the current simulation object. The current simulation object is the object on which the currently processed event is acting, or the object whose `Init` function is executing. |
| `SpObjHandle SpGetObjHandle( int simObjKindId)` | Returns the object handle of the object whose kind id is passed in and whose type is the current simulation object's type. This function cannot be used if the current simulation object is managed by a manual manager. Manual managers are managers created without using `DEFINE_SIMOBJ`. |
| `SpObjHandle SpGetObjHandle( char* objType, int   simObjKindId)` | Looks up an object handle by its type string name and kind id. The string name of a type is the name of the simulation object's name postfixed with the string "`_MGR`". This function cannot be used if the current simulation object is managed by a manual manager. An example of the method is shown below: `class S_Ship : public SpSimObj { ... }; ... SpGetObjHandle("S_Ship_MGR", 0);` |
| `SpObjHandle SpGetObjHandle( char* objType, char* objName)` | If kind id is not known, it is also possible to identify the object by object type and name. An example of the method is shown below: `class S_Ship : public SpSimObj { ... }; ... SpGetObjHandle("S_Ship_MGR", "S_Ship_MGR 0");` In this example, the object name is the auto-generated name for the simulation object. If method `SetName` is used to override the auto-generated name then that name must be used. This function cannot be used during simulation object initialization (i.e. `Init`). |

| **Function** | **Description** |
|---|---|
| `SpObjHandle SpGetObjHandle( int mgrId, int simObjKindId)` `SpObjHandle SpGetObjHandle( int mgrId, char* objName)` | These functions are the same as the previous two, except that they use the simulation object manager id, rather than string look ups. The simulation object manager id is discussed in Section 3.2. |

Table 3.11: Object Handle Global Functions

## 3.4 Tips, Tricks, and Potholes

- Many actions fail or cause the simulation to crash if they are performed inside of the constructor. These actions include adding event handlers and subscribing to interactions. In fact, it is recommended that little be done in the constructor and all of the initialization of the simulation object be performed in the `Init` method.
- Looking up an object handle using strings for either the type or the object name can be relatively slow (especially when there are a large number of objects in the simulation). Consequently, passing the integer representing kind id is much faster, as is the passing of the integer identifying the object manager. Another recommendation is to grab the object handle once and store it if it is to be used often. Finally, if the string has a typographical error in it, the compiler will not catch the error.
- As previously explained, you may refer to a simulation object type by referring to its class name with some extra characters appended. When using the class name as a C++ string, the string "`_MGR`" needs to be appended to the class name. When its integer id is used, the string "`_MGR_ID`" needs to be appended to the class name. Here is an example which illustrates this:

```
class S_Ship : public SpSimObj {
    ...
};
...
SpGetObjHandle("S_Ship_MGR", 0)
SpGetObjHandle( S_Ship_MGR_ID, 0)
```

# Chapter 4

# Rollbackable Built-in Types

Simulation objects need to be made rollbackable, because SPEEDES is an optimistic framework. This means that any changes to the state of a simulation object must be recorded so that the state may be restored in case an event is rolled back. This goal is usually achieved through use of the built-in, rollbackable data types that are included with the framework. When these are not sufficient, rollbackable types can be created by the user in order to support additional functionality. This will be discussed in Section 5.4.

## 4.1 Rollbackable Data

From a C++ point of view, a simulation object can be as generic as desired as long as the simulation object inherits from `SpSimObj`. Using this object in an optimistic PDES environment raises a number of issues which we will explore.

When processing a simulation in parallel, one or more of the CPUs (i.e. nodes) on which the simulation is distributed may run ahead in time relative to other nodes. An event on a simulation object being processed on a slower node may schedule an event for a simulation object being processed on one of the faster nodes. Since the scheduling node is running slow, it may schedule an event for the past as measured by the fast node. Such an event is called a straggler event. Events that were processed on the faster simulation object need to be reprocessed when an event in the past occurs. The event that was processed may have changed the state of the simulation object on the fast node, and those changes must be undone.

For example, suppose a tank simulation object contained an event which fires ammunition and decrements its ammunition count (e.g. ammunition count drops from 5 to 4 on weapon fire). Then, suppose an event is scheduled for the tank prior to its fire event. For this case, the tank needs to undo the fire event and restore the ammunition count back to its previous value (e.g. ammunition count is restored to 5). The easiest way to manage change to a simulation object's state is through use of built-in rollbackable data types. Additionally, an object's state can be managed through the use of an advanced feature of SPEEDES called exchange, which is discussed in Chapter 15.

The most basic rule that should be followed is: *If the value of a variable in the simulation object changes as the result of processing an event, then that variable needs to be rollbackable.* This rule applies to container classes, pointers, basic data types, and aggregated classes that are contained in the simulation object.

When the built-in or user-defined rollbackable classes are used to build the contents of a simulation object, then all changes to these attributes are recorded. During a simulation execution, if a simulation object is rolled back, then the rollbackable attributes on that simulation object are restored to their original values. On GVT updates, all rollbackable data which was saved is released and reused. Rollbackable classes should never be used to declare temporary variables on the stack (i.e. local variables in class methods). When this occurs, it is very likely that this will result in memory corruption and application run-time failure (i.e. core). When users mistakenly use rollbackable variables on the stack, SPEEDES will output an error similar to what is shown in Figure 4.1.

```
***********************************************************************
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
-----------MAJOR ERROR!!!-------------
Memory range [0x7ffff6f8, 0x7ffff6fc] is being accessed in the rollback
queue in an event of type 0
rolling back an SpAltIntValue
at time Time = {500,0,0,6,0}
on an object of type 0
and name S_RBTest_MGR 0
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
***********************************************************************
```

Figure 4.1: Rollback Variable on Stack Error Message

The name of a rollbackable data type or function is specified by prefixing the name of the corresponding non-rollbackable type or function with the string "`RB_`". For example, the rollbackable types for `int` and `strdup` are `RB_int` and `RB_strdup`, respectively. The following sections introduce the SPEEDES built-in rollbackable data types.

## 4.2 Basic Data Types

### 4.2.1 Rollbackable Integers and Doubles

Built-in rollbackable data types `RB_int` and `RB_double` behave almost exactly like their non-rollbackable counterparts `int` and `double`, respectively. These two types provide rollbackable data holders for integers and doubles. When variables of these types are declared, they are initialized to zero. The definitions for these classes are described in include files `RB_int.H` and `RB_double.H`. Figures 4.2 and 4.3 show overloaded operators for these classes.

```
     operator    int()
int  operator    ++()
int  operator    ++(int)
int  operator    --()
int  operator    --(int)
int  operator     =(int)
int  operator     =(const int&)
int  operator    +=(int)
int  operator    -=(int)
int  operator    *=(int)
int  operator    /=(int)
int  operator    %=(int)
int  operator    ^=(int)
int  operator    &=(int)
int  operator    !=(int)
int  operator   >>=(int)
int  operator   <<=(int)
```

Figure 4.2: RB_int Operators

```
        operator   double()
double  operator   ++()
double  operator   ++(double)
double  operator   --()
double  operator   --(double)
double  operator    =(double)
double  operator    =(const double&)
double  operator   +=(double)
double  operator   -=(double)
double  operator   *=(double)
double  operator   /=(double)
```

Figure 4.3: RB_double Operators

### 4.2.2 Rollbackable Strings and Void Pointers

Classes `RB_SpString` and `RB_voidPtr` provide the capability of having class attributes, which handle strings and pointers in a rollbackable fashion. The `RB_SpString` class is a rollbackable class that otherwise behaves like an ordinary `NULL` terminated string. Declare a variable as `RB_SpString` when you need a string whose value changes from event to event. The `RB_voidPtr` class is a rollbackable class for handling pointers to generic objects. The definitions for these classes are described in include files `RB_SpString.H` and `RB_voidPtr.H`.

In addition to the default constructor and destructor, class `RB_SpString` supports the functionality shown in Figure 4.4.

```
RB_SpString(const char*)
                operator const char*() const
const char* operator =(const char*)
```

Figure 4.4: RB_SpString Operators

When `RB_SpString` default constructor is used, `RB_SpString` is initialized to a `NULL` pointer and not an empty string. In other words, it is a character pointer (i.e. `char*`) that points to address, `0x0`. In addition to the default constructor, another constructor exists which allows users to construct a `RB_SpString` initialized to some input string value.

The next two operators are the default conversion to `const char*` and assignment. The assignment operator copies the input string into storage within the `RB_SpString` object. The input string does not need to reside in permanent storage. Changes to a specific element of a string are not permitted (i.e. operator `[]` is private).

Class `RB_SpString` handles strings for users in a rollbackable fashion. Strings are a special type of pointer (i.e. `char*`). The built-in class `RB_voidPtr` has been provided to handle the more generic pointer cases. For example, you might need a rollbackable `void*` or a rollbackable `blob*`, where `blob` is some arbitrary class name. You may need to set the pointer to point somewhere else, but if the event gets rolled back, then you need the address to be restored to the old address. For the first case, you may use `RB_voidPtr`, and for other cases, like `blob*`, you may still use `RB_voidPtr`, but then cast `RB_voidPtr` to `blob*` when reading it. This class behaves similarly to a standard `void*` in C++. Figure 4.5 shows the operators available for class `RB_voidPtr`.

```
RB_voidPtr(void value = NULL)
              operator void*() const
void*         operator =(void*)
```

Figure 4.5: RB_voidPtr Operators

Class `RB_voidPtr` constructor has an optional parameter, which can be used to initialize the class with a pointer value. If one is not provided, then the pointer is defaulted to `NULL`. The first operator is a standard conversion operator that will change a `RB_voidPtr` to a `void*`. The second is a rollbackable assignment operator. Notice that, while the pointer assignment is rollbackable, it does not copy the storage referenced by the pointer (as does the `RB_SpString` assignment).

### 4.2.3 Rollbackable Booleans

`RB_SpBool` is a boolean class that can take on values of `RB_SpBool::SpFALSE` or `RB_SpBool-::SpTRUE` The default initial value is `RB_SpBool::SpFALSE`. This class supports rollbackable events by restoring prior boolean states. The definitions for this class are described in include file `RB_SpBool.H`. Figure 4.6 shows the methods available in class `RB_SpBool`. There are several methods available for examining and changing the state of an `RB_SpBool`

```
enum SpBoolState {
  SpFALSE,
  SpTRUE}
            operator int()    const
            operator SpBool() const
SpBool      operator ==(SpBool)
SpBool      operator !=(SpBool)
SpBool      operator  !()
SpBool      operator  =(SpBool)
RB_SpBool&  operator  =(const RB_SpBool&)
SpBoolState IsTrue()  const
SpBoolState IsFalse() const
```

Figure 4.6: RB_SpBool Operators

The return value of many of the `RB_SpBool` operators belongs to a class called `SpBool`, which is a non-rollbackable version of the `RB_SpBool` class. The integer conversion operator allows an object of this class to be used in conditional statements. Additionally, `SpFALSE` or `SpTRUE` allow testing for equality.

### 4.2.4 Rollbackable Streams

Frequently, it is necessary to output information to the user of the simulation. Use of standard C++ streams, such as `cout`, can result in incorrect output, since an event may be rolled back and reprocessed, giving different output or may not be reprocessed at all. To avoid this problem, several classes are provided that behave similarly to C++ streams, but only stream (output or transfer) the data when the event is committed (see Chapter 15) and is therefore certain not to be rolled back. Classes `RB_ostream` and `RB_exostream` provide rollbackable stream capability. The definitions for these classes are described in include files `RB_ostream.H` and `RB_exostream.H`.

The class `RB_ostream` provides standard `ostream` type functionality. This class has the following constructors:

```
RB_ostream(char*     filename)
RB_ostream(ostream*  ostream)
```

The first constructor creates a `RB_ostream` which writes to a file named filename. The second creates a `RB_ostream` from the already existing output stream, `ostream`, but will only write to it in a rollback protected fashion. This results in the desired behavior when used with a stream such as `cout` or `ofstream` but often leads to unexpected behavior with an `ostrstream` (i.e. the data is not written to the stream until the event is committed). Figure 4.7 shows the available operators for class `RB_ostream`.

```
RB_ostream operator <<(char data)
RB_ostream operator <<(short data)
RB_ostream operator <<(int data)
RB_ostream operator <<(long data)
RB_ostream operator <<(unsigned char data)
RB_ostream operator <<(unsigned short data)
RB_ostream operator <<(unsigned int data)
RB_ostream operator <<(unsigned long data)
RB_ostream operator <<(float data)
RB_ostream operator <<(double data)
RB_ostream operator <<(void* data)
RB_ostream operator <<(char* data)
RB_ostream operator <<(const char* data)
RB_ostream operator <<(unsigned char* data)
RB_ostream operator <<(SpOmanip manipulator)
RB_ostream operator <<(SpBool data)
RB_ostream operator <<(SpSimTime data)
```

Figure 4.7: RB_ostream Operators

These all behave as expected for outputting data. SPEEDES also includes two global variables, `RB_cout` and `RB_cerr`, for writing to the standard streams. Note that, unlike `cerr`, data that is written to `RB_cerr` is not written to standard error until the event is committed. `RB_cout` is analogous to cout.

Use of `RB_cout` can result in the output appearing out of time order when running on multiple nodes. The rollbackable external stream (i.e. `RB_exostream`) has a similar interface as class `RB_ostream`, but writes data to an external module and the output is sorted by time. Class `RB_exostream` supports the same "write to" operators but supports just one constructor: `RB_exostream (char *streamName)`, where `streamName` is the name of the stream. This name is used by external modules to separate out the data coming from different external streams. Data inserted into an instance of `RB_exostream` will not be sent out across the network until an `endl`, `flush`, or `ends` stream manipulator is inserted into the `RB_exostream`. Unfortunately, `RB_exostream` does not work when SPEEDES is executing in sequential mode. If SPEEDES is run on one node, then `optimize_sequential` must be set to false in `speedes.par`. See Appendix C.1 for more information on how to set this parameter.

Note that these classes do not support all of the features found in the built in `ostream` system. In particular, they do not support most manipulators or user-defined output operators. For the example shown below, the first line works fine but the second line has compilation errors.

```
  cout    << "Set width to 3: " << setw(3) << 8 << endl;
  RB_cout << "Set width to 3: " << setw(3) << 8 << endl;
```

The work around for this problem is to put the desired string with manipulators into an ordinary `strstream` and then put that `strstream` to the `RB_ostream`.

## 4.3 Container Classes

For the development of basic simulation models, most data needs are satisfied by the basic types, such as `RB_int` and `RB_cout`. As complexity increases, however, the need for container classes increases.

SPEEDES provides several container classes with a variety of different performance characteristics, to aid in modeling.

Each rollbackable container described in the following sections has a non-rollbackable counterpart. The name for the non-rollbackable container will be the same as the rollbackable version, minus the "`RB_`" name prefix. Also the user should avoid adding `NULL` to the containers. The iterator and search algorithm depends on `NULL` terminated data, and an unexpected `NULL` on a container will result in incorrect behavior.

### 4.3.1 Rollbackable Binary and Hash Trees

Class `RB_SpBinaryTree` is a standard binary tree that can behave in either normal, balanced, or splay tree modes. Class `RB_SpHashTree` supports a similar interface but differs in that the elements cannot be traversed in a sorted fashion. It does have a consistent order, only it is an arbritray order rather than a sorted order. These classes support using either a double or a string as the key for inserting, sorting, or searching a tree, but not both at the same time. These classes allow for the insertion of duplicate keys. Items inserted into a tree with duplicate keys are preserved in insertion order (i.e. those inserted first will be found first when walking through the tree). The definitions for these classes are described in include files `RB_SpBinaryTree.H` and `RB_SpHashTree.H`.

These containers store all data as `void*` pointers, thus leaving the user responsible for properly creating and deleting data that is inserted into the container. The methods available for modifying the contents of a tree are shown in Figure 4.8.

```
void  Insert(void*  item, double keyNum)
void  Insert(void*  item, char*  keyName)
void* Remove(double keyNum)
void* Remove(char*  keyName)
void* RemoveFirstElement()
void* RemoveFirstElement(double& keyNum)
void* RemoveLastElement()
void* RemoveLastElement(double& keyNum)
```

Figure 4.8: RB_SpBinaryTree and RB_SpHashTree Modifier Methods

`Insert` always succeeds and all `Remove` methods return the item that was removed, or `NULL` if no such element was found. In the case of multiple items with the same key, `Remove` will return the first of those items that was inserted into the tree. `RemoveFirstElement` and `RemoveLastElement` will also accept a reference, in which they will return the key corresponding to the first or last element, respectively.

Figure 4.9 shows the methods available for searching and traversing a tree.

All of the search and iterator methods shown in Figure 4.9 return a `NULL` pointer if the tree is empty, the end of the tree is reached, or if the item being searched for is not found. Method `Find` searches the tree for the item specified by the key and, if found, returns a pointer to that item. The item is not removed from the tree.

New items are inserted into a tree via a key which must be either a double or a string. If a tree has a key of one type, then functions accepting other types are not allowed. Once the key type has been selected, then the appropriate `Find` or `GetCurrentKey` method must be used. If the incorrect `Find` or `GetCurrentKey` is used, then an error message will be output. For example, if `GetCurrentKey`

```
int    GetNumElements() const
void*  Find(const double key)
void*  Find(const char* key)
void*  GetFirstElement()
void*  GetCurrentElement()
void*  GetLastElement()
void*  GetNextElement()
void*  GetPreviousElement()
void*  operator ++()
void*  operator ++(int)
void*  operator --()
void*  operator --(int)
double GetCurrentKey()
char*  GetCurrentKeyName()
```

Figure 4.9: RB_SpBinaryTree and RB_SpHashTree Search and Iterator Methods

was used on a tree which has string keys, then an error message is displayed and this method returns `-1`. Likewise, if the tree was built using doubles as keys, then method `GetCurrentKeyName` would return `NULL`.

The tree container classes also provide prefix and postfix increment and decrement operators which move forward and backward through the containers just like the Get methods. The iterator functions are not rollbackable and `GetNextElement` should only be called after calling some function which has set the current element, like `GetFirstElement` or `GetLastElement`. The following code shows how to traverse all of the elements in a binary tree whose name is `AccountTree`.

```
Account* myAccount;
myAccount = (Account *) AccountTree->GetFirstElement();
while (myAccount != NULL) {
  // Examine account data
  myAccount = (Account *) AccountTree->GetNextElement();
}
```

Reverse searches can be performed by replaceing `GetFirstElement` and `GetNextElement` with `GetLazyElement` `GetPreviousElement`, respectively.

There is only one iterator for each tree object. Therefore, do not write a loop which contains an inner loop interating on the same container. When processing an item in the body of a loop like the one above, do not call another function which also iterates over the same tree object. If the container class is modified (i.e. an insert or remove method used) between two accessing methods, undefined results may occur. Therefore, used methods `GetFirstElement` or `GetLastElement` after elements have been added or removed from a tree container and before any other Get methods or operators are applied to the container. To get around these limitations, use the interator class on these containers (see Section 4.4).

Finally, tree container classes can operate in one of three modes called normal, splay, or balanced. Figure 4.10 shows methods which can set or query the tree mode. The default mode after constructing `RB_SpBinaryTree` is normal tree mode. The default for `RB_SpHashTree` is balanced tree mode.

```
void SetNormalTreeMode()
void SetSplayTreeMode()
void SetBalancedTreeMode()
int  IsNormalTreeMode()
int  IsSplayTreeMode()
int  IsBalancedTreeMode()
```

Figure 4.10: RB_SpBinaryTree and RB_SpHashTree Mode Methods

Choosing which tree mode to use is data specific and can determine the performance of the tree. In normal tree mode, no rotation of the tree is performed. If the data being stored in the tree has no order before being inserted, normal tree mode is the best choice. In balanced tree mode, a heuristic is used to balance an unbalanced tree. This is a good mode to use if the data is somewhat ordered before being inserted or if the minimal or maximal element will always be removed from the tree. Splay tree mode always rotates the most recently inserted or accessed item to the top of the tree. If the tree is accessed in a random manner, this mode will yield poor performance. On the other hand, it can provide a caching-type feature if elements close to the recently accessed element will be accessed.

In general, a `RB_SpBinaryTree` should be used whenever a simulation object needs sorted data with $O(\log n)$ insertion and removal time. This makes the `RB_SpBinaryTree` a good general purpose data structure. An `RB_SpHashTree` should be used whenever it is not necessary to access the data in sorted order, but faster look ups and insertions of items are desired.

### 4.3.2 Rollbackable Lists

Class `RB_SpList` is ideal to use when items in the list do not need to be sorted. The definitions for this class are found in include file `RB_SpList.H`.

The methods available for modifying the contents of a list are shown in Figure 4.11.

```
enum {
  TOP_FLAG,
  BOT_FLAG
}
void  Insert(void* item, int TopBotFlag = BOT_FLAG)
void* Remove(void* item)
void* RemoveFirstElement()
void* RemoveLastElement()
```

Figure 4.11: RB_List Modifier Methods

Items may be inserted at the top or bottom of the list by specifying the location with the `TopBotFlag` in the `Insert` method (list classes insert at the bottom by default). `RemoveFirstElement` and `RemoveLastElement` can be used to quickly empty a list, while `Remove` can be used to remove a specific item from the list. If there is more than one item in the list matching the input argument, then the first such match is removed and no other matches are removed. When an item is removed and the list is empty, then `NULL` is returned. Method `Remove` performance grows as $O(n)$, but all the other list methods are $O(1)$.

Figure 4.12 shows the methods available for traversing a list.

```
int   GetNumElements()
void* GetFirstElement()
void* GetLastElement()
void* GetNextElement()
void* GetPreviousElement()
void* operator ++()
void* operator --()
```

Figure 4.12: RB_List Iterator Methods

The above methods are analogous to those in tree container class methods. Similarly, with the tree containers, adding or removing elements between `Get` methods calls can result in undefined behavior. Furthermore, iterating is not rollbackable.

Similar to all other container classes, `RB_SpList` stores all data as `void` pointers, thus leaving the user responsible for properly creating and deleting data that is inserted into the container.

### 4.3.3 Rollbackable Priority Trees

Class `RB_SpPriorityTree` is a specialization of the `RB_SpBinaryTree`, which behaves like a priority list. This means that items can be inserted into the container, but only removed from the front of the list or retracted through a retraction handle. Furthermore, the item at the front of the list is always of highest priority (i.e. lowest numeric priority value). The container can be searched for elements that match a known priority and, as with the `RB_SpBinaryTree`, duplicates are allowed and retain the priority of their insertion order. The definitions for this class are described in include file `RB_SpPriorityTree.H`.

The methods available for modifying the contents of a priority tree are shown in Figure 4.13.

```
SpRetractionHandle* Insert(void* item, double priority)
void*               Remove(double& priority)
void*               Remove()
int                 Retract(SpRetractionHandle* retractionHandle)
```

Figure 4.13: RB_SpPriorityTree Modifier Methods

The first method inserts an item with the specified priority and returns a retraction handle, which can be used later to retract that item from the container. The second method removes the item at the front of the list and returns its priority in the argument passed in, or returns `NULL` if the container is empty. The third method is the same, except that it does not return the priority by reference. The final method retracts an item that was inserted at an earlier point and returns 1 if the retraction was successful, and 0 otherwise.

The methods available for reading a priority tree are shown in Figure 4.14.

```
int    GetNumElements()
double GetCurrentPriority()
void*  Find(double priority)
```

Figure 4.14: RB_SpPriorityTree Reading Methods

Method `GetNumElements` returns the number of elements in the priority list. `GetCurrent-`

`Priority` returns the priority of the next item to be removed. `Find` returns the item which has the specified priority (`NULL` is returned if no item is found).

As with other container classes, this class stores all data as `void` pointers, thus leaving the user responsible for properly creating and deleting data that is inserted into the container.

### 4.3.4 Rollbackable Dynamic Pointer Arrays

`RB_SpDynPtrArray` behaves like an array of `RB_voidPtr`. The data structure is dynamic in the sense that the array size can grow whenever an element beyond the current end is accessed. The definitions for this are described in include file `RB_SpDynPtrArray.H`. Figure 4.15 shows the interfaces for the dynamic array.

```
RB_SpDynPtrArray(int arraySize = 2)
~RB_SpDynPtrArray()
RB_voidPtr& operator [](int index)
```

Figure 4.15: RB_SpDynPtrArray Interface

The default constructor allows for one optional integer argument, which specifies the initial array length. The array grows in size by factors of at least 2 whenever elements beyond the end are accessed using the overloaded `[]` operator. The growth is performed in a rollbackable fashion, as are assignments to elements of the array.

Here again, this class stores all data as `void` pointers, thus leaving the user responsible for properly creating and deleting data that is inserted into the container.

## 4.4 Independent Iterators

### 4.4.1 Introduction

As shown in Section 4.3, various container classes have methods for looping through all items held by an instance. These include `SpList`, `SpBinaryTree`, `SpHashTree`, and their rollbackable counterparts. The problem with all of these is the fact that, because the container keeps track of the current item, it can handle only one iteration at a time (e.g. a loop iterating through an `SpList` will not work properly if it contains a nested loop that traverses the same `SpList`). For this reason, it is strongly recommended that these looping facilities not be used.

What is needed is a separate object (for each iteration) to keep track of the current item. This is what an independent iterator is. The following is an example showing how an iterator can be used to loop through all the items in a container (in this case, an `RB_SpBinaryTree`):

```
RB_SpBinaryTree tree;
//... Add items to the tree

// Construct iterator so that it points to first item in tree.
SpIterator_RB_SpBinaryTree iterator(tree);
RB_cout << "Tree Items:" << endl;

TreeItem* item = (TreeItem *) iterator.GetFirstElement();
```

```
  while (item != NULL) {
    RB_cout << "Value= " << item->GetValue() << endl;
    item = (TreeItem *) iterator.GetNextElement(); // Get next item in tree
  }
```

Notes:

- There is a specialized iterator class corresponding to each container class. In this case, a `SpIterator_RB_SpBinaryTree` must be used to step through the elements in an `RB_SpBinaryTree`.
- The iterator class has a one-argument constructor that takes an instance of its corresponding container class. This builds the iterator and initializes it to point to the first element in the container (or to `NULL` if the container is empty).
- The `GetFirstElement` method returns a void pointer to the first item in the container, or `NULL` if there are none.
- The `GetNextElement` method advances the iterator to the next item in the container and returns a void pointer to the item. If the iterator is already located at the last item, it returns `NULL`.

#### 4.4.1.1 Iteration Order

There is an iteration ordering associated with each container class. This is the order in which an iterator visits items in a container. For lists, the ordering is determined by the way the items were inserted into the list (i.e. the sequence in which items were inserted and whether they were inserted at the top or bottom of the list). For binary trees, the keys determine the iteration order (by ascending key value). With hash trees, however, the iteration sequence is not defined this way. Its ordering is determined by an efficient traversal of the internal data structures. This order may appear random with respect to the keys, but is consistent in the following sense: for a given hash tree, repeated iterations will yield the same ordering as long as no new items have been inserted and no existing items have been removed from the object. Also, backwards iterations (e.g. going from the last item to the first) are consistent in that they visit items in the reverse order as the corresponding forward iteration (again assuming a fixed set of items in the hash tree).

#### 4.4.1.2 Multiple Simultaneous Iterations

As was mentioned above, the essential advantage of iterators is that they permit more than one iteration at a time on a container. The following example shows such a case: a nested loop searching through an `RB_SpList`.

Assume that each item in the list contains an integer. There is “special” pair of items in which the integer value of one is exactly double that of the other. There is only one such pair. The following function finds the pair (using nested iterator loops) and returns the two integers via reference parameters. The algorithm is not an efficient one and there is no error checking; it is only intended to demonstrate the use of nested iterator loops.

```
  void FindIntPair(RB_SpList& list, int& k, int& twoK) {
    SpIterator_RB_SpList outerIt(list); // outer loop iterator
    SpIterator_RB_SpList innerIt(list); // inner loop iterator
    // Get first item pointer.
    ListItem* item0 = (ListItem *) outerIt.GetFirstElement();
    ListItem* item1;

    while (item0 != NULL) {
      int i = item0->GetInt();

      // search entire list for 2 * i
      item1 = (ListItem *) innerIt.GetFirstElement();
      while (item1 != NULL) {
        int j = item1->GetInt();
        if (j == 2 * i) {
          k = i;
          twoK = j;
          return;
        }
        item1 = (ListItem *) innerIt.GetNextElement();
      }

      item0 = (ListItem *) outerIt.GetNextElement();
    }
  }
```

### 4.4.2 Iterator Interface

The various iterators are intended to have a very similar "look and feel." Figure 4.16 shows the interfaces common to all the iterators (except for `GetCurrentKey` and `GetCurrentKeyName`, which are not provided by the two list iterators, since lists do not use keys). In it, `<Container>` is a meta-symbol denoting one of the following container class names. Table 4.1 shows the container class names and their respective iterator class name.

```
SpIterator_<Container>()
SpIterator_<Container>(const <Container> &container)
SpIterator_<Container>(const <Container> &container, int TopOrBot)
void*          GetCurrentElement() const
operator void* () const
void*          GetFirstElement()
void*          GetLastElement()
void*          GetNextElement()
void*          GetPreviousElement()
void*          operator++()
void*          operator++(int)
void*          operator--()
void*          operator--(int)
// Note: the next 2 methods not provided by list iterators:
double         GetCurrentKey() const
char*          GetCurrentKeyName() const
```

Figure 4.16: Independent Iterator Interface

| Container Name | Iterator Name |
|---|---|
| SpList | SpIterator_List |
| RB_SpList | SpIterator_RB_SpList |
| SpBinaryTree | SpIterator_SpBinaryTree |
| RB_SpBinaryTree | SpIterator_RB_SpBinaryTree |
| SpHashTree | SpIterator_SpHashTree |
| RB_SpHashTree | SpIterator_RB_SpHashTree |

Table 4.1: Container and Iterator Cross Reference

#### 4.4.2.1 Constructing Iterators

The first three rows in Figure 4.16 show the iterator constructors. The one-argument constructor takes an instance of the corresponding container class and initializes the iterator to point to the first element in the container (see Section 4.4.1.1 for iteration ordering). To begin the iteration at the last element, use the two-argument constructor, passing `SpList::BOT_FLAG` as the second argument. The zero-argument constructor creates an iterator that points "nowhere" (i.e. its internal item pointer is `NULL`). It can only be used after it has been assigned the value of another iterator of the same type that is already linked to a container object. These ideas are illustrated in the example below:

```
SpList list;
// ...Insert items into list

SpIterator_SpList head(list);                  // Points to first item
SpIterator_SpList tail(list, SpList::BOT_FLAG); // Points to last item
SpIterator_SpList newIterator;                 // Points nowhere

newIterator = tail;                            // newIterator points to
                                               //  last item
```

#### 4.4.2.2 Accessing Information With Iterators

Once an iterator has been created and positioned, the `GetCurrentElement` method can be used to access the element that the iterator is currently pointing to. `GetCurrentElement` returns a `void*` pointer to the current item, which must then be cast to the correct pointer type. Each iterator has an overloaded `void*` operator. This is equivalent to `GetCurrentElement` and provides a convenient syntax for accessing the current item. Also, all the positioning methods and operators (discussed below) return a pointer to the current item, in addition to moving the iterator.

If the container class is one that uses keys (`SpBinaryTree`, `RB_SpBinaryTree`, `SpHashTree`, or `RB_SpHashTree`), the key associated with the current item can be retreived using either `GetCurrentKey` (for `double` keys) or `GetCurrentKeyName` (for string keys).

#### 4.4.2.3 Positioning An Iterator

There are several ways to move an iterator to different locations in the container's item sequence. `GetFirstElement` positions the iterator at the first item and returns a pointer to that item. `GetLastElement` positions the iterator at the last item and returns a pointer to that item. `GetNextElement` moves the iterator to the next item in the container and returns a pointer to that item. If the iterator is currently at the last item, `GetNextElement` moves the iterator past the end of the list (i.e. it points to NULL) and returns NULL. Subsequent calls just return NULL. GetPreviousElement moves the iterator to the prior item in the container and returns a pointer to that item. If the iterator is currently at the first item, GetPreviousElement moves the iterator off the beginning of the list (i.e. it points to NULL) and returns NULL. Subsequent calls just return NULL.

In the example below, an iterator is used to locate the last two items in a list:

```
RB_SpList list;
//... Add 5 items to list

SpIterator_RB_SpList iterator(list);

// Get last item and move iterator there.
ListItem *item = (ListItem *) iterator.GetLastElement();
RB_cout << "Last item Value= " << item->GetValue() << endl;

item = (ListItem *) iterator.GetPreviousElement(); // back up one element
RB_cout << "Next-to-last item Value= " << item->GetValue() << endl;
```

Iterators provide both prefix and postfix versions of the ++ and -- operators. The prefix version of ++ is equivalent to GetNextElement: it moves the iterator to the next item and returns a pointer to that item. The postfix version of ++ returns a pointer to the current item and moves the iterator to the next item. The prefix version of -- is equivalent to GetPreviousElement: it moves the iterator to the prior item and returns a pointer to that item. The postfix version of -- returns a pointer to the current item and moves the iterator to the prior item.

The example below is a function that prints the key and value of all items in a hash tree:

```
void S_TestObject::PrintHashTree(RB_SpHashTree& hash) {
  SpIterator_RB_SpHashTree iterator(hash);

  // Get first (current) item pointer.
  HashItem *hItem = (HashItem *) iterator.GetFirstElement();

  RB_cout << "Hash Keys and Vals:" << endl;
  while (hItem != NULL) {
    RB_cout << "Key= " << iterator.GetCurrentKey()
            << ", Value= " << hItem->GetValue() << endl;

    // Move to next item; return pointer.
    hItem = (HashItem *) iterator.GetNextElement();
  }
}
```

### 4.4.3 Finding an Iterator

As shown in Section 4.3.1, the RB_SpBinaryTree and RB_SpHashTree containers have a Find method that locates an item whose key is given (SpBinaryTree and SpHashTree also have this capability). Sometimes it is useful to examine other items in the container that are "near" the found item (especially in the case of binary trees, whose items can be traversed by increasing or decreasing key value). The container's FindIterator method is provided for this purpose. These methods are similar to the Find methods, except that instead of returning the element found, they return an iterator pointing to that element. This iterator can then be used to examine adjacent elements.

For example, suppose the Admiral object in a naval simulation stores its ships in a binary tree, keyed by ammunition level, as shown below:

```
SpBinaryTree ships;
ships.Insert("Enterprise", 34.0);
ships.Insert("Reliant"   , 28.0);
ships.Insert("Lexington" , 30.0);
ships.Insert("Galileo 7" , 12.0);
```

The Admiral needs two ships to attack a certain enemy target, each of which must have at least 30 units of ammunition. The following example shows how `FindIterator` can be used to help locate the ships. First, `FindIterator` finds the ship with 30 units of ammunition and returns an iterator pointing to that ship. Then, the iterator is moved to the next item in the binary tree. Since the iteration order goes by increasing key value, the next item must have 30 or more units of ammunition.

```
SpIterator_SpBinaryTree iterator = ships.FindIterator(30.0);
if (iterator.GetCurrentElement()) {
  cout << "Assigning ship " << (char*)iterator.GetCurrentElement()
       << " to attack new target." << endl;
  // Add code here which schedules attack event for this ship.
}
if (iterator.GetNextElement()) {
  cout << "Assigning ship " << (char*)iterator.GetCurrentElement()
       << " to attack new target." << endl;
  // Add code here which schedules attack event for this ship.
}
```

Notes:

- In the above example, the Admiral object knew that there was a ship with exactly 30.0 units of ammunition; that is how `FindIterator` was able to locate the first ship. If no item in the container has a key that precisely matches the argument, `FindIterator` returns an iterator pointing to `NULL`. As always, one must exercise extreme caution when doing exact comparisons of floating-point numbers.
- In general, a `FindIterator` call like the one above will be more efficient than using an iterator to search through the entire `SpBinaryTree`. The former does a binary search ($O(\log n)$ time complexity), while the latter does a linear search ($O(n)$ time complexity).

In the case of hash trees, it is less obvious why one might want to use `FindIterator`. Since the iteration ordering is based on the internal data structures, there is no way of knowing which items will be “in the vicinity” of a found item. There is, however, one exception to this rule: all items with the same key will be stored in a “contiguous block” of items, the first of which will be returned by `FindIterator`. Thus, to find all items with a given key, simply call `FindIterator` and then iterate forward until an item with a different key is found. This is shown below:

```
SpHashTree hTree;
// Add HashItems to hTree, using string keys

// Count #items whose key is "ABC"
SpIterator_SpHashTree iterator = hTree.FindIterator("ABC");
```

```
int count = 0;

while (iterator.GetCurrentElement()) {
  if (strcmp("ABC", iterator.GetCurrentKeyName())) // key != "ABC" {
    break;
  }
  else {
    count++;
  }
  iterator.GetNextElement();
}

cout << "#Items with KeyName ABC = " << count << endl;
```

## 4.5 Guidelines for Making Data Rollbackable

The most basic rule for when to make simulation object data rollbackable is: *Any data contained in the simulation object, which changes value from event to event, must be rollbackable*. This must include pointers, base data types, aggregated classes, and container classes. Making a variable rollbackable in the simulation object will never result in incorrect operation, although it does use CPU resources and a limited amount of memory resources.

Obviously, the use of a rollbackable variable must incur some overhead since states must be saved. However, users of SPEEDES need not overly concern themselves with these overheads, which are insignificant when compared to the amount of work done in processing a normal event. According to some brief tests, accesses of rollbackable variables cost, at most, a 10% overhead as compared to those of the non-rollbackable varieties. In most simulations, accessing state variables constitutes the majority of use. Writes to rollbackable variables, however, are expensive as compared to their non-rollbackable counterparts. For the simplest rollbackable variables (`RB_int` and `RB_double`), writes can incur as much as 10 times the overhead associated with the corresponding non-rollbackable version. The penalty is smaller for container classes. What this means is that users do not need to be overly concerned about inefficiencies incurred as a result of reading state variables. However, for optimal performance, writes to state variables should only occur on an as needed basis.

## 4.6 Tips, Tricks, and Potholes

- An `RB_int` cannot serve as an array index. The American National Standards Institute (ANSI) C++ standard states that only an `int`, `char`, `long`, or its unsigned version may be used as an array index. This means an `RB_int` cannot be used as an index to an array. The work around for this issue is to assign the `RB_int` to an `int` and index the array with the `int`. This "feature" of C++ will not be encountered with all compilers, only those that strictly follow the ANSI standard.

- Some compilers give warnings about casting a `RB_voidPtr` to a pointer type other than `void*`. A work around which removes this warning is:

```
class      bar;
bar*       someInstancePtr;
RB_voidPtr myVoidPtr;
...
someInstancePointer = (bar*)((void*)myVoidPtr);
```

- Rollbackable variables should only be used to represent state in a single simulation object. For this reason, rollbackable variables should never be used as:

  – parameters to a method or function.
  – local variables for a method or function.
  – global variables.
  – static data members of a class.

  As in the C++ memory management "rule of three" (*every class that uses dynamic memory needs a copy constructor, assignment operator, and destructor*), you must ensure that rollbackable variables (and classes that are aggregations of rollbackable variables) are not created as temporary variables. This can be accomplished by making the copy constructor and assignment operator private in a class containing rollbackable variables.

- The classic indicator that non-rollbackable variables have been used instead of state variables (i.e. rollbackable variables) is simulation non-repeatability. If the simulation produces different results during different runs, has no external interactions and is always running on the same number of nodes, then an event is probably getting rolled back, and errors are arising because that event's manipulation of the state is not being "undone". The amount of non-repeatability will vary and may itself not be repeatable.

- To sort a `RB_SpPriorityTree` in reverse order, either use negative priorities or specialize the class to encapsulate the use of negative priorities.

- To remember two different positions in a container, do not use the built-in iterators. Instead, use two different instances of an `SpIterator` class.

- If changes are made to a container (i.e. items added to or removed from the container), all iterators working on that container are considered invalid. There are several reasons for this, including the following:

  – Since the iterator might be pointing to a removed (and possibly deleted) item, returning a pointer to the item or repositioning the iterator relative to the item could produce ruinous consequences.
  – If an iterator is positioned at the first (or last) item and then a new item is inserted into the container, the iterator may no longer point to the first (or last) item.
  – If a new item is inserted before the current iterator position, the iterator will never visit that item.

  For these reasons, the only valid thing to do after the contents of a container have been altered is to reset the iterator so that it points to a legal position. This can be done by calling `GetFirstElement`, `GetLastElement`, or setting the iterator equal to another iterator that is currently valid.

- All the iterators inherit from a common base class (`SpIterator`) in which the positioning and accessing methods are declared `virtual`. This means that one can write functions that take an `SpIterator` argument and use it to perform some operation on items in the container, without having to specify the exact iterator or container type. For example, if all container items have a `virtual Print` method, then one could write such a function that prints all items in a container. This one function could be passed any iterator object, and thus could be used with any of the `SpIterator` container classes (lists, binary trees, or hash trees).

- None of the iterators are rollbackable. Although some contain "`RB`" in the class name (e.g. `SpIterator_RB_SpHashTree`), this means that instances iterate over a rollbackable container, not that the iterator itself is rollbackable. Therefore, if an iterator is advanced during the processing of an event (for example), and that event is rolled back, then the iterator position will not be changed back to the old position.

# Chapter 5

# Utilities

This chapter discusses a number of additional utilities that are useful in creating simulation objects, managing their state, and evaluating their performance.

## 5.1 Rollbackable Memory Management

As stated in Chapter 4, all attributes on a simulation object must be rollbackable, if the attributes contain state information (i.e. must be rollback safe). The same is true of dynamically allocated memory. Dynamic memory management must be handled in a rollbackable fashion in order to avoid memory leaks (multiple calls to `new` due to rollbacks) and heap corruption (multiple calls to `delete` due to rollbacks). These issues arise in addition to the usual C++ concerns regarding good memory management practices.

Macro `RB_DEFINE_CLASS` (available in `RB_SpDefineClass.H`) generates several functions and classes useful for dynamic memory management. For example, if class `foo` is passed in as the argument to macro `RB_DEFINE_CLASS`, the following functions are created:

```
RB_DEFINE_CLASS(foo);
foo*  RB_NEW_foo();                // Replaces new foo
foo*  RB_NEW_ARRAY_foo(int size); // Replaces new foo[size]
void  RB_DELETE_foo(foo*);        // Replaces delete foo
void  RB_DELETE_ARRAY_foo(foo*);  // Replaces delete [] foo
class RB_PTR_foo;                 // Replaces foo*
```

The functions and classes output by the macro serve as the rollbackable counterparts to ordinary `new` and `delete` C++ constructs. A call to the `RB_NEW_foo` function from within an event will allocate memory. In case of an event rollback, memory previously allocated will be deleted. Similarly, calls to `RB_DELETE_foo` do not delete memory until the event is committed. Failure to use these functions in events will almost certainly result in unexpected behavior due to memory leaks and heap corruption.

The extra overhead incurred when using these rollbackable functions, rather than the ordinary `new` and `delete`, is small relative to the overall cost of memory management calls. These functions should also be used whenever dynamic memory is added to a rollbackable container class.

Class `RB_PTR_foo` serves as an easy-to-use, rollbackable replacement for a `foo` pointer. `RB_PTR` attributes handle updates to its value in a rollbackable fashion. The class `RB_PTR_foo` is superior to `RB_voidPtr`, since it requires no type casting.

## 5.2 Rollbackable Random Number Generator

A simulation is considered to be repeatable when, for fixed initial states and random number seeds, the mapping from inputs to outputs is well-defined. In other words, a simulation, when run twice with the same initial states and random number seeds, will produce identical results both times. In order to guarantee repeatable simulations, random number generators need to be rollbackable.

SPEEDES provides a rollbackable random number generator that is accessible through the global function `SpGetRandom` in `SpGlobalFunctions.H`. This random number generator is built into every simulation object and its seed can be set with the operator `RB_SpRandom::SetSeed(int seed)`. The seed inside the `RB_SpRandom` class is an `RB_int` and can be set at anytime during a simulation without affecting the rollbackable nature of the `RB_SpRandom` class.

Figure 5.1 shows the public interface for class `RB_SpRandom` and `RB_SpFastRandom`.

```
class RB_SpRandom {
  public:
    double GenerateUniform();                  // Uniform between [0, 1]
    int    GenerateInt(int l, int h);          // Uniform int
                                               // between [l, h]
    double GenerateDouble(double l, double h); // Uniform double
                                               // between [l, h]
    double GenerateExponential(double t);      // Exponential distribution
    double GenerateLaplace(double timeConst);  // Laplace distribution
    double GenerateRayleigh(double alpha);     // Rayleigh distribution
    double GeneratePower(double p);            // (p + 1) * x ˆ p,
                                               //  where 0 <= x <= 1
    double GenerateReversePower(double p);     // (p + 1) * (1 - x) ˆ p
                                               //  where 0 <= x <= 1
    double GenerateTriangleUp();               // f = 2 * x
                                               //  where 0 <= x <= 1
    double GenerateTriangleDown();             // f = 2 * (1 - x)
    double GenerateBeta(int m, int n);         // (m + n + 1)! / (m! * n!)
                                               // * [t ˆ n * ( 1 - t ˆ n)]
    double GenerateGaussian(double mean, double sigma);
                                               // Gaussian distribution
    double GenerateDensityFunction(SpDensityFunction* densityFctn);
                                               // User-defined
                                               //  distribution
    void   GenerateVector(double vector[3], double magnitude = 1.0);
                                               // Random uniform position
    double GenerateCauchy(double alpha);       // Cauchy distribution
}
```

Figure 5.1: Random Number Generator

The class `RB_SpRandom` implements a high fidelity random number generator that generates random numbers through the computation of random bits. This is an expensive calculation but provides much better random numbers than those output by a linear congruential generator. A down side of this generator is that there are only approximately $250,000$ random numbers available before they are recycled. For a faster random number generator (at the cost of lower fidelity randomness), the class `RB_SpFastRandom` is also defined in the header file `RB_SpRandom.H`. This class provides exactly the same interface but has a longer cycle before numbers repeat and generates random numbers much faster. In general, if models do not need random numbers which are "truly" random, then using the class

`RB_SpFastRandom` is often a good choice. Otherwise, use class `RB_SpRandom`.

User-defined distributions can be added using the density function. To accomplish this, inherit from `SpDensityFunction` and implement the required pure virtual methods. Figure 5.2 shows the interface for this class.

```
class SpDensityFunction {
  public:
    virtual double f(double x) = 0;        // Density function
    virtual double GetMaxAmplitude() = 0; // Max amplitude of function
    virtual double GetLoLimit() = 0;      // Minimum value of function
    virtual double GetHiLimit() = 0;      // Maximum value of function
}
```

Figure 5.2: Density Function

## 5.3 Other Rollbackable Functions

The following subsections discuss a number of rollbackable functions that provide functionality which emulates many standard C and C++ standard function calls or provides additional framework capability. These function prototypes are defined in include file `RB_SpFrameworkFuncs.H`.

### 5.3.1 Rollbackable Assert

The standard C library function `assert` prints an error message and calls abort if its argument is false. SPEEDES provides a rollbackable function called `RB_assert`, which performs similarly, but is not called until the event is committed. This differs from `assert` in that `assert` would abort the program immediately.

```
  void RB_assert(assertion) // Aborts if assertion is false
                            //  (i.e. aborts if assertion = 0
```

### 5.3.2 Rollbackable Memory Copy and String Duplication

Functions `RB_memcpy` and `RB_strdup` are rollbackable versions of the C standard functions `memcpy` and `strdup`, respectively. The API for these functions are:

```
  void* RB_memcpy(char* destination, char* source, int size)
  char* RB_strdup(const char* source)
```

`RB_memcpy` copies `size` bytes from memory area from `source` to `destination`. The copy will be undone if the event is rolled back.

`RB_strdup` makes a copy of the string `source` on the heap using `new` in a rollbackable fashion. Notice that this is different from the C library function `strdup`, which allocates and deletes memory using `malloc` and `free`. For `RB_strdup`, `RB_DELETE_ARRAY_char` must be used delete the memory in a rollbackable fashion.

## 5.4 Creating New Rollbackable Functions or Objects

Sometimes it may be necessary to create your own rollbackable function or objects. For example, you may wish to provide a rollbackable function for committing data to a database or writing data to a disk. SPEEDES provides a simple means to create new rollbackable functions or objects.

In order to create a custom rollbackable object, an Alterable Item (AltItem) for the function or object must be created. The class being designed must inherit from `SpAlt` and must implement the following pure virtual methods:

```
class SpAlt {
  public:
    void  Alter();
    virtual void  Cleanup() = 0;
    virtual void  Rollback() = 0;
    virtual char* GetType() = 0;
    virtual int   CheckUseOfMemoryRange(void* basePtr, int size) = 0;
};
```

- `Alter`:
  Initializes the AltItem. This method needs to be called anytime changes are made to a newly created rollbackable item. This method will save off the old data for the item so that it can be restored during rollbacks.

  Note: This method is not actually on class `SpAlt`. Users must supply a method that performs the same functions as described above. As part of the implementation for the new rollbackable function, users must call this method as shown in Example 5.1. By convention, this method is called `Alter` for all rollbackable functions contained in SPEEDES.

- `Cleanup`:
  Performs any action required during the commit phase of an event (see Chapter 15 for more detail on the event phases). For example, in the case of `RB_cout`, the `Cleanup` method prints the input data to the screen.

- `Rollback`:
  Returns the function or class back to the state it was before the function was executed or the class changed. Method `Rollback` should be implemented in such a way that it correctly handles rollbacks and also handles the rollforward condition (i.e. used when lazy is enabled (see Section 15.2.1).

- `GetType`:
  Returns the name of this AltItem.

- `CheckUseOfMemoryRange`:
  Identifies misuse of rollbackable variables. This method returns 0 if the memory between the range of `basePtr` and `basePtr + size` was modified. Additional information on how to implement this method is provided in the “SPEEDES API Reference Manual”.

Let us examine these methods more closely with the following example. Suppose a simulation needs to output data to a database during execution and the following function will handle the output of the data:

```
void WriteToDatabase(tmData);
```

The input parameter has a type of `Telemetry`. Every time function `WriteToDatabase` is called, the data currently contained in `tmData` is saved off into the database. If this function is called inside of an event, then the data is immediately written to the database. Since events are executed optimistically, telemetry data could be written to the database prematurely. If the event is rolled back and data was prematurely written to the database then there will be invalid data in the database. Let us assume for our application that only "committed" data in our database is acceptable. What we need is a function which only outputs the data once we are assured that the data can no longer be rolled back. This can be achieved by creating a rollbackable function, `RB_WriteToDatabase`. The first step is to write the rollbackable function `RB_WriteToDatabase`, as shown if Example 5.1.

```
// RB_WriteToDatabase.C
#include "Telemetry.H"
#include "SpAltTelemetry.H"

extern void WriteToDatabase(Telemetry& tmData);

void RB_WriteToDatabase(Telemetry& tmData) {
  SpAltTelemetry* altItem;                // Defined in SpAltTelemetry.H

  if (SpCurrentAltMgr->GetOptimistic() {
    altItem = ALLOCATE_SpAltTelemetry(); // ALLOCATE_SpAltTelemetry
                                         //  defined by macro
                                         //  DEFINE_MEMPOOL in
                                         //  SpAltTelemetry.H
    altItem->Alter(tmData);
    SpCurrentAltMgr->Insert(altItem);    // Save changes for cleanup
                                         //  or rollback
  }
  else {
    WriteToDatabase(tmData);
  }
}
```

Example 5.1: Rollbackable Function Definition (RB_WriteToDatabase)

The basics here are that, if the simulation is running optimistically, then the current data passed into the method is saved in the `SpCurrentAltMgr`, which is a class contained on every event. When events are executed and changes are made to rollbackable data, then this data is saved in `SpCurrentAltMgr`. If the simulation is not running optimistically (i.e. simulation is executing on one node and the parameter `optimize_sequential` is set to true in `speedes.par`), then the data is immediately written to the database (i.e. function `WriteToDatabase` is called). This is because the event cannot be rolled back.

Next, class `SpAltTelemetry` has to be implemented, as shown in Example 5.2. The required methods of `Alter` `Cleanup` `Rollback` `GetType`, and `CheckUseOfMemoryRange` are implemented in this class.

```
// SpAltTelemetry.H
#ifndef SpAltTelemetry_H
#define SpAltTelemetry_H

#include "SpAlt.H"
#include "Telemetry.H"
extern void WriteToDatabase(Telemetry& tmData);
```

```
class SpAltTelemetry : public SpAlt {
  public:
    void Alter(Telemetry& tmData) {
      AltItemSet = 1;
      MytmData   = tmData;
    }
    virtual void Cleanup() {
      if (AltItemSet == 1) {
        WriteToDatabase(MytmData);
      }
    }
    virtual void Rollback() {AltItemSet = !AltItemSet;}

    // See SPEEDES API Reference Manual for the following two methods
    virtual char* GetType() {return "SpAltTelemetry"; }
    virtual int   CheckUseOfMemoryRange(void* baseAddr, int size) {
      return 1;
    }

  private:
    int       AltItemSet;
    Telemetry MytmData;
};
// Generates function ALLOCATE_SpAltTelemetry
DEFINE_MEMPOOL(SpAltTelemetry, ALT_CHUNK_SIZE);
#endif
```

Example 5.2: Alterable Item Class Definition (SpAltTelemetry.H)

Notice that method `Alter` saves the current data. Once GVT has increased to a point that RB_`WriteToDatabase` can no longer be rolled back, then method `Cleanup` is called. This then writes the telemetry data to the database. Method `Rollback` is implemented so that it works correctly in both rollback and rollforward conditions. If the event is rolled back, then the `AltItemSet` is set to 0, which will prevent function `WriteToDatabase` from being executed in method `Cleanup`. If the event is rolled forward, then method `Rollback` is executed again, which will set `AltItemSet` back to 1.

The macro called `DEFINE_MEMPOOL` has been introduced above. This macro serves a similar function for creating AltItem as `DEFINE_SIMOBJ` does for creating simulation objects. In order to maximize run-time performance, SPEEDES avoids using `new` and `delete` regularly. Instead, for dymamic memory management, SPEEDES pre-allocates memory in large chunks and stores this memory off for later use. Memory is retrieved from this storage as needed (e.g. `ALLOCATE_SpAltTelemetry` in Example 5.1). After the event has been committed, the memory used is no longer needed and is returned to its storage by the SPEEDES framework.

Finally, the AltItem just created must be plugged in. This is done with `PLUGIN_MEMPOOL` which serves a similar purpose as `PLUGIN_SIMOBJ` and `PLUGIN_EVENT`. The plug-in macro plugs the new AltItem into the SPEEDES framework. The code for this is shown in Example 5.3.

```
  // SpAltTelemtry.C
  #include "SpMempool.H"
  #include "SpAltTelemetry.H"

  PLUGIN_MEMPOOL(SpAltTelemetry);
```

Example 5.3: Alterable Item Plug-In (SpAltTelemetry.C)

Now, anytime the telemetry data needs to be recorded in the database, simply call to the new rollbackable function `RB_WriteToDatabase`. The data will then be written to the database at the appropriate time.

Changes similar to those shown above can be applied to a class, in order to create a rollbackable version of a non-rollbackable class. This has already done for SPEEDES built-in classes like `RB_int`, `RB_double`, `RB_SpBinaryTree`, etc. While an example is not shown here, an example is described in the “SPEEDES API Reference Manual”.

## 5.5 Parameter File Parsing

SPEEDES provides a powerful parser that is useful for making run-time changes to the simulation, rather than hard coding values. This parser can be used to read in initial values for simulation objects Several terms need to be defined for the purpose of this section:

- Parameter file:
  This file contains the values for various simulation parameters to be set at run time. The file must be written using a particular format, which will be discussed shortly.

- Set:
  The fundamental recursive data structure containing zero or more elements. Class (`SpSet`) operates on sets.

- Data parser:
  An object that converts a parameter file to a set. This set is given the name “Top Level Set”. Class (`SpParser`) builds this top level set.

- Element:
  A set, integer, integer array, real, real array, string, string array, logical, or logical array.

A parameter file can have a fairly generic structure using brace notation. Curly braces, “{” and “}”, delimit the beginning and end of sets. Note that the entire parameter file is considered to be the top level set. White space is generally ignored, except when text is to be enclosed within double quotes. Figure 5.3 shows an example parameter file.

```
// Example parameter file.
// The "//" is a comment in a parameter file.
Set_0 {                              // Set 0 definition
  int SetZerosInt          4         // Integer definition
  int SetZerosDefaultValue 6         // Integer definition
}

Set_1 {                              // Set 1 definition
  Set_4 {                            // Set 4 definition
10      inherit Set_0                    // This set also contains set 0
11      int     SetZerosDefaultValue 17 // Overrides SetZerosDefault Value
12      int     setFoursInt 6           // Integer definition
13    }
14    Set_5 {                           // Set 5 definition
15      Set_6 {                         // Set 6 definition
16        include   "SetSixIncludedValues.par"
17                                      // Set 6 contains all of the
18                                      //  definitions found in
19                                      //  SetSixIncludedValues.par
20        reference Set_2
21        string    myString "hello"    // String array definition
22          "Good bye"
23      }
24      int     firstInt 4              // Integer definition
25      float   double1  6.4 1e12       // Float array definition
26      logical boolean1 f F t T        // Boolean array definition
27      int     x        1 2 3          // Integer array definition
28    }
29  }
30
31  Set_2 {                             // Set 2 definition
32    int SetTwosInt 4                  // Integer definition
33    Set_3 {                           // Set 3 definition
34      string string1 StringsWithOutSpacesDoNotNeedQuotes
35                                      // String entry with one element
36      string string2 "String" "Array" "Requires" "Quotes"
37                                      // String array definition
38      string string3 "As well as strings with spaces"
39                                      // String with spaces
40    }
41  }
```

Figure 5.3: Example Parameter File

| Line Number(s) | Description |
|---|---|
| 1 - 2 | Comment characters are "//". All text following characters are ignored. |
| 3 | Definition for set `Set_0`. Set names can use any alphanumeric and underscore characters. They must start with an alphabet letter. The set name must be followed by an open bracket "{". |
| 4-5 | Two integer parameters defined for `Set_0`. |
| 6 | Closing bracket for `Set_0`. This ends the definition for `Set_0`. |
| 7 | Blank line. White space is allowed in parameter files. |
| 8 | `Set_1` definition. |
| 9 | `Set_4` definition contained inside `Set_1`. Sets can be composed of other sets. This set is not accessable by other sets outside of `Set_1`. |
| 10 | `Set_4` inherits parameters defined in `Set_0`. Therefore, parameters `SetZerosInt` and `Set-ZerosDefaultValue` are defined in `Set_4`. |
| 11 | This line overrides the `SetZerosDefaultValue` from `6` to `17`. |
| 12 | A new integer parameter definition. |
| 13 | Closing bracket for `Set_4`. |
| 14 | `Set_5` definition. |
| 15 | Yet another embedded set definition for `Set_6`. |

| Line Number(s) | Description |
|---|---|
| 16 | All of the contents from file `SetSixIncludedValues.par` is read and place at this location in `Set_6`. |
| 17-19 | Comments. |
| 20 | `Set_2` is placed here. When the keyword `reference` is used, then the entire set is pulled in, including the set name `Set_2` (unlike `include`, which only pulls in the contents of the set). Also, `reference` can be used in advance of the actual set definition. |
| 21-22 | String array parameter definition. String arrays must have double quotes around each element in the array. Data values do not have to be on the same line as shown. |
| 23 | Closing bracket for `Set_6`. |
| 24 | Integer parameter added to `Set_5`. |
| 25 | Float array parameter added to `Set_5`. Data values for floats can use either the dot or exponential notation. Like string arrays, parameter data values can be on new lines. |
| 26 | Logical array parameter added to set `Set_5`. Logicals behave like type `bool`. Data values for logicals can be "f", "F", "t", or "T". Like string arrays, parameter data values can be on new lines. |
| 27 | Integer array parameter added to `Set_5`. Like string arrays, parameter data values can be on new lines. |
| 28-29 | Closing bracket for `Set_5` and `Set_1`. |
| 31 | `Set_2` definition. |
| 32 | Integer parameter definition. |
| 33 | Embedded set `Set_3` definition. |
| 34 | String parameter definition. If string does not contain spaces the quotes are optional. |
| 36 | String array parameter definition. Quotes must be used to indicate array data elements. |
| 38 | String parameter definition. Strings that contain spaces must be enclosed in quotes. |
| 40-41 | Closing bracket for `Set_3` and `Set_2`. |

Table 5.1: Parameter File Description

### 5.5.1 Parameter File Language Overview

#### 5.5.1.1 Sets

Sets are the fundamental unit of data collection in a parameter file. In fact, a parameter file is simply a succession of keyword declarations followed by a succession of sets.

Set syntax consists of an identifier representing the set name, followed by a braced set of parameters (or followed by nothing if it is a reference). Sets are comprised of the following:

- nested sets
- inherited sets
- referenced sets
- integers
- integer arrays
- floats
- float arrays

- logicals (booleans)
- logical arrays
- strings
- string arrays

#### 5.5.1.2 Parameters

The most basic element of a set consists of integer, float, boolean, and string definitions. Sets use the keywords “int”, “float”, “logical”, and “string” as type specifiers for each parameter. For example:

```
MySet1 {
  float x 72   // Valid
  float x z    // Error, ’z’ is not a floating point number
  int   y 77.6 // Error, 77.6 is not an integer
}
```

Parameters can also be specified as arrays. Integer, float, and logical arrays are simply space delimited data values. Strings arrays data values are not space delimited but each data value must be inside double quotes. For example:

```
MySet2 {
  int     iArray1 1 2 7 8         // This is a valid int array
  int     iArray2 3 5
                  2               // Arrays can be multi-lined
  int     iArray3 9 5 4.2 3       // Not valid since 4.2 is not an int
  float   fArray1 4.2 3.7 5 6.7   // Valid float array.
                                  //  5 is promoted to a float.
  logical lArray1 f F t T         // Valid logical array
  logical lArray2 f F t T s       // Invalid logical character ’s’
  string  sArray1 "Valid" "Array" // Valid string array
  string  sArray2 "This"          // Valid string array with
                  "is a"          //  4 elements
                  "Valid"
                  "Array"
}
```

#### 5.5.1.3 Set Inheritance

A set may inherit another set’s parameters and its values (including all nested sets). This is achieved by the user by using the reserved keyword “`inherit`”, followed by the set name to be inherited. When a set is inherited, all of the parameters become a part of the new set, however the outer set name is not part of the set. For example, consider the following parameter file:

```
BaseSet {
  BaseSetSubSet {
    int parameter_0 17
    int parameter_1 289
  }
```

```
    int parameter_2   4913
  }

  DerivedSet {
    inherit BaseSet
    BaseSetSubSet {
      int parameter_1 83521 // Override parameter_1 value.
                            //  parameter_0 remains unchanged
    }
    int parameter_3   1457  // Extend BaseSet with more elements
  }
```

Set `DerivedSet` contains all of the parameters in `BaseSet` (i.e. set `BaseSetSubSet` and parameters `parameter_0`, `parameter_1`, and `parameter_2`), but not the actual set name `BaseSet` itself. This is unlike `reference` which also will include the set name. Also, in order to inherit a set, it must be previously defined in the parameter file. In addition, any inherited parameter's value may be overridden by simply restating the element with its new value. For overriding subparameter values, the appropriate nesting must be replicated, as shown above with `BaseSetSubSet`.

#### 5.5.1.4 Set References

A set may reference another set's parameters and its values (including all nested sets). This is achieved by the user by using the reserved keyword "`reference`", followed by the set name to be referenced. When a set is reference, all of the parameters become a part of the new set including the reference set name (unlike inherit describe above). Some differences between inheriting a set and referencing a set include:

- Values of elements in referenced sets cannot be overridden, while values of elements in inherited sets can.
- Inherited sets must be fully defined before inheriting, while referenced sets can appear later but still in scope.

#### 5.5.1.5 Set Includes

A set may include another parameter file. This is achieved by the user by using the reserved keyword "`include`", followed by the name of the parameter file in quotes. The only limitation is that included sets can only be referenced (not inherited).

### 5.5.2 General Usage

The `SpParser` has only a few methods. The actual work is performed in the `SpSet` (see Section 5.5.3). The API for `SpParser` is shown below:

```
  SpParser(char*    filename)
  SpParser(istream* infile)
  SpSet* Parse(SpSet* set = NULL)
```

The two constructors allow construction either from a filename or from an already created stream. If the first constructor is used, SPEEDES first tries to open the file specified in the current directory, or in the location specified if `filename` is an absolute pathname. If the file is not found, then the parser will check in the directory specified by environment variable `SPEEDES_PAR_PATH`. If, after all these tries, the file cannot be opened, an error message is printed and the method returns. There is no destructor for the `SpParser`. Therefore, all sets that were allocated by the parser must be deleted separately.

After a `SpParser` has been created, a set is created by calling method `Parse`. Method `Parse` returns a set, which class `SpSet` operates on in order to retrieve the set contents.

### 5.5.3 SpSet Usage

Once a file or stream has been parsed, the data can be accessed through the returned `SpSet`. `SpSet` is a flexible class containing a large number of methods. Each type of integer, float, logical, and string has an equilent accessor method for extracting parameter data values from a set, as shown in Figure 5.4.

```
int     GetInt(char* key, int defaultValue, int& status)
int     GetInt(char* key)
double  GetFloat(char* key, double defaultValue, int& status)
double  GetFloat(char* key)
bool    GetLogical(char* key, bool defaultValue, int& status)
bool    GetLogical(char* key)
char*   GetString(char* key)
int*    GetIntArray(char* key, int& length)
double* GetFloatArray(char* key, int& length)
char**  GetStringArray(char* key, int& length)
bool*   GetLogicalArray(char* key, int& length)
```

Figure 5.4: Set Accessor Methods

The non-array `Get` methods return the value represented by `key`, or the `defaultValue` if `key` is not found. `GetString` differs in that it returns a pointer to the string if found, and `NULL` otherwise. Do not delete the string that is returned and be sure to copy the data out of the string if it is to live beyond the scope of the `SpSet`.

The array methods return a pointer to the private array of values, and set the length of the array to `length`. Do not delete or modify the values that are pointed to by the return value. The arrays also disappear when `SpSet` is deleted. So, copy the values if the elements will be modified or must live beyond the scope of the `SpSet`. If `key` is not found, a `NULL` pointer is returned.

In addition to the set accessor methods, there are methods provided that allow users to navigate through sets, which are shown in Figure 5.4.

```
  SpSet*  GetSet(char* name)
  SpSet*  GetParent()
  SpSet*  GetAncestor(char* setName)
  SpSet*  GetAncestor(int numLevels = 1)
  SpSet*  Root()
  int     GetNumSets()
  SpSet*  GetFirstSet()
  SpSet*  GetNextSet()
  int     GetNumElements()
```

Figure 5.5: Set Traversal Methods

- `GetSet`:
  Returns the set contained in the current set with the given name. Returns `NULL` if it is not found.
- `GetParent`:
  Returns the parent of the current set. If there is no parent set, then `NULL` is returned.
- `GetAncestor`:
  Returns the ancestor of the current set with either the given name or the given number of levels up. If the given ancestor is not found, `NULL` is returned.
- `Root`:
  Returns the top level set for the given set.
- `GetNumSets`:
  Returns the number of sets contained in the given set.
- `GetFirstSet`:
  Returns the first set within the current set.
- `GetNextSet`:
  Returns the next set with the current set.
- `GetNumElements`:
  Returns the number of parameters in the given set.

### 5.5.4 An Example

Let us consider a case where submarine simulation objects are to be initialized based on an input parameter file called `submarine.par`. The `submarine.par` will use inheritence and references in order to reduce the complexity of the par file. The first set definition in file `submarine.par` is a set called `SimulationEntity`, which will represent the base set for all other simulation objects. It contains all of the common parameter for all simulation objects.

```
// submarine.par

SimulationEntity {
  int    allegence  0        // Team number id
  int    type      -1        // Should be overridden
  double health     1        // Always start off with full health
  string name       "NoName" // Name should always be assigned
}

Objects {
  reference Nautilus
  reference Hunley
  reference RedOctober
}

BasicSubmarine {
  inherit SimulationEntity
  int     type             17 // Override type
  int     EngineType      -1  // -1=Unknown, 0=human, 1=electric,
                              //  2=diesel, or 3=nuclear
  int     ConstructionYear 0
  double  LengthInMeters  -1
}

Nautilus {
  inherit BasicSubmarine
  string  name             "Nautilus"
  int     allegence        -1   // Rogue ship, no team
  int     EngineType       1
  int     ConstructionYear 1870
  double  LengthInMeters   71.4
}

Hunley {
  inherit BasicSubmarine
  string  name             "Conferate Ship Hunley"
  int     EngineType       0
  int     ConstructionYear 1863
  double  LengthInMeters   10.2
}

RedOctober {
  inherit BasicSubmarine
  string  name             "Red October"
  int     EngineType       2
  double  LengthInMeters   171
  int     ConstructionYear 1984
}
```

Figure 5.6: Parameter File submarine.par Example

Suppose that the above `submarine.par` will initialize a submarine simulation object. The code shown in Example 5.4 shows one potential implementation for the submarine's `Init` method.

```
// S_Submarine Init Method Implementation
void S_Submarine::Init() {
  SpParser* subPar;
  SpSet*    rootSet;
  SpSet*    objectSet;
  SpSet*    mySet;
  int       i;

  subPar    = new SpParser("submarine.par");
  rootSet   = subPar.Parse();
  objectSet = rootSet->GetSet("Objects");

  if (objectSet == NULL) {
    cout << "**Error, no set named Objects in submarine.par!" << endl;
    exit(-1);
  }

  mySet = objectSet->GetFirstSet();
  for (i = 0; i < SpGetSimObjKindId(); ++i) {
    mySet = objectSet->GetNextSet();
  }
  /*
   * Now let the parent set initialize its values from this set.
   * It will set the values of allegence, type, health, and name.
   */
  S_SimulationEntity::Init(mySet);

  SetEngineType(mySet->GetInt("EngineType"));
  SetConstructionYear(mySet->GetInt("ConstructionYear"));
  SetLengthInMeters(mySet->GetFloat("LengthInMeters"));
}
```

Example 5.4: Submarine Initializaton from .par File

## 5.6 Tips, Tricks, and Potholes

- Some compilers, notably the Silicon Graphics, Incorporated C++ compiler, gives an error when the macro `RB_DEFINE_CLASS(foo)` is called, where `foo` is a primitive type such as a `int`, `char`, or `double`. This behavior is due to the overloaded `->` operator in the `RB_PTR_foo` class. To avoid these problems, the macro `RB_DEFINE_CLASS_PRIMITIVE(foo)` can be used when the compiler gives errors.

- When memory is deleted by a call to `RB_DELETE`, it is not actually deleted at the point of the function call. Instead, memory is deleted during the event commit phase. `RB_DELETE` is implemented in this manner because the event which called this function may be rolled back and the memory not deleted.

- Always use `RB_NEW` and `RB_DELETE`, even in simulation object constructor, destructor, and `Init` methods. There is virtually no overhead for use of these functions at this time, and their use will avoid other problems when persistence or checkpoint/restart is used.

- `RB_DEFINE_CLASS` does not make a class rollbackable; it merely creates the functions and classes which can `new` and `delete` classes in a rollbackable fashion (see Section 5.1).

- Some classes are difficult to make rollbackable for a host of reasons such as non-standard data members (e.g. bitfields) or because access to class internals is not possible (e.g. third party software). If a class does not contain dynamic memory, it can be made rollbackable by using `RB_memcpy` in the copy constructor and assignment operator. For example, consider the class which has two bit fields with one being 12 bits long and the other 20 bits long.

```
class MyStuff {
  public:
    MyStuff(int f1, int f2) : field1(f1), field2(f2) {}
    /*
     * Copy constructor
     */
    MyStuff(const MyStuff& rhs) {
      RB_memcpy(this, (char *) &rhs, sizeof(MyStuff));
    }
    /*
     * Assignment operator.
     */
    MyStuff& operator = (const MyStuff& rhs) {
      RB_memcpy(this, (char *) &rhs, sizeof(MyStuff));
      return *this;
    }
    void SetField1(int f1) {
      MyStuff temp;
      temp.field1 = f1;
      temp.field2 = field2;
      *this = temp;
    }
    void SetField2(int f2) {
      MyStuff temp;
      temp.field1 = field1;
      temp.field2 = f2;
      *this = temp;
    }
    int GetField1() {return field1;}
    int GetField2() {return field2;}

  private:
    int field1:12;
    int field2:20;
};
```

Example 5.5: Rollbackable Class using RB_memcpy

Using `RB_memcpy` can make classes rollback safe. However, implementing a class, as shown above, is not as optimal as creating a class with `SpAlt`.

- If memory is created using `RB_NEW_ARRAY_foo`, it must be deleted using `RB_DELETE_ARRAY-_foo`. Using different functions will result in undefined behavior.

- An rollbackable variable should never be used as a local variable or as a non-reference argument to a method. Doing so creates a rollbackable variable on the stack. When a rollbackable variable is locally in a function or method, then AltItems are created for information which will become obsolete when the method is exited. However, if the event is rolled back then an attemp is made to restore the data. Hence, stack data can be overwritten. An error message, as shown in Figure 4.1, will be displayed when a rollbackable variable is used incorrectly.

- If any pointer values are obtained from a `SpSet`, especially when the set comes from the `SpParser`, the data should be copied if the `SpSet` will be deleted. Also, deleting the `SpParser` will not delete the associated set.

# Part III

# Events

Chapter 6

# Point-to-Point Events

Point-to-point events are a direct mapping to the way the SPEEDES event management system processes events. That is, every event in SPEEDES represents one action scheduled by one simulation object, to act on one other simulation object, at one point in simulated time. The term “point-to-point” refers to the fact that events are one-way actions scheduled at one point (the scheduling simulation object), to be sent to act on another point (the scheduled simulation object).

There are three API styles for point-to-point events: simulation object events, local events, and autonomous events. In each case, the code to be invoked (i.e. the callback) for the event is implemented as a method that acts upon a simulation object. That method may be directly on the simulation object, contained within the simulation object, or on a separate object that acts on the simulation object. Where the method resides distinguishes the event’s style. A simulation object event method resides directly in an object inheriting from `SpSimObj` or one of its children (e.g. `S_SpHLA`). A local event method resides in any object in the simulation and must be self-scheduled. This means a simulation object must only schedule local events for methods on its nested sub-objects. An autonomous event method resides on an object that is separate from the simulation object on which it acts. All three of these event types will be described in the sections that follow.

Each of the three API point-to-point method styles serves the basic purpose of enabling time-tagged, one-way event invocations. Understanding the differences between the different event styles and their usage enables the user to choose the most appropriate style for each point-to-point event.

To design and use events, follow these three basic steps:

1. Define an event method. An event method is a method on any class, representing work to be done by that event on a specific simulation object.

2. Plug the user-defined event into the SPEEDES framework.

3. Call a schedule function anywhere within the simulation to execute the event. The schedule functions cannot be called during simulation object construction (i.e. constructors).

SPEEDES provides a set of macros that turn methods on simulation objects into events, plug these events into the SPEEDES framework, and generate functions for scheduling these events. To make scheduling events convenient, the macros automatically build a global function for each event defined, which users can use to invoke their events. But before events can be discussed in detail, simulation time must be understood.

## 6.1 Time, As Represented by SPEEDES

The previous chapters have introduced simulation objects and variables used to maintain simulation object state. Events are used to operate on or change the values of the simulation object state variables, which will be shown later in this chapter. When events are scheduled, the user specifies a time at which the event is to be executed. In SPEEDES, time is represented by the class `SpSimTime`, which contains five parts consisting of a double for physical execution time and four tie breaking fields. The constructor for `SpSimTime` is shown below:

```
SpSimTime(double time      = 0,
          int    priority1 = 0,
          int    priority2 = 0,
          int    counter   = 0,
          int    uniqueId  = 0)
```

| Parameter | Description |
|---|---|
| time | This is the floating point value of time. The units normally used are the number of seconds since the start of the simulation. |
| priority1 | This is a user-modifiable field. This field is used by SPEEDES as a tie-breaking field when the floating point values of two `SpSimTime` objects are identical. Lower numbers for this field indicate higher priority. If two `SpSimTime` objects have the same floating point value, then an object with a lower number for priority1 occurs earlier in time than a object with a higher number for priority1. This field can be set by the SPEEDES framework. Therefore, users should adjust this field by incrementing its value. |
| priority2 | This is a user-modifiable field. This field is used by SPEEDES as a tie-breaking field when the floating point values of two `SpSimTime` objects are identical and when the priority1 values are identical. This field can be set by the SPEEDES framework. Therefore, users should adjust this field by incrementing its value. |
| counter | This is a monotonically increasing value set by SPEEDES in order to guarantee unique ordering of events. |
| uniqueId | This is another field set by SPEEDES in order to guarantee a unique ordering of events. This is set to the global id of the simulation object that scheduled this event. |

Table 6.1: SpSimTime Constructor API

Parameters `counter` and `uniqueId` are set by the SPEEDES framework in order to ensure unique and repeatable simulation time for every event. Hence, changes made to these fields by the user are ignored. The `uniqueId` is always set to the global id of the simulation object scheduling the event. The `counter` parameter is a more difficult parameter to explain. All simulation objects contain an internal data counter for `counter`. This parameter will be greater than or equal to the total number of events scheduled by this simulation object. During event scheduling, SPEEDES assigns the counter as the larger of the simulation object counter or the current event execution time counter (once again, `counter` retrieved from `SpGetTime`) plus 1 for first event scheduled. For the second event scheduled it adds 2 and so on. At the end of the event, the simulation object counter is incremented to the last `counter` used during event scheduling.

For example, if the current simulation object has a counter of 4 and the current event has a counter of 12, the counter of scheduled events are set to 13, 14, . . ., (12 + 1 + number of events scheduled). The counter of the simulation object is then set to 12 + 1 + number of events scheduled.

The useful APIs for `SpSimTime` are shown in Figure 6.1. In addition to the APIs shown below, the operator $<<$ has been overloaded to print both to `ostream` and `RB_ostream`. This prints out all of the `SpSimTime` current attributes.

```
double     GetTime()                          // Get physical time
void       SetTime(double time)               // Set physical time
int        SetPriority1(int priority)         // Set priority 1
int        GetPriority1()                     // Get priority 1
SpSimTime& IncrementPriority1(int value = 1) // Increment priority 1
SpSimTime& DecrementPriority1(int value = 1) // Decrement priority 1
int        SetPriority2(int priority)         // Set priority 2
int        GetPriority2()                     // Get priority 2
SpSimTime& IncrementPriority2(int value = 1) // Increment priority 2
SpSimTime& DecrementPriority2(int value = 1) // Decrement priority 2
```

Figure 6.1: Simulation Time (SpSimTime) API

## 6.2 Simulation Object Events

Simulation object events are "public" events, in that any simulation object in the simulation may directly schedule a simulation object event. Because they are defined at the highest and most accessible level of an object, simulation object events provide the top-level interface between one simulation object and another.

The first step in creating a simulation object event is to add a method to the simulation object. This method can contain up to eight parameters of any non-pointer type. If a class is used as a parameter type, it is highly recommended that the class does not contain pointers. This is because, when sending objects as event arguments, pointer data becomes corrupted. Also, if a copy constructor exists for these classes, then the copy constructor must be public.

Once the method has been created, turn it into an event by using a SPEEDES built-in macro. This macro will automatically generate additional code, turning your simulation object method into an event.

The macro's syntax is as follows:

```
DEFINE_SIMOBJ_EVENT_<numParam>_ARG(eventName,
                                   className,
                                   methodName,
                                   [paramList])
```

| Parameter | Description |
|---|---|
| numParam | The number of parameters used in the method being converted into an event (valid range is 0 to 8). |
| eventName | Any user-defined string representing the name of the event (legal characters for string names include alphanumeric and underscore characters). |
| className | The name of the simulation object class that contains the method that is to be turned into an event. |
| methodName | The name of the method that is to be turned into an event. |
| paramList | Comma-delimited list of the parameter types found in the method. |

Table 6.2: Macro DEFINE_SIMOBJ_EVENT API

After the events have been defined, you need to plug the events into the SPEEDES framework. The plug-in macro creates an event name versus event class database (i.e. the event class we created during the define simulation object event phase), so that the framework can effectively manage events. The plug-in macro syntax is:

```
PLUG_IN_EVENT(eventName)
```

The `eventName` used in the `PLUG_IN_EVENT` macro is the same name as was used in the macro `DEFINE_SIMOBJ_EVENT` shown above.

A convention that works well is to add the define event macro at the end of the class definition and create a function call for the event plug-in. The following shows an example of this convention.

```
#ifndef S_MySimObj_H
#define S_MySimObj_H

#include "SpMainPlugIn.H"
#include "SpSimObj.H"
#include "SpDefineSimObj.H"
#include "SpDefineEvent.H"
#include "MyClass.H"

class S_MySimObj: public SpSimObj {
  public:
    MyMethod(int param1, double param2, MyClass param3);
};
// Define 1 S_MySimObj simulation object
DEFINE_SIMOBJ(S_MySimObj, 1, SCATTER);
// Define event MySimObj_MyMethod
DEFINE_SIMOBJ_EVENT_3_ARG(MySimObj_MyMethod, S_MySimObj, MyMethod,
                          int, double, MyClass);

void PlugInMySimObj() {
  PLUG_IN_EVENT(MySimObj_MyMethod);
  PLUG_IN_SIMOBJ(S_MySimObj);
}

#endif
```

Example 6.1: Generic Simulation Object Definition

The above example shows an event being defined (i.e. `MySimObj_MyMethod`) and a plug-in function that automates the registration process for event `MySimObj_MyMethod` and simulation object `S_MySimObj` for the SPEEDES framework. To use this object and its event, first add the plug-in function call to `main` prior to the execution of `ExecuteSpeedes`. Then, as is described next, you can schedule the event from any simulation object in the simulation. To facilitate the scheduling of events, the define event macro creates a global scheduling function for each user-defined event. The schedule functions created have the following syntax:

```
SpCancelHandle
SCHEDULE_<eventName>(const SpSimTime&   simTime,
                     const SpObjHandle& objHandle,
                                        [paramList],
                     const char*        data = NULL,
                     int                dataBytes = 0)
```

| Parameter | Description |
|---|---|
| eventName | This is the same name used when the event was defined. |
| simTime | This parameter specifies the time at which the event will be executed. The time scheduled can be the present time or a future time, but not a time in the past. |

| Parameter | Description |
|---|---|
| objHandle | This parameter uniquely specifies the simulation object on which the event will be executed. There are convenience functions provided in the SPEEDES framework which can look up object handles for simulation objects in SPEEDES. Please see Chapter 3 for additional information on class `SpObjHandle`. |
| paramList | This is a comma-delimited list of the parameters that are to be passed to the simulation object method (e.g. for the above example event method, `MyMethod`, the list would be: an `int`, a `double`, a `MyClass`). |
| data | This optional parameter allows users to send data to the receiving event for further processing. The data can be binary or character stream data. To send a data structure which contains pointers, make sure to write "wrap" and "unwrap" functions to enable packing and unpacking of a buffer that represents the data. SPEEDES copies this data, so the user does not need to keep the data available after the function call. |
| dataBytes | This parameter represents the size, in bytes, of the buffer sent as the "data" parameter. If you do not use the "data" parameter, then there is no need to use this parameter either. |

Table 6.3: Function SCHEDULE API

We have just discussed the basics for creating, plugging in, and scheduling events. To illustrate these concepts in action, let us build an example program that simulates a car whose position changes with time. The simulation will contain a car simulation object (or many car objects), and will contain a traffic light simulation object that will interact with the car objects by telling them when to start and stop. To simplify the problem, the car simulation objects will only move in a straight line along one axis. The traffic lights will be located every quarter mile and will change states every 30 seconds. The code shown in Examples 6.2 through 6.5 shows the definitions and implementations files for the stop light and car simulation objects.

```
// S_StopLight.H
#ifndef S_StopLight_H
#define S_StopLight_H

#include "SpSimObj.H"
#include "SpDefineSimObj.H"
#include "SpDefineEvent.H"

class S_StopLight : public SpSimObj {
  public:
    S_StopLight() {}
    virtual ~S_StopLight() {}
    virtual void Init();
    void         Red();
    void         Green();
  protected:
  private:
};

DEFINE_SIMOBJ(S_StopLight, 1, SCATTER);
DEFINE_SIMOBJ_EVENT_0_ARG(StopLight_TurnsRed, S_StopLight, Red);
DEFINE_SIMOBJ_EVENT_0_ARG(StopLight_TurnsGreen, S_StopLight, Green);
#endif
```

Example 6.2: Point-to-Point Event StopLight Object Definition File

```
// S_StopLight.C
#include "SpGlobalFunctions.H"
```

```
3  #include "SpMainPlugIn.H"
4  #include "RB_ostream.H"

5  #include "S_StopLight.H"
6  #include "S_Car.H"

7  void PlugInStopLight() {
8    PLUG_IN_SIMOBJ(S_StopLight);
9    PLUG_IN_EVENT(StopLight_TurnsRed);
10   PLUG_IN_EVENT(StopLight_TurnsGreen);
11 }

12 void S_StopLight::Init() {
13   SCHEDULE_StopLight_TurnsRed(30.0, SpGetObjHandle());
14 }

15 void S_StopLight::Red() {
16   int i;
17   for (i = 0; i < 4; ++i) {
18     SpObjHandle objHandle = SpGetObjHandle("S_Car_MGR", i);
19     SCHEDULE_Car_Stop(SpGetTime(), objHandle, SpGetTime(),
20                       (char *) "Red Light On",
21                       strlen("Red Light On") + 1);
22   }
23   SCHEDULE_StopLight_TurnsGreen(SpGetTime() + 30.0, SpGetObjHandle());
24 }

25 void S_StopLight::Green() {
26   int i;
27   for (i = 0; i < 4; ++i) {
28     SpObjHandle objHandle = SpGetObjHandle("S_Car_MGR", i);
29     SCHEDULE_Car_Go(SpGetTime(), objHandle,
30                     (char *) "Green Light On",
31                     strlen("Green Light On") + 1);
32   }
33   SCHEDULE_StopLight_TurnsRed(SpGetTime() + 30.0, SpGetObjHandle());
34 }
```

Example 6.3: Point-to-Point Event StopLight Object Implementation File

Example 6.2 defines two methods called `Red` and `Green`, which are turned into events by using the define event macros shown on lines 18 and 19. The define event macro will create global functions called `SCHEDULE_StopLight_TurnsRed` and `SCHEDULE_StopLight_TurnsGreen`. These names are created by prepending `SCHEDULE_` to the name provided in the first parameter of the define event macro. Notice that the define simulation object macro `DEFINE_SIMOBJ` (line 17) defines one stop light simulation object.

Example 6.3 shows the implementation code for the stop light. Function `PlugInStopLight` is provided to plug-in the events and simulation object for the stop light. This function will be called by `main`. All simulation object initialization should occur in the virtual method `Init`. In general, no initialization should be done in the simulation object constructor (see section 3.1 for more detail). This method schedules the event `StopLight_TurnsRed` by calling `SCHEDULE_StopLight_TurnsRed` at time 30.0 seconds for the current simulation object. Method `Red` is used to schedule event `Car_Stop` on each instance of a car simulation object (lines 17-22) and schedule event `StopLight_TurnsGreen` on itself at 30.0 seconds in the future (line 23). Event `Car_Stop` is scheduled for immediate processing, which most likely will cause a rollback on each car object. Notice that this event is sending the current simulation time via an input argument. This shows how to use the parameter fields when an event requires such a field. The receiving car event then prints this data out. Event `StopLight_TurnsRed` is using the optional data field to send the data string `"Red Light On"` to the receiving event. Method `Green` is similar to method `Red` except that it schedules event `Car_Go`.

Examples 6.4 and 6.5 show the code for the car simulation object definition and implementation files.

```
// S_Car.H
#ifndef S_Car_H
#define S_Car_H

#include "SpSimObj.H"
#include "SpDefineSimObj.H"
#include "SpDefineEvent.H"

class S_Car : public SpSimObj {
  public:
    S_Car() {}
    virtual ~S_Car() {}
    virtual void Init();
    void         Stop(double stopTime);
    void         Go();
    void         StopCar();

  protected:
  private:
    RB_double    XPos;               // Car Position
    double       Velocity;           // Car Velocity
    RB_double    LastTimeStopped;    // Time at last Car Stoppage
    RB_int       StopState;          // 0 = Stopped; 1 = Moving
    RB_int       TimesStopped;       // Number of times Car
                                     //  Stopped
};

DEFINE_SIMOBJ(S_Car, 4, SCATTER);
DEFINE_SIMOBJ_EVENT_1_ARG(Car_Stop, S_Car, Stop, double);
DEFINE_SIMOBJ_EVENT_0_ARG(Car_Go, S_Car, Go);
DEFINE_SIMOBJ_EVENT_0_ARG(Car_StopCar, S_Car, StopCar);
#endif
```

Example 6.4: Point-to-Point Event Car Object Definition File

```
// S_Car.C
#include "SpGlobalFunctions.H"
#include "SpMainPlugIn.H"
#include "RB_ostream.H"

#include "S_Car.H"

static int QuarterMile = 5280 / 4; // In feet

void PlugInCars() {
  PLUG_IN_SIMOBJ(S_Car);
  PLUG_IN_EVENT(Car_StopCar);
  PLUG_IN_EVENT(Car_Go);
  PLUG_IN_EVENT(Car_Stop);
}

void S_Car::Init() {
  XPos            = 0;
  Velocity        = (SpGetSimObjGlobalId() * 10.0 + 11.0) / 3600 * 5280;
  StopState       = 0;
  LastTimeStopped = 0.0;
  TimesStopped    = 0;

  SCHEDULE_Car_Go(0.0, SpGetObjHandle());
}

void S_Car::Stop(double stopTime) {
  double currentXPos;
  int    nextStopLightPos;
  double stopLightArrivalTime;

  currentXPos =(XPos + Velocity * (SpGetTime() - LastTimeStopped));
  nextStopLightPos =
       (int) ((((int) currentXPos) / QuarterMile + 1) * QuarterMile);
  stopLightArrivalTime = (nextStopLightPos - currentXPos) / Velocity;
  RB_cout << "Received stop signal for Car #" << SpGetSimObjGlobalId()
          << ", Received argument= " << stopTime
          << ", Sim" << SpGetTime()
          << " Received Data= " << SpGetMsgData()
          << " stopLightArrivalTime= " << stopLightArrivalTime
          << endl;
  SCHEDULE_Car_StopCar(SpGetTime() + stopLightArrivalTime,
                       SpGetObjHandle());
}

void S_Car::Go() {
  if (StopState == 0) {
    LastTimeStopped = SpGetTime();
    StopState       = 1;
  }
}

void S_Car::StopCar() {
  XPos = XPos + (Velocity * (SpGetTime() - LastTimeStopped));
  StopState = 0;
  ++TimesStopped;

  RB_cout << "Car #" << SpGetSimObjGlobalId()
          << " is stopping at " << SpGetTime()
          << " at X coordinate of " << XPos
          << ". Car has stopped " << TimesStopped << " times."
          << " LastTimeStopped= " << LastTimeStopped
          << endl;
}
```

Example 6.5: Point-to-Point Event Car Object Implementation File

Example 6.4 defines three methods called `Stop`, `Go`, and `StopCar`, which are turned into events by using the define event macros, as shown on lines 25 through 27. Notice that method `Stop` is defined with one parameter of type double. This requires that the one parameter define simulation event macro be used with its type specified as double (line 25). The `DEFINE_SIMOBJ` macro on line 24 specifies that 4 cars are to be created in this simulation.

Example 6.5 shows the implementation code for the car simulation objects. Function `PlugInCars` is used to plug-in the Car simulation objects and their events. The function should be called by `main`. Method `Init` is used to initialize the car attributes, which includes a different velocity for each car instance. SPEEDES assigns global ids to objects in the order that they are executed at run time (i.e. the order in which objects are plugged in). Consequently, if the car objects are plugged in first, then the first car object's global id will be 0; the second car object's global id will be 1; and so on. In the end, the four car simulation objects will be assigned global id's of 0 through 3. Finally, because we calculated each car's velocity based on its global id, the cars in this example will have velocities of 11, 21, 31, and 41 miles per hour. Event `Car_Go` starts the simulation a time 0.

When event `Car_Stop` is executed (i.e. method `Stop`), then the car needs to stop at the next red light. The stop lights are located at fixed locations every quarter mile. If the car is not yet at a stop light, then there is no reason for it to stop. Therefore, the car calculates when it will arrive at the next stop light and schedules an event for itself to stop when it arrives at the light (lines 25 through 36). Event `Car_Go` (method `Go`) turns the traffic signals back to green. Event `Car_StopCar` (method `StopCar`) stops the car and updates the car's current position.

The final step in completing the car simulation is to create `main`. The code in Example 6.6 shows what `main` would look like for this simulation.

```
// Main.C
#include "SpMainPlugIn.H"

void PlugInCars();
void PlugInStopLight();

int main(int argc, char** argv) {
  PlugInCars();
  PlugInStopLight();

  ExecuteSpeedes(argc, argv);
}
```

Example 6.6: Car and Stop Light main

## 6.3 Event Cancellation

During the normal execution of events, any event rolled back will automatically cancel any event that it scheduled, as defined within the SPEEDES framework algorithms. The capability to cancel events is also available to the user. That is, when events are scheduled in the future, the user may later decide to change or cancel the events. This capability is useful when an unanticipated event affects the outcome of an event that has been previously scheduled. For example, a missile may schedule an event at a specific impact location, but the missile may be destroyed prior to impact. The destroyed missile's already scheduled impact event needs to be canceled. To enable such event cancellation, the SPEEDES framework contains a global function that allows previously scheduled events to be canceled. The cancel event API is:

```
void SCHEDULE_CANCEL_EVENT(SpCancelHandle& cancelHandle)
```

When events are scheduled, a cancel handle is returned to the user, which then can be used to cancel the scheduled event, if necessary.

Now let us take a closer look at our previous example. Car 0 is traveling at 11 miles per hour. At 30.0 seconds the light changes from green to red and event `Car_Stop` is scheduled on Car 0. The car has traveled 484 feet, which makes it still 836 feet away from the stop light. This event then calculates that, at 11 miles per hour, Car 0 will not reach the next light until 51.8 seconds from now, consequently it schedules an event to stop the car at 81.8 seconds (30.0 + 51.8). However, the light turns green at 60.0 seconds, so when the car arrives at the traffic light at 81.8 seconds, the light is green. Unfortunately, the car will now stop at a green light (must be a California driver). To solve this, we can save the cancel handle from the original scheduling of the `Car_Stop` event. When the light turns green, we can use the cancel handle to cancel the event.

To fix the previous example, we need to modify the `S_Car` simulation object by saving the cancel handle for event `Car_StopCar` when event `Car_Stop` schedules it. We also save the time at which the event is scheduled, so that we do not try and remove an event in the past. Add the following lines of code to the private section of the `S_Car` definition file:

```
RB_SpCancelHandle CancelHandle;
RB_double         EventScheduledTime;
```

Four changes are required to the `S_Car` implementation file. Replace lines 35 and 36 in Example 6.5 with:

```
EventScheduledTime = 0.0;
CancelHandle = SCHEDULE_Car_Go((double) EventScheduledTime,
                               SpGetObjHandle());
```

Replace line 42 with:

```
EventScheduledTime = SpGetTime() + stopLightArrivalTime;
CancelHandle = SCHEDULE_Car_StopCar((double) EventScheduledTime,
                                    SpGetObjHandle());
```

Next, replace all of the code in method `S_Car::Go` with:

```
if (EventScheduledTime > SpGetTime()) {
  SCHEDULE_CANCEL_EVENT(CancelHandle);
  EventScheduledTime = -1.0;
}
else {
  if (StopState == 0) {
    LastTimeStopped = SpGetTime();
    StopState       = 1;
  }
}
```

Lastly, add the following include file:

```
#include "SpSchedule.H"
```

Event `Car_StopCar` is still scheduled when the light changes from green to red, but the cancel handle is saved so that the event can be later canceled if necessary. Now, when the light changes from red to green, the time that the light turned green is compared to the time for when the car will stop. If the light turned green prior to the car reaching the light (i.e. `SpGetTime() < EventScheduledTime`), then the event for stopping the car is canceled (i.e. `Car_StopCar`).

## 6.4 Local Events

Local events are “private” events. A simulation object may have arbitrarily-nested sub-objects that represent its encapsulated implementation. Many of these sub-objects may be dynamically created and destroyed. This means only the simulation object, itself, has access to them. Thus, local point-to-point events begin and end at the same “point” (i.e. the same simulation object). The purpose of local events is to allow a simulation object to manage and animate its sub-objects with self-scheduled events.

Local events can be created on any object that is a part of your simulation object state. An important consequence of this is that the object that contains the local events does not have to inherit from the class `SpSimObj`. To create local events, the first step is to create a class. Methods on the class can be turned into events by using the macro designed for local events. The API for this macro is:

```
DEFINE_LOCAL_EVENT_<numParam>_ARG(eventName,
                                  className,
                                  methodName,
                                  [paramList])
```

| Parameter | Description |
|---|---|
| numParam | The number of parameters used in the method being converted into a local event (valid range is 0 to 8). |
| eventName | Any user-defined string representing the name of the event (legal characters for string names include alphanumeric and underscore characters). |
| className | The name of the class that contains the method that is to be turned into an event. |
| methodName | The name of the method that is to be turned into an event. |
| paramList | Comma-delimited list of the parameter types found in the method. |

Table 6.4: Macro DEFINE_LOCAL_EVENT API

These events are plugged into the SPEEDES framework via the same plug-in macro described previously (see page 89). To enable scheduling local events, the define local event macro creates a global schedule function for each local event defined. Scheduling these events is very similar to scheduling the simulation object events, as previously described. The only difference is: for local events, pass in a reference to the local object instead of passing in the simulation object handle (since SPEEDES already knows that the scheduled object handle must be the current object handle). An example is shown later. The local event schedule function has the following API:

```
SpCancelHandle
SCHEDULE_<eventName>(const SpSimTime&  simTime,
                     const className&  object,
                                       [paramList],
                     const char*       data = NULL,
                     int               dataBytes = 0)
```

| Parameter | Description |
|---|---|
| eventName | This is the same name used when the event was defined. |
| simTime | This parameter specifies the time at which the event will be executed. The time scheduled can be the present time or a future time, but not a time in the past. |
| object | This is the object instance on which the local event will act. |
| paramList | This is a comma-delimited list of the parameters that are to be passed to the event method. |
| data | This optional parameter allows users to send data to the event for further processing. The data can be binary or character stream data. To send something with pointers, make sure to write "wrap" and "unwrap" functions to enable packing and unpacking of a buffer that represents the data. |
| dataBytes | This parameter represented the size, in bytes, of the buffer sent as the "data" parameter. If you do not use the "data" parameter, then there is no need to use this parameter either. |

Table 6.5: Function SCHEDULE API for Local Events

The next step in using local events is to create an instance of the local object on your simulation object. One word of caution: local events cannot schedule local events on other simulation objects. Simulation objects are identified by their object handles. Those object handles uniquely identify the object instance (including on which node or processor the object is located). However, local events reside on objects that have no such identification, since they are not necessarily simulation objects. So, there is no way to identify the objects on which local events act, unless those objects are contained in the current simulation object. This is why it is only possible to schedule local events on the current (i.e. local) simulation object.

We have just discussed the basics of local events. Now, let us enhance our prior stop light example to show local events in action. Let us give our car a radio which will be a local object with local events. The code shown in Examples 6.7 and 6.8 shows the definition and implementation files for the local Radio object.

```
// Radio.H
#ifndef RADIO_H
#define RADIO_H

#include "SpDefineEvent.H"
#include "RB_int.H"
#include "RB_double.H"
#include "RB_SpCancelHandle.H"

class Radio {
  public:
    Radio();
    ~Radio() {}

    void Off();
    void On();
    void Scan(int scheduler);
    void SendFrequency();
    int  GetState() {return (State);}

  protected:
  private:
    RB_int            State;         // 0 = Off, 1 = On, 2 = Scan
    RB_double         Frequency;     // Radio frequency
    RB_int            TimesScanned;  // Number of frequencies scanned
    RB_SpCancelHandle CancelHandle;  // Radio_Scan Cancel Handle
    RB_double         EventScheduledTime;
                                     // Time Scan event was scheduled
};

DEFINE_LOCAL_EVENT_0_ARG(Radio_Off,  Radio, Off);
DEFINE_LOCAL_EVENT_0_ARG(Radio_On,   Radio, On);
DEFINE_LOCAL_EVENT_1_ARG(Radio_Scan, Radio, Scan, int);
DEFINE_LOCAL_EVENT_0_ARG(Radio_SendFrequency, Radio, SendFrequency);
#endif
```

Example 6.7: Local Event Radio Object Definition

```
// Radio.C
#include "SpGlobalFunctions.H"
#include "SpSchedule.H"
#include "SpMainPlugIn.H"
#include "RB_ostream.H"

#include "Radio.H"
#include "S_Car.H"

void PlugInRadio() {
  PLUG_IN_EVENT(Radio_Off);
  PLUG_IN_EVENT(Radio_On);
  PLUG_IN_EVENT(Radio_Scan);
  PLUG_IN_EVENT(Radio_SendFrequency);
}

Radio::Radio() : State(0), Frequency(100.0), TimesScanned(0) {}

void Radio::Off() {
  RB_cout << "Turning radio in car #" << SpGetSimObjGlobalId()
          << " off at " << SpGetTime() << endl;
  State = 0;
}

void Radio::On() {
  RB_cout << "Turning radio in car #" << SpGetSimObjGlobalId()
          << " on at " << SpGetTime() << endl;
  State        = 1;
  TimesScanned = 0;
  SCHEDULE_Radio_SendFrequency(SpGetTime(), *this);
}

void Radio::Scan(int scheduler) {
  RB_cout << "Radio in car #" << SpGetSimObjGlobalId()
          << " is scanning for new station at " << SpGetTime() << endl;
  if (State == 2) {       // Are we currently scanning?
    if (scheduler == 0) { // Did the Car push the Scan Button
      /*
       * Car pushed the scan button and we were already scanning.
       * Therefore, stop the Radio at the current station and cancel
       * the next self scheduled scan event.
       */
37       SCHEDULE_Radio_On(SpGetTime(), *this);
38       if (EventScheduledTime > SpGetTime()) {
39         SCHEDULE_CANCEL_EVENT(CancelHandle);
40         EventScheduledTime = -1.0;
41       }
42     }
43     else {
44       /*
45        * The radio scheduled this event.  If we have changed radio
46        * frequencies less than 10 times then change the frequency
47        * and reschedule the scan event.  Otherwise, stop the
48        * radio scanning at the current radio frequency.
49        */
50       if (TimesScanned < 10) {
51         Frequency = Frequency + 101.0;
52         ++TimesScanned;
53         EventScheduledTime = SpGetTime() + 50.0;
54         CancelHandle =
55           SCHEDULE_Radio_Scan((double) EventScheduledTime, *this, 1);
56         SCHEDULE_Radio_SendFrequency(SpGetTime(), *this);
57       }
58       else {
59         SCHEDULE_Radio_On(SpGetTime(), *this);
60       }
61     }
62   }
63   else {
64     /*
65      * Since the current radio state is 1, by default the car just
66      * pushed the radio scan button.  Therefore, the radio
67      * should start scanning frequencies for a new radio station.
68      */
69     State = 2;
70     ++TimesScanned;
71     Frequency = Frequency + 101.0;
72     EventScheduledTime = SpGetTime() + 50.0;
73     CancelHandle =
74       SCHEDULE_Radio_Scan((double) EventScheduledTime, *this, 1);
75     SCHEDULE_Radio_SendFrequency(SpGetTime(), *this);
76   }
77 }

78 void Radio::SendFrequency() {
79   SCHEDULE_Car_RadioStation(SpGetTime(), SpGetObjHandle(), Frequency);
80 }
```

Example 6.8: Local Event Radio Object Implementation

Our radio will be able to take on the states of "on", "off", and "scan". The car will cycle through the different radio states via scheduling events on the radio object (i.e. local events). Each time the radio station changes, the Radio object schedules an event on its creator (Example 6.8, line 79), which notifies the car of the current radio frequency. Notice that we use global function `SpGetObjHandle` to get the object handle for the simulation object that originally started the sequence of events on this local object.

The scan event scans through ten frequencies, after which it stops scanning. However, the car could push the scan button while the radio is scanning, indicating that the Radio is to stop scanning. Therefore, the cancel handle for the scan event is saved so that any scan event already scheduled can be canceled.

The code shown in Examples 6.9 and 6.10 shows the new definition and implementation for the car simulation object.

```
// S_Car.H
#ifndef S_Car_H
#define S_Car_H

#include "SpSimObj.H"
#include "SpDefineSimObj.H"
#include "SpDefineEvent.H"
#include "RB_SpCancelHandle.H"
#include "Radio.H"

class S_Car : public SpSimObj {
  public:
    S_Car() {}
    virtual ~S_Car() {}
    virtual void Init();
    void         Stop(double stopTime);
    void         Go();
    void         StopCar();
    void         RadioController();
    void         RadioStation(double frequency);

  protected:
  private:
    RB_double         XPos;              // Car Position
    double            Velocity;          // Car Velocity
    RB_double         LastTimeStopped;   // Time at last Car Stoppage
    RB_int            StopState;         // 0 = Stopped; 1 = Moving
    RB_int            TimesStopped;      // Number of times Car
                                         //  Stopped
    RB_SpCancelHandle CancelHandle;      // StopCar Cancel Handle
    RB_double         EventScheduledTime; // StopCar Event Scheduled
    Radio             MyRadio;           // Car Radio
};

DEFINE_SIMOBJ(S_Car, 4, SCATTER);
DEFINE_SIMOBJ_EVENT_1_ARG(Car_Stop, S_Car, Stop, double);
DEFINE_SIMOBJ_EVENT_0_ARG(Car_Go, S_Car, Go);
DEFINE_SIMOBJ_EVENT_0_ARG(Car_StopCar, S_Car, StopCar);
DEFINE_SIMOBJ_EVENT_0_ARG(Car_RadioController, S_Car, RadioController);
DEFINE_SIMOBJ_EVENT_1_ARG(Car_RadioStation, S_Car, RadioStation, double);
#endif
```

Example 6.9: Local Event Car Object Definition File

```
// S_Car.C
#include "SpGlobalFunctions.H"
#include "SpMainPlugIn.H"
#include "RB_ostream.H"
#include "SpSchedule.H"

#include "S_Car.H"

static int QuarterMile = 5280 / 4; // In feet

void PlugInCars() {
  PLUG_IN_SIMOBJ(S_Car);
  PLUG_IN_EVENT(Car_StopCar);
  PLUG_IN_EVENT(Car_Go);
  PLUG_IN_EVENT(Car_Stop);
  PLUG_IN_EVENT(Car_RadioController);
  PLUG_IN_EVENT(Car_RadioStation);
}

void S_Car::Init() {
  XPos            = 0;
  Velocity        = (SpGetSimObjGlobalId() * 10.0 + 11.0) / 3600 * 5280;
  StopState       = 0;
  LastTimeStopped = 0.0;
  TimesStopped    = 0;

  EventScheduledTime = 0.0;
  CancelHandle = SCHEDULE_Car_Go((double) EventScheduledTime,
                                 SpGetObjHandle());
  SCHEDULE_Car_RadioController(SpGetSimObjGlobalId() * 10.0,
                               SpGetObjHandle());
}

void S_Car::Stop(double stopTime) {
  double currentXPos;
  int    nextStopLightPos;
  double stopLightArrivalTime;

  currentXPos = (XPos + Velocity * (SpGetTime() - LastTimeStopped));
  nextStopLightPos  =
    (int) ((((int) currentXPos) / QuarterMile + 1) * QuarterMile);
  stopLightArrivalTime = (nextStopLightPos - currentXPos) / Velocity;
  RB_cout << "Received stop signal for Car #" << SpGetSimObjGlobalId()
          << ", Received argument= " << stopTime
          << ", Sim" << SpGetTime()
          << " Received Data= " << SpGetMsgData()
          << " stopLightArrivalTime= " << stopLightArrivalTime
          << endl;
  EventScheduledTime = SpGetTime() + stopLightArrivalTime;
  CancelHandle = SCHEDULE_Car_StopCar((double) EventScheduledTime,
                                              SpGetObjHandle());
}

void S_Car::Go() {
  if (EventScheduledTime > SpGetTime()) {
    SCHEDULE_CANCEL_EVENT(CancelHandle);
    EventScheduledTime = -1;
  }
  else {
    if (StopState == 0) {
      LastTimeStopped = SpGetTime();
      StopState       = 1;
    }
  }
}

void S_Car::StopCar() {
  XPos = XPos + (Velocity * (SpGetTime() - LastTimeStopped));
  StopState = 0;
  ++TimesStopped;

  RB_cout << "Car #" << SpGetSimObjGlobalId()
          << " is stopping at " << SpGetTime()
          << " at X coordinate of " << XPos
          << ". Car has stopped " << TimesStopped << " times."
          << " LastTimeStopped= " << LastTimeStopped
          << endl;
}

void S_Car::RadioController() {
  int currentRadioState = MyRadio.GetState();
  int nextRadioState;

  if (currentRadioState != 2) {
    nextRadioState = (currentRadioState + 1) % 3;
  }
  else {
    if (GetRandom()->GenerateDouble() > 0.75) {
      nextRadioState = currentRadioState;
    }
    else {
      if (GetRandom()->GenerateDouble() > 0.5) {
        nextRadioState = 0;
      }
      else {
        nextRadioState = 1;
      }
    }
  }

  switch (nextRadioState) {
    case 0: SCHEDULE_Radio_Off  ( SpGetTime() + 10.0, MyRadio);
            break;
    case 1: SCHEDULE_Radio_On   ( SpGetTime() + 15.0, MyRadio);
            break;
    case 2: SCHEDULE_Radio_Scan ( SpGetTime() + 20.0, MyRadio, 0);
            break;
    default: break;
  }
  SCHEDULE_Car_RadioController(SpGetTime() + 400.0, SpGetObjHandle());
}
```

```
 99 void S_Car::RadioStation(double frequency) {
100   RB_cout << "Radio Station for car #" << SpGetSimObjGlobalId()
101           << " is at frequency " << frequency << endl;
102 }
```

Example 6.10: Local Event Car Object Implementation File

In Example 6.9, the definition file added two methods called `RadioController` and `RadioStation` on lines 17 and 18, respectively. The radio local object was added to the car simulation object on line 29. Finally, macro `DEFINE_SIMOBJ_EVENT` is used to turn `RadioController` and `RadioStation` into events, as shown on lines 35 and 36.

The car simulation object contains two additional methods. In Example 6.10, lines 69 through 102 show the implementation for methods that cycle though the radio's states, as well as displaying the radio's frequency.

Finally, Example 6.11 shows the new file for `main`.

```
// Main.C
#include "SpMainPlugIn.H"

void PlugInCars();
void PlugInStopLight();
void PlugInRadio();

int main(int argc, char** argv) {
  PlugInCars();
  PlugInStopLight();
  PlugInRadio();

  ExecuteSpeedes(argc, argv);
}
```

Example 6.11: Local Event main

## 6.5 Autonomous Events

The final SPEEDES API point-to-point event type is the autonomous event. Where, as previously discussed, point-to-point events all are based on methods residing directly or indirectly in the simulation object, autonomous events have been separated from the simulation object upon which they act.

Autonomous events inherit from class `SpEvent`. This gives users access to several virtual methods on the event, which provides greater control over the event during its different event phases (see Section 15.2). The most likely method to be used would be `lazy`, which allows users to rollforward a rolled back event to prevent reexecution of the event. The allows for events that are CPU intensive to not be reexecuted if not necessary. The downside to this approach is the separation of the event from the simulation object it works on, which can add to event code complexity and makes code more difficult to maintain. Chapter 15 describes autonomous events in detail.

When defining autonomous events, another SPEEDES macro is used which unifies the API with the rest of SPEEDES event definitions. The autonomous event macro API is:

Let us expand the car example to show how to make an autonomous event by converting event `Car_Stop` (i.e. method `S_Car::Stop`) to an autonomous event. The code shown in Example 6.12 and 6.13 shows the previous car event `Car_Stop` turned into the new autonomous event by the same name.

```
DEFINE_AUTONOMOUS_EVENT_<numParam>_ARG(eventName,
                                       className,
                                       methodName,
                                       [ParamList])
```

| Parameter | Description |
|---|---|
| numParam | The number of parameters used in the method being converted into an event (valid range is 0 to 8). |
| eventName | Any user-defined string representing the name of the event (legal characters for string names include alphanumeric and underscore characters). |
| className | The name of the event class that contains the method that is to be turned into an event. |
| methodName | The name of the method that is to be turned into an event. |
| paramList | A comma-delimited list of the parameter types found in the method. |

Table 6.6: Macro DEFINE_AUTONOMOUS_EVENT API

```
// E_Car_Stop.H
#ifndef E_Car_Stop_H
#define E_Car_Stop_H

#include "SpDefineEvent.H"

class E_Car_Stop : public SpEvent {
  public:
    E_Car_Stop();
    virtual ˜E_Car_Stop() {};

    virtual int lazy();
    void        Stop(double stopTime);

  protected:
  private:
    double XPos;
};

DEFINE_AUTONOMOUS_EVENT_1_ARG(Car_Stop, E_Car_Stop, Stop, double);
#endif
```

Example 6.12: Autonomous Event Stop Event Definition File

```
// E_Car_Stop.C
#include "SpMainPlugIn.H"
#include "RB_ostream.H"

#include "E_Car_Stop.H"
#include "S_Car.H"

static int QuarterMile = 5280 / 4; // In feet

void PlugInECarStop() {
  PLUG_IN_EVENT(Car_Stop);
}

E_Car_Stop::E_Car_Stop() {
  enable_lazy();
}

int E_Car_Stop::lazy() {
  int    returnValue = 0;
  S_Car* carObj      = (S_Car *) SpGetSimObj();
  if (XPos == carObj->GetXPos()) {
    /*
     * Since Xpos did not change from last time there is no need
     * to reprocess the Car_Stop event.  Therefore return non-zero.
     */
    returnValue = 1;
  }
  return(returnValue);
}

void E_Car_Stop::Stop(double stopTime) {
  int            nextStopLightPos;
  double         stopLightArrivalTime;
  S_Car*         carObj = (S_Car *) SpGetSimObj();
  double         eventScheduledTime;
  SpCancelHandle cancelHandle;

  XPos = carObj->GetXPos()      +
         (carObj->GetVelocity() *
         (SpGetTime() - carObj->GetLastTimeStopped()));
  nextStopLightPos     =
       (int) ((((int) XPos) / QuarterMile + 1) * QuarterMile);
  stopLightArrivalTime = (nextStopLightPos - XPos) /
                          carObj->GetVelocity();
  RB_cout << "Received stop signal for Car #" << SpGetSimObjGlobalId()
          << ", Received argument= " << stopTime
          << ", Sim" << SpGetTime()
          << " Received Data= " << SpGetMsgData()
          << " stopLightArrivalTime= " << stopLightArrivalTime
          << endl;
  eventScheduledTime = SpGetTime() + stopLightArrivalTime;
  cancelHandle = SCHEDULE_Car_StopCar((double) eventScheduledTime,
                                               SpGetObjHandle());
  carObj->SetEventScheduledTime(eventScheduledTime);
  carObj->SetCancelHandle(cancelHandle);
}
```

Example 6.13: Autonomous Event Stop Event Implementation File

The format for these files is very similar to both the simulation object event and local event definition and implementation files. Example 6.12 shows the definition for our new autonomous event. By SPEEDES convention, autonomous events are prefixed with an `E_` This example implements the virtual method `lazy` and user method `Stop` (Example 6.12, lines 9 and 10). The method lazy, is one of several optional virtual methods which, if implemented, conditionally prevents work from being redone during rollbacks. In other words, during normal event processing (i.e. when lazy is not enabled), if a simulation object has been rolled back, then events on the object will always be reprocessed. However, reprocessing these events may not always be necessary. When lazy is enabled, lazy methods check whether reprocessing rolled back events could possibly produce an outcome different than when the rolled back event was initially processed. If not, then the event can be rolled forward, saving execution time by not re-executing the rolled back event (e.g. method `E_Car_Stop::Stop` in this case). Also, events that the rolled back event scheduled are not canceled. This allows additional simulation run-time

```
parameters {
  logical lazy T
}
```

Figure 6.2: Autonomous Event speedes.par Lazy Cancellation

performance increases.

Line 15 shows how to use the autonomous event macro to turn the method `Stop` into event `Car_Stop`. The code in Example 6.13 shows the implementation for the autonomous event. Function `PlugInECarStop` is used to plug the event into the SPEEDES framework. Since our example uses lazy cancellation, we must enable lazy (line 11) and implement the method `lazy` (line 13 through 24), which defines how to determine if lazy passed or failed. Method `E_Car_Stop::Stop` is the same code as method `S_Car::Stop`. Notice, however, that this new method does not have access to the private data in class `S_Car`. This requires that accessors and mutators for class `S_Car` be written. When this event is executed, access to the `S_Car` object is gained by using global function `SpGetSimObj` and type casting its output to a `S_Car`.

To use the new autonomous event, we need to make some minor changes to files `S_Car.H` and `S_Car.C`. First, delete the `Stop` method from files `S_Car.H` and `S_Car.C`. In `S_Car.C`, delete the plug-in for event `Car_Stop`. In `S_Car.H`, delete the define simulation object event macro for event `Car_Stop`. Finally, add accessors and mutators for the data elements that our autonomous event needs. In our example, add the following to `S_Car.H`:

```
  int          GetXPos() {return(XPos);}
  double       GetVelocity() {return(Velocity);}
  double       GetLastTimeStopped() {return(LastTimeStopped);}
  void         SetCancelHandle(SpCancelHandle& ch) {
    CancelHandle = ch;
  }
  void         SetEventScheduledTime(double& est) {
    EventScheduledTime = est;
  }
```

Next, modify code S_StopLight.C slightly by adding:

```
  #include "E_Car_Stop.H"
```

Finally, we must plug-in our new autonomous event. Do this by adding the following to `Main.C`.

```
  void PlugInECarStop();

  PlugInECarStop();
```

When running the example car simulation, lazy must be enabled in the `speedes.par` file. The `speedes.par` file shown in Figure 6.2 shows what this would look like.

## 6.6 Choosing an Interface Style

There are two interface styles for point-to-point events: merged and separate. The examples in this User's Guide use the merged style. However, if compile time is a concern, the separate interface style may be a better choice, as it reduces unnecessary dependencies between files. This section explores the difference between these two styles to enable users to make informed decisions in choosing between them.

The separate interface style requires defining and including a separate interface header file. This interface file is then also included by files that contain code that schedules the event. In this way, a compilation-independent interface serves to assure type consistency between the event method and the schedulers of that event.

The merged interface style does not require a separate interface file. Instead, the header file with the event method definition serves as the interface for schedulers of the event. Rather than residing in an independent file, the interface is "merged" with the event method definition macro, requiring the event method header file to serve both as a header for the class in which it is contained, and also as an interface for schedulers.

The separate interface style minimizes compilation dependencies while still maintaining complete event parameter type checking. So, when compilation efficiency is a concern, the separate interface style is recommended for simulation object events and sometimes for autonomous events as well. For local events, it only has value when files with schedulers are able to compile the pointer to the event method's object by means of a forward class reference. Otherwise, they require including the very event method header file that the separate interface style is designed not to require including, thus eliminating the benefit of a separate interface file.

While the merged interface style requires less code and has equivalent type checking, it creates unnecessary include file dependencies, thus lengthening the average time required for compilation during software development. For instance, when using the merged interface style for simulation object events, whenever a software developer modifies a simulation object header file containing one or more event methods, all files that schedule events corresponding to any of those methods include that simulation object header file and, thus, must get recompiled. Conversely, none of the event scheduling files get recompiled using the separate interface style.

In summary, the separate interface style minimizes compile times, since it minimizes the dependency between the event method and the event scheduler to the one and only actual dependency: the event parameters. Since this is the one and only condition where both the method and the schedulers must change their code, this is the one and only condition where a compilation dependency is needed. All other dependencies are spurious, and only degrade compilation efficiency.

However, the merged interface style is more convenient, and requires less files. The merged interface may prove expedient when porting legacy systems. It may also simplify development when the event method header files have few dependencies, are changed infrequently, or both. Finally, it may make sense in some situations to use the more expedient merged interface style during initial prototyping or legacy system porting, and then as the code matures, eventually port to the separate interface style.

## 6.7 The Separate Interface Style

The separate interface style is recommended when compilation efficiency is important. The independence of the interface file is the key to minimizing compile times, since schedulers will not be recompiled when the event method header file changes, but only when the interface file changes.

To create an event using the separate interface style, first create the interface as an independent file. The file needs to include `SpDefineEvent.H` and then, insert the following macro:

```
DEFINE_EVENT_INTERFACE_<numParam>_ARG(eventName,
                                       [paramList])
```

| Parameter | Description |
|---|---|
| numParam | The number of parameters used in the method being converted into an event (valid range is 0 to 8). |
| eventName | Any user-defined string representing the name of the event (legal characters for string names include alphanumeric and underscore characters). |
| paramList | A comma-delimited list of the parameter types found in the method. |

Table 6.7: Macro DEFINE_EVENT_INTERFACE API

The corresponding interface file for the event in Example 6.1 is shown in Example 6.14

```
#ifndef I_MY_EVENT
#define I_MY_EVENT

#include "SpDefineEvent.H"
#include "MyClass.H"
DEFINE_EVENT_INTERFACE_3_ARG(MyEvent, int, double, MyClass);
#endif
```

Example 6.14: Separate Interface Style

Next, write a method with a signature matching the `paramList` in the associated event interface macro. Then, use the appropriate macro for the type of event created (analogous to sections 6.2 through 6.5). The event interface macros used are `DEFINE_SIMOBJ_EVENT`, `DEFINE_LOCAL_EVENT`, and `DEFINE_AUTONOMOUS_EVENT`. The parameters for all three macros are identical. The API for `DEFINE_SIMOBJ_EVENT` is shown below:

```
DEFINE_SIMOBJ_EVENT(eventName,
                    className,
                    methodName,
                    numParam)
```

| Parameter | Description |
|---|---|
| eventName | The same eventName which was used in the Macro DEFINE_EVENT_INTERFACE. |
| className | The name of the class that contains the method that is to be turned into an event. |
| methodName | The name of the method that is to be turned into an event. |
| numParam | The number of parameters used in the method being converted into an event (valid range is 0 to 8). |

Table 6.8: Macro DEFINE_SIMOBJ_EVENT (Interface) API

These macros differ from the merged interface style in that the macro names are shortened and a new parameter `numParam` replaces the `paramList` of the merged style. This is so the `paramList` can be contained in a separate interface file so as to allow the compiler to type-check both the event method and the schedulers with an independent file. Example 6.15 illustrates the same event defined in Example 6.1 above, but using the separate interface style.

```
#include "SpDefineSimObj.H"
#include "SpDefineEvent.H"
#include "I_MyEvent.H"

class S_MySimObj: public SpSimObj {
 public:
   MyMethod(int param1, double param2, MyClass param3);
};
DEFINE_SIMOBJ(S_MySimObj, 1, SCATTER);
DEFINE_SIMOBJ_EVENT(MyEvent, S_MySimObj, MyMethod, 3);
```

Example 6.15: Defining a Method Using the Separate Interface Method

The separate interface style uses the same event plug-in style as the merged interface style.

Finally, Example 6.16 demonstrates scheduling an event, as defined in Example 6.14, with a variable length argument representing the character string “`hello`”.

```
#include "I_MyEvent.H"
#include "SpSimTime.H"

MyScheduleObj::ScheduleEventForMySimObj(MyClass myObj) {
   SpSimTime time(3.2); // schedule event for time = 3.2
   SpObjHandle objHandle =
     GetObjHandle("S_MySimObj", 3); // kind Id = 3

   SCHEDULE_MyEvent(time, objHandle, 4, 6.5, myObj,
                    "hello", strlen("hello") + 1);
}
```

Example 6.16: Scheduling using the Separate Interface Style

## 6.8 Tips, Tricks, and Potholes

- It is recommended that the simulation object class contain only a small number of direct event methods (simulation object events). Any events that are self-scheduled should use the local event style. If the class is getting bogged down by too many event methods, making the class hard to maintain, it may make sense to convert some or all of those simulation object events to autonomous events.

- Do not use rollbackable types as local variables in events. Rollbackable types used as local variables cause run-time errors and application crashes. An error message, as shown in Figure 4.1, will be displayed when rollbackable types are used incorrectly. However, it is fine for local pointer variables to point to rollbackable variables.

- Local events will rarely benefit from using the separate interface style option, since they normally already require what the separate interface file is designed to remove: including the event method file. Schedulers of local events require a pointer to the event method's object, which presumes that the scheduler also needs the event method object's header file, regardless of the scheduling interface. The only exception to this is when a scheduler uses a forward class reference for the event method object pointer, which will probably be rare in practice.

- Never schedule an event in the past. SPEEDES will identify this situation, print an error message, and attempt to continue on. This condition will possibly result in an application crash or an infinite loop. One way to avoid this problem is to determine the schedule using `SpGetTime() + delta`. For example:

```
SpSimTime scheduleTime = SpGetTime();
...
scheduleTime += deltaTimeInFuture;
SCHEDULE_<eventName>(scheduleTime,...);
```

The `+=` operator on `SpSimTime` will preserve the priority fields that may have been set by other events.

If an event is scheduled in the past, then the following error message is output by SPEEDES.

```
Error, event list out of order
```

- Only attempt to cancel events that are in the future. Ensure that the physical part (i.e. the double part, ignoring the integer tie-breaking fields) of the scheduled time is greater than the physical part of the current time (i.e. `((double) SpGetTime() < ScheduledEventTime)`).

# Chapter 7

# Event Handlers

Event handlers differ from point-to-point events in two fashions. First, event handlers allow for both one-to-one events as well as one-to-many events. Second, event handlers can be run-time configured rather than compile time configured like point-to-point events. For example, a user could have one method registered for a given trigger and, at a later point, remove that method and add a different method to respond to that trigger or remove all sensitivity for that trigger.

As a more concrete example: a simulation object may create handlers out of two simulation object methods “Invincible” and “Vulnerable” through the `DEFINE_HANDLER` macro. At the start of the simulation, the simulation object may register the handler “Invincible” for the trigger “Check For Damage”. At a later point in the simulation, the simulation object can remove the “Invincible” handler and add in the handle “Vulnerable” to change its response.

Handler events may be scheduled for a specified simulation object (directed) or for the entire simulation (undirected). Directed handler events are analogous to point-to-point events in that they represent a one-way invocation from one simulation object to another. Undirected handler events are not point-to-point, but rather they are broadcasted out to all simulation objects that have subscribed to that trigger.

Event handler have two levels of filtering: a handler event type and an optional trigger string. The handler event type is determined at compile time, while the trigger string is determined at run time. The number of possible normal handler event triggers is unlimited, since after filtering by type, the second trigger is dynamically specified by string. However, since strings are specified dynamically, their binding to registered handlers cannot be verified at compile time. To achieve compile-time type verification, simply avoid passing the optional dynamic trigger string.

Handler types come in three styles: standard event handlers, interactions, and interface event handlers. The first two are predefined styles, while interface handler events are user-defined.

- Standard handler events are the simplest style and they send a variable-length buffer to recipients.
- Interactions differ from the standard style in that they send a variable-length, generic, parameterized data set using the `SpParmSet` rather than passing a variable length buffer as data.
- Finally, interface handler events use callback method signatures (as analogous to all three point-to-point event styles). In addition, interface handler events may send a variable length buffer to invoked handlers just like standard event handlers or point-to-point events.

The process for creating and using event handlers is simple when using the API. The basic steps for creating and using event handlers are:

1. Create an event handler from any method on any object in the application. This method can be on a simulation object (i.e. a child object of `SpSimObj`) or on any other object.
2. Apply one of the SPEEDES API macros to the method. This turns the method into an event handler that can then be scheduled by other events.
3. Register the event handler with the simulation object.
4. Schedule events for the simulation object, calling the event handler.

These items are discussed in more detail in the following sections.

## 7.1 Standard Event Handlers

Standard event handlers are the easiest to implement when no data is needed for the handlers. This is because there are no special interfaces to set up at compile time, and because their associated handler methods require no signature (i.e. standard handler methods have no parameters). However, when data does need to be sent, they are much harder to implement, since all data must be sent via a variable length buffer. This means standard event handlers are the most appropriate style when sending data is either unnecessary or erratic, and when a single, run-time level of handler filtering (i.e. trigger strings) is sufficient.

To create a standard event handler, the first step is to create a method on any object. The method should return a void and it cannot have any parameters. Once the method has been defined, then one of the event handler macros is applied to the method, which creates the event handler. The syntax for the macros are:

```
DEFINE_HANDLER(handlerName,
               className,
               methodName)
DEFINE_SIMOBJ_HANDLER(handlerName,
                      className,
                      methodName)
```

| Parameter | Description |
|---|---|
| handlerName | Any user-defined string representing the name of the event handler (legal characters for the handler name include alphanumeric and underscore characters). |
| className | The name of the class containing the method that is to be turned into an event handler. |
| methodName | The name of the method that is to be turned into an event handler. This method cannot have any parameters (i.e. input parameters must be void). |

Table 7.1: Macro DEFINE_HANDLER and DEFINE_SIMOBJ_HANDLER API

If the method being converted into a handler is located on a simulation object (i.e. a child of `SpSimObj`), then the macro `DEFINE_SIMOBJ_HANDLER` should be used when defining the handler. `DEFINE_HANDLER` must be used when the method is not on a simulation object. Use of these macros will result in new auto-generated classes whose name will be the name supplied in parameter `handlerName` appended by the string "`_HDR_ID`". Instances of this class are needed when registering and unregistering event handlers with simulation objects. Three methods on class `SpSimObj` allow users to add, subscribe, and remove event handlers. These methods are shown below:

```
AddHandler      (const SpHandlerId& handlerId,
                       char*        trigger = NULL)
SubscribeHandler(const SpHandlerId& handlerId,
                       char*        trigger = NULL)
RemoveHandler   (const SpHandlerId& handlerId,
                       char*        trigger = NULL)
```

| Parameter | Description |
|---|---|
| handlerId | This parameter is an instance of a macro-generated class created by macro `DEFINE_HANDLER` or `DEFINE_SIMOBJ_HANDLER`. A product of these macros is a class called `handlerName` (input parameter to the macro) appended by the string "`_HDR_ID`". When the macro `DEFINE_HANDLER` is used then the class instance containing the handler must be used as an argument to the macro built `_HDR_ID` class constructor. For example, suppose object `W` (non-simulation object) uses macro `DEFINE_HANDLER` with an input argument of `Abc`. Then the output from this macro will be a class called `Abc_HDR_ID`. Classes that want to register this handler for use must then create an instance of class `W` and use this instance as the `handlerId` for `AddHandler`, `RemoveHandler`, or `SubscribeHandler` (e.g. `Abc_HDR_ID(Winstance)`). When the macro `DEFINE_SIMOBJ_HANDLER` is used, the object for which the handler resides on is already known. Therefore, the object is not needed as input to the `_HDR_ID` class (i.e. `handlerId` will be `Abc_HDR_ID`). |
| trigger | Optional parameter which associates a string with the handler. If this parameter is provided, then the caller of this handler (i.e. scheduler) must also provide the trigger string in order for the event handler to be invoked. If an event handler is added or subscribed with a trigger, then it must also be removed with the same trigger. |

Table 7.2: Handler Methods Add, Subscribe, and Remove API

Choosing whether to use `AddHandler` or `SubscribeHandler` to register handlers with their simulation object affects the conditions under which they are called. When method `AddHandler` is used, then the handler will be invoked only when the scheduler schedules a directed event handler. If, on the other hand, method `SubscribeHandler` is used then the handler will be invoked when the scheduler schedules a directed or undirected event handler.

Directed event handlers specify an object handle for which the object on which the handlers should be called, hence the term "directed". In other words, directed event handlers are invoked on a specific simulation object (and sensitive to a particular trigger if the scheduler of the event handlers provided one). Conversely, schedulers of undirected handler events do not provide a particular object handle, but instead intend all handlers on all simulation objects in the simulation (that are sensitive to a particular trigger if the scheduler provided one) to be invoked. In other words, the handler event is not directed toward one particular simulation object, hence the term undirected.

The final step necessary for using event handlers is the scheduling of the handlers. This is done by using one of the following two global functions:

```
SpCancelHandle
SCHEDULE_HANDLER(const SpSimTime&   simTime,
                 const SpObjHandle& objHandle,
                 const char*        trigger   = NULL,
                 const char*        data      = NULL,
                       int          dataBytes = 0)

SpCancelHandle
SCHEDULE_HANDLER(const SpSimTime&   simTime,
                 const char*        trigger   = NULL,
                 const char*        data      = NULL,
                       int          dataBytes = 0)
```

| Parameter | Description |
|---|---|
| simTime | This parameter specifies the time at which the event handlers will be invoked. It must be at the present or future simulation time with respect to when it is scheduled (i.e. it cannot be scheduled in the simulation time past). |
| objHandle | This parameter uniquely specifies the simulation object for which handlers will be invoked. There are convenience functions provided in the SPEEDES framework which can look up object handles for simulation objects in SPEEDES. Please see Section 3.3 for additional information on class `SpObjHandle`. |
| trigger | This optional parameter specifies a trigger string to further filter which handlers are to be invoked. The trigger used must match the triggers specified when the handlers were registered using `AddHandler` or `SubscribeHandler`. |
| data | This optional parameter allows users to send a character buffer (can be binary or a character string) to the invoked handlers. |
| dataBytes | The parameter specifies the size of the optional data, if sent, in bytes. |

Table 7.3: SCHEDULE_HANDLER API

Notice that the above schedule functions come in two forms. When an object handle is provided as an argument to a schedule function, then directed event handlers are scheduled (i.e. handler on specific object). If a trigger is provided, then a handler with the exact trigger must have been registered on the receiving object. When the scheduling function is used without an object handle, undirected event handlers are being scheduled (i.e. all handlers on all objects which match the trigger specifications). Notice that event handlers can be scheduled without specifying an object handle or trigger. This will invoke all handlers on any object that were subscribed without triggers.

Let us revisit the car and stop light example first introduced in Chapter 6. Specifically, let us modify the example described in Section 6.3 to use event handlers. The code shown in Example 7.1 shows the new definition file for the stop light object. Changes made to the stop light definition file are:

1. Changed `SpDefineEvent.H` to `SpDefineHandler.H` (line 6).
2. Added class `H_Red` to process the red stop light signal (lines 7 through 11).
3. Turned method `H_Red::Red` into an event handler by using the macro `DEFINE_HANDLER` (line 12).
4. Added class `H_Green` to process the green stop light signal (lines 13 through 17).
5. Turned method `H_Green::Green` into an event handler by using macro `DEFINE_HANDLER` (line 18).

6. Deleted methods `Red` and `Green` from class `S_StopLight`, since these capabilities have been moved to class `H_Red` and `H_Green`.

7. Deleted event definition macro `DEFINE_SIMOBJ_EVENT_0_ARG` used to define events, since they are no longer needed.

```
// S_StopLight.H
#ifndef S_StopLight_H
#define S_StopLight_H

#include "SpSimObj.H"
#include "SpDefineSimObj.H"
#include "SpDefineHandler.H"

class H_Red {
  public:
    H_Red() {};
    void Red();
};
DEFINE_HANDLER(RedLight, H_Red, Red);

class H_Green {
  public:
    H_Green() {};
    void Green();
};
DEFINE_HANDLER(GreenLight, H_Green, Green);

class S_StopLight : public SpSimObj {
  public:
    S_StopLight() {};
    virtual ~S_StopLight() {};
    virtual void Init();
  protected:
  private:
    H_Red         *RedHandler;
    H_Green       *GreenHandler;
};
DEFINE_SIMOBJ(S_StopLight, 1, SCATTER);
#endif
```

Example 7.1: Standard Event Handler Stop Light Object Definition File

Notice that, in this example, macro `DEFINE_HANDLER` was used rather than `DEFINE_SIMOBJ_HANDLER`. The methods being turned into handlers are not on a simulation object (i.e. child of class `SpSimObj`). Rather, they are on self-contained objects `H_Red` and `H_Green`. By SPEEDES convention, handlers not on simulation objects, but rather self-contained in their own class definition are prefixed with an `H_`.

The code shown in Example 7.2 shows the new implementation file for the stop light object. Changes made to the stop light implementation are:

1. Added `#include "SpSchedule.H"` (line 5).

2. Delete macro `PLUG_IN_EVENT` used to define events `StopLight_TurnsRed` and `StopLight_TurnsGreen` since this functionality has been changed to event handlers.

3. Instantiated handlers `H_Red` and `H_Green` (lines 12 through 13). Added these handlers to the simulation object using method `AddHandler` (lines 14 through 15).

4. Implemented method `H_Red::Red` (lines 18 through 29).

5. Implemented method `H_Green::Green` (lines 30 through 41).

```
1  // S_StopLight.C
2  #include "SpGlobalFunctions.H"
3  #include "SpMainPlugIn.H"
4  #include "RB_ostream.H"
5  #include "SpSchedule.H"

6  #include "S_StopLight.H"
7  #include "S_Car.H"

8  void PlugInStopLight() {
9    PLUG_IN_SIMOBJ(S_StopLight);
10 }

11 void S_StopLight::Init() {
12   RedHandler   = new H_Red;
13   GreenHandler = new H_Green;
14   AddHandler(RedLight_HDR_ID(*RedHandler));
15   AddHandler(GreenLight_HDR_ID(*GreenHandler), "Light Turned Green");
16   SCHEDULE_HANDLER(30.0, SpGetObjHandle());
17 }

18 void H_Red::Red() {
19   int i;
20   for (i = 0; i < 4; ++i) {
21     SpObjHandle objHandle = SpGetObjHandle("S_Car_MGR", i);
22     SCHEDULE_Car_Stop(SpGetTime(), objHandle, SpGetTime(),
23                       (char *) "Red Light On",
24                       strlen("Red Light On") + 1);
25   }
26   SCHEDULE_HANDLER(SpGetTime() + 30.0, SpGetObjHandle(),
27                    "Light Turned Green",
28                    (char *) "Go", strlen("Go") + 1);
29 }

30 void H_Green::Green() {
31   int i;
32   RB_cout << "H_Green::Green Message Data= "
33           << SpGetMsgData() << endl;
34   for (i = 0; i < 4; ++i) {
35     SpObjHandle objHandle = SpGetObjHandle("S_Car_MGR", i);
36     SCHEDULE_Car_Go(SpGetTime(), objHandle,
37                     (char *) "Green Light On",
38                     strlen("Green Light On") + 1);
39   }
40   SCHEDULE_HANDLER(SpGetTime() + 30.0, SpGetObjHandle());
41 }
```

Example 7.2: Standard Event Handler Stop Light Object Implementation File

This example contains several interesting differences, which require further discussion. Notice that, since the handler methods are not contained in the simulation object, the classes containing the handler methods must be instantiated prior to use (lines 12 and 13). If these methods were on the simulation object, then this step would not be necessary (as shown later).

The next step for using the handlers is to register them with the simulation object. This is done with method `AddHandler` (lines 14 and 15). Since `AddHandler` was used in the above example, these handlers can only be invoked through directed event handlers (i.e. scheduler must specify the object handle for the stop light object). If `SubscribeHandler` had been used, then this object would be capable of receiving undirected events.

Also, notice that the red handler is added without a trigger string (line 14) and the green handler is registered with a trigger string (line 15). This requires the scheduler to use the appropriate trigger when scheduling this handler. The handler object is specified in the "`*_HDR_ID`" constructor when adding the handler (lines 14 and 15). This is due to the handler methods not being on the simulation object. Finally, the global function `SCHEDULE_HANDLER` is called to schedule a handler. The object handle for the stop light is specified with no trigger which means the red handler will be executed at time 30.0.

The red handler shown on lines 18 through 29 is very similar to the `red` method described in Section 6.3. The only difference is in the way the events are scheduled (lines 26 through 28). In this example, the global function `SCHEDULE_HANDLER` is used to schedule the green event handler, as opposed to function `SCHEDULE_StopLight_TurnsGreen` described in the point-to-point example. `AddHandler` was used to add the handler to the simulation object with a trigger. Consequently, the stop light's object handle must be specified in the function `SCHEDULE_HANDLER` along with the appropriate trigger (i.e. "`Light Turned Green`"). Finally, for example purposes, the string "`Go`" was sent as optional data.

The green handler shown on lines 30 through 41 is very similar to the `green` method described in Section 6.3. Once again, there is a difference with the scheduling of the red event handler on the stop light (shown on line 40). The differences are the same as described in the previous paragraph. However, since the red handler was added without a trigger, it is scheduled without one as well. Lines 32 and 33 were added so that this handler can print out the optional data that was sent from the red event handler.

## 7.2 Interaction Event Handlers

Interactions are similar to standard handler events, except in the way they pass data. Interaction schedulers and interaction handlers both use one variable-length parameter of type `SpParmSet`, which allows schedulers to send any number of primitive data types and non-typed buffers to interaction handlers.

Thus, interactions are preferred over using standard handler events, when their handlers need primitive typed parameters passed to them. The interaction style is particularly useful when handlers need different sets of parameters each time they are called. The class `SpParmSet` is designed to handle varying sets of parameters.

The SPEEDES framework provides macros which turn object methods into interaction handlers. These macros are similar to the macros described in Section 7.1. The macros used to define interaction handlers are:

```
DEFINE_INTERACTION_HANDLER(handlerName,
                           className,
                           methodName);
DEFINE_SIMOBJ_INTERACTION_HANDLER(handlerName,
                                  className,
                                  methodName);
```

The descriptions of these parameters are the same as those for `DEFINE_HANDLER`, which are described in Table 7.1, with one exception. Method `methodName` must have one argument of type `SpParmSet`.

In order to schedule an interaction handler, the following global functions are provided:

```
SpCancelHandle
SCHEDULE_INTERACTION(const SpSimTime&   simTime,
                     const SpObjHandle& objHandle,
                     const char*        trigger,
                     const SpParmSet&   parmSet)

SpCancelHandle
SCHEDULE_INTERACTION(const SpSimTime&   simTime,
                     const char*        trigger,
                     const SpParmSet&   parmSet)
```

| Parameter | Description |
|---|---|
| simTime | This parameter specifies the time at which the event handlers are invoked. It must be at the present or future simulation time with respect to when it is scheduled (i.e. it cannot be scheduled in the simulation time past). |
| objHandle | This parameter uniquely specifies the simulation object on which handlers will be invoked. There are convenience functions provided in the SPEEDES framework which can look up object handles for simulation objects in SPEEDES. Please see Section 3.3 for additional information on class `SpObjHandle`. |
| trigger | This optional parameter specifies a trigger string to filter which handlers are to be invoked. The trigger used must match the triggers specified when the handlers were registered using `AddHandler` or `SubscribeHandler`. |
| parmSet | This parameter contains the data which is passed along to the interaction handler. Class `SpParmSet` allows users to build data structures of non-pointer type data (i.e. primitive types such as integers, doubles, and other classes, which contain only primitive base types and strings). |

Table 7.4: SCHEDULE_INTERACTION API

Methods `AddHandler`, `SubscribeHandler`, and `RemoveHandler` are used to add, subscribe, and remove interaction event handlers in a similar fashion as described in Section 7.1. The “`trigger`” parameter is not an optional parameter when adding, subscribing, or removing interaction handlers.

Let us continue enhancing the car and stop light simulation to illustrate interaction event handlers. Also, some additional standard event handler capabilities are provided. Example 7.3 shows some additional changes made to the stop light object which include:

1. Lines 19 through 25 in Example 7.2 were replaced with lines 19 through 22 in Example 7.3 as shown below.

2. Lines 34 through 39 in Example 7.2 were replaced with lines 30 and 31 in Example 7.3 as shown below.

```
// S_StopLight.C
#include "SpGlobalFunctions.H"
#include "SpMainPlugIn.H"
#include "RB_ostream.H"
#include "SpSchedule.H"

#include "S_StopLight.H"
#include "S_Car.H"

void PlugInStopLight() {
  PLUG_IN_SIMOBJ(S_StopLight);
}

void S_StopLight::Init() {
  RedHandler   = new H_Red;
  GreenHandler = new H_Green;
  AddHandler(RedLight_HDR_ID(*RedHandler));
  AddHandler(GreenLight_HDR_ID(*GreenHandler), "Light Turned Green");
  SCHEDULE_HANDLER(30.0, SpGetObjHandle());
}

void H_Red::Red() {
  SpParmSet tmpParm;
  tmpParm.InsertDouble("FirstData", SpGetTime());
  tmpParm.InsertString("SecondData", "Red Light On");
  SCHEDULE_INTERACTION(SpGetTime(), "Red", tmpParm);

  SCHEDULE_HANDLER(SpGetTime() + 30.0, SpGetObjHandle(),
                   "Light Turned Green",
                   (char *) "Go", strlen("Go") + 1);
}

void H_Green::Green() {
  RB_cout << "H_Green::Green Message Data= "
          << SpGetMsgData() << endl;

  SCHEDULE_HANDLER(SpGetTime(), "Green", "Green Light On",
                   strlen("Green Light On") + 1);
  SCHEDULE_HANDLER(SpGetTime() + 30.0, SpGetObjHandle());
}
```

Example 7.3: Interaction Event Handler Stop Light Object Implementation File

The stop light simulation object has been modified such that it now communicates with the car simulation objects via handlers instead of point-to-point events. Methods `Red` and `Green` have both been modified by replacing the `for` loop with a call to a schedule handler. If global communication is needed for many objects, handlers are often a good communication choice.

Method `Red` has been modified to send an interaction handler with data (i.e. parameter `SpParmSet` has been initialized). Since the object handle for the call to `SCHEDULE_INTERACTION` has not been used, an undirected interaction handler is being sent. The trigger for this call is “`Red`”. Therefore, all simulation objects that have registered a handler with a trigger of “`Red`” will be invoked.

Method `Green` has been modified to send a standard handler with string data (i.e. “`Green Light On`”). Since the object handle for the call to `SCHEDULE_INTERACTION` has not been used, an undirected interaction handler is being sent. The trigger for this call is “`Green`”. Therefore, all simulation objects that have registered a handler with a trigger of “`Green`” will be invoked.

The car simulation object has been modified so that it will invoke the appropriate event handlers that are scheduled by the stop light simulation object. Example 7.4 shows the changes made to the car object definition file. The changes made to the definition file were:

1. Changed `SpDefineEvent.H` to `SpDefineHandler.H` (line 6).
2. Added `#include "RB_SpString.H"` (line 7).
3. Changed method `Stop` parameter to `(SpParmSet& interactionData)` (line 13). This is required, since method `Stop` is being converted to an interaction event handler.
4. Added attribute `RB_SpString TriggerString;` (line 24). This attribute keeps track of the current trigger used for method `StopCar`.
5. Changed event `Car_Stop` from a point-to-point event to an interaction event handler using macro `DEFINE_SIMOBJ_INTERACTION_HANDLER` (line 28).
6. Changed events `Car_Go` and `Car_StopCar` from point-to-point events to standard event handlers using macro `DEFINE_SIMOBJ_HANDLER` (line 29 through 30).

```
 1  // S_Car.H
 2  #ifndef S_Car_H
 3  #define S_Car_H

 4  #include "SpSimObj.H"
 5  #include "SpDefineSimObj.H"
 6  #include "SpDefineHandler.H"
 7  #include "RB_SpString.H"

 8  class S_Car : public SpSimObj {
 9    public:
10      S_Car() {};
11      virtual ~S_Car() {};

12      virtual void Init();
13      void         Stop(SpParmSet& interactionData);
14      void         Go();
15      void         StopCar();

16    protected:
17    private:
18      RB_double         XPos;               // Car Position
19      double            Velocity;           // Car Velocity
20      RB_double         LastTimeStopped;    // Time at last Car Stoppage
21      RB_int            StopState;          // 0 = Stopped; 1 = Moving
22      RB_int            TimesStopped;       // Number of times Car
23                                            //  Stopped
24      RB_SpString       TriggerString;      // The trigger string for
25                                            //  Car_StopCar handler
26  };

27  DEFINE_SIMOBJ(S_Car, 4, SCATTER);
28  DEFINE_SIMOBJ_INTERACTION_HANDLER(Car_Stop, S_Car, Stop);
```

```
DEFINE_SIMOBJ_HANDLER(Car_Go, S_Car, Go);
DEFINE_SIMOBJ_HANDLER(Car_StopCar, S_Car, StopCar);
#endif
```

Example 7.4: Interaction Event Handler Car Object Definition File

Notice that this example uses `DEFINE_SIMOBJ_HANDLER` as opposed to `DEFINE_HANDLER`, as shown in Section 7.1. This is because the methods being converted into handlers are located on the car simulation object. This allows for additional compile time type checking when adding, subscribing, or removing handlers.

Example 7.5 shows the changes made to the car object implementation file. The specific changes made to the implementation file are as follows:

1. Deleted the macros for plugging in the point-to-point events, since these events have been converted to event handlers.
2. Added code which initializes the event handlers (line 17 through 26).
3. Changed method `Stop` parameter to `(SpParmSet& interactionData)` (line 27).
4. Added code to display the data in `interactionData` (line 37 through 44).
5. Modified the code required for scheduling the old point-to-point event `Car_StopCar` to a standard event handler schedule (line 45 through 51).
6. Modified code in method `Go` for processing the green stop light signal (line 54 through 61).

```
1  // S_Car.C
2  #include "SpGlobalFunctions.H"
3  #include "SpMainPlugIn.H"
4  #include "RB_ostream.H"
5  #include "SpSchedule.H"

6  #include "S_Car.H"

7  static int QuarterMile = 5280 / 4; // In feet

8  void PlugInCars() {
9    PLUG_IN_SIMOBJ(S_Car);
10 }

11 void S_Car::Init() {
12   XPos             = 0;
13   Velocity         = (SpGetSimObjGlobalId() * 10.0 + 11.0) / 3600 * 5280;
14   StopState        = 0;
15   LastTimeStopped = 0.0;
16   TimesStopped     = 0;

17   SubscribeHandler(Car_Stop_HDR_ID(), "Red");
18   SubscribeHandler(Car_Go_HDR_ID(), "Green");
19   TriggerString = "TOA";
20   char tempStr[80];
21   strcpy(tempStr, TriggerString);
22   AddHandler(Car_StopCar_HDR_ID(), tempStr);
```

```
  SCHEDULE_HANDLER(0.0, SpGetObjHandle(),
                        "Green", "Green On",
                        strlen("Green On") + 1);
}

void S_Car::Stop(SpParmSet& interactionData) {
  double currentXPos;
  int    nextStopLightPos;
  double stopLightArrivalTime;

  currentXPos          =
       (XPos + Velocity * (SpGetTime() - LastTimeStopped));
  nextStopLightPos     =
       (int) ((((int) currentXPos) / QuarterMile + 1) * QuarterMile);
  stopLightArrivalTime =
         (nextStopLightPos - currentXPos) / Velocity;
  RB_cout << "Received stop signal for Car #" << SpGetSimObjGlobalId()
          << ", Received argument= "
          << interactionData.GetDouble("FirstData")
          << ", Sim" << SpGetTime()
          << " Received Data= "
          << interactionData.GetString("SecondData")
          << " stopLightArrivalTime= " << stopLightArrivalTime
          << endl;

  char tempStr[80];
  sprintf(tempStr, "TOA%.4f",
          SpGetRandom()->GenerateDouble(0.0, 100.0));
  TriggerString = tempStr;
  AddHandler(Car_StopCar_HDR_ID(), tempStr);
  SCHEDULE_HANDLER(SpGetTime() + stopLightArrivalTime,
                   SpGetObjHandle(), tempStr);
}

void S_Car::Go() {
  char tempStr[80];
  RB_cout << "S_Car::Go Message Data= " << SpGetMsgData() << endl;
  strcpy(tempStr, TriggerString);
  RemoveHandler(Car_StopCar_HDR_ID(), tempStr);
  if (StopState == 0) {
    LastTimeStopped = SpGetTime();
    StopState       = 1;
  }
}

void S_Car::StopCar() {
  XPos = XPos + (Velocity * (SpGetTime() - LastTimeStopped));
  StopState = 0;
  ++TimesStopped;

  RB_cout << "Car #" << SpGetSimObjGlobalId()
          << " is stopping at " << SpGetTime()
          << " at X coordinate of " << XPos
          << ". Car has stopped " << TimesStopped << " times."
          << " LastTimeStopped= " << LastTimeStopped
          << endl;
```

```
}
```

Example 7.5: Interaction Event Handler Car Object Implementation File

Let us examine this example in more detail. On lines 17 and 18, subscriptions are made for processing the red and green stop light signal. Method `SubscribeHandler` is used to register these handlers with the SPEEDES framework, which allows both directed and undirected schedules to be used. For this example, the scheduler is scheduling undirected event handlers, as shown on lines 22 and 30 in Example 7.3. Zero parameters are passed to the "`*_HDR_ID`" constructor. This is because macros `DEFINE_SIMOBJ_HANDLER` and `DEFINE_SIMOBJ_INTERACTION_HANDLER` were used to define the handlers. Lines 19 through 22 add the handler that will handle the car when it actually stops. Only directed events can cause this handler to be invoked.

Method `Stop` is used to process the red stop light signal, which was sent via an interaction event handler (Example 7.3 line 22). The code for calculating the car's position remains unchanged. Some additional code was added to extract the interaction data. Lines 45 through 51 are used to schedule the handler for the actual stopping of the car. Recall that Section 6.3 described how to use cancel handles to cancel events. This example could have used the same technique to cancel unnecessary events. However, for illustration purposes, this example adds a handler with a unique trigger. If the event needs to be canceled, then the handler is removed. The event still goes off, but since the event handler was removed, the code for processing the stopped car is not executed, thus producing the correct results. Lines 45 through 49 were added to calculate the unique handler trigger name and register the handler with the SPEEDES framework.

The counterpart to method `Stop` is method `Go`. Every time this method is executed, the handler for stopping the car is removed. This prevents the car from stopping at green lights.

## 7.3 Interface Event Handlers

The last type of handler is called the interface event handler. This type of handler allows handler methods to contain multiple arguments, as opposed to the zero argument standard event handlers previously discussed.

The process for using interface handlers requires the use of two macros. The first macro defines the interface name, number of arguments that the method takes and their types. This method has the following API:

```
DEFINE_HANDLER_INTERFACE_<numParam>_ARG(interfaceName,
                                        [paramList])
```

| Parameter | Description |
|---|---|
| numParam | The number of parameters used in the method being converted into an interface event handler (valid range is 0 to 8). |
| interfaceName | Any user-defined string representing the name of the interface (legal characters for the handler include alphanumeric and underscore characters). |
| paramList | Comma delimited list of the parameter types found in the method. The types specified can be any primitive base type (i.e. integer, double, etc.) or class whose attributes contain primitive base types. Pointer types are not allowed. |

Table 7.5: Macro DEFINE_HANDLER_INTERFACE with Arguments API

The interface signature can be placed in its own independent header file. Then, by including this header file, the arguments to the schedulers and parameters to the handler methods are type checked.

After the interface signature (or type) has been defined, the macro for turning an object method into an interface event handler can be used. The API for these macros is:

```
DEFINE_INTERFACE_HANDLER(handlerName,
                         className,
                         methodName,
                         interfaceName,
                         numParam)
DEFINE_SIMOBJ_INTERFACE_HANDLER(handlerName,
                                className,
                                methodName,
                                interfaceName,
                                numParam)
```

| Parameter | Description |
|---|---|
| handlerName | Any user-defined string representing the name of the event handler. (legal characters for the handler include alphanumeric and underscore characters.) |
| className | The name of the object or simulation object class that contains the method that is to be turned into an event handler. |
| methodName | The name of the method that is to be turned into an interface event handler. |
| interfaceName | This parameter uses the name which was defined with macro `DEFINE_HANDLER_INTERFACE_<numParam>_ARG` above. |
| numParam | This parameter is number of parameters specified when using the macro `DEFINE_HANDLER_INTERFACE_<numParam>_ARG`. |

Table 7.6: Macro DEFINE_INTERFACE_HANDLER API

The code shown in Example 7.6 shows an example of how to use these macros.

```
#ifndef S_My_Sim_Obj_H
#define S_My_Sim_Obj_H

#include "SpSimObj.H"
#include "SpDefineSimObj.H"
#include "SpDefineHandler.H"

class  S_My_Sim_Obj : public SpSimObj {
  public:
    S_My_Sim_Obj() {};
    virtual ~S_My_Sim_Obj() {};

    void MyMethod(int arg1, double arg2);
};
DEFINE_SIMOBJ(S_My_Sim_Obj, 1, SCATTER);
DEFINE_HANDLER_INTERFACE_2_ARG(MyInterfaceDefinition, int, double)
DEFINE_SIMOBJ_INTERFACE_HANDLER(MyHandlerName,
                                S_My_Sim_Obj,
                                MyMethod,
                                MyInterfaceDefinition,
                                2);
#endif
```

Example 7.6: Interface Event Handler

A byproduct of the `DEFINE_INTERFACE_HANDLER` and `DEFINE_SIMOBJ_INTERFACE_HANDLER` is a function which is used for scheduling interface handlers. The API for this function is:

```
SpCancelHandle
SCHEDULE_HANDLER_<interfaceName>(const SpSimTime&   simTime,
                                 const SpObjHandle& objHandle,
                                                    [paramList],
                                 const char         trigger = NULL,
                                 const char*        data = NULL,
                                 int                dataBytes = 0);

SpCancelHandle
SCHEDULE_HANDLER_<interfaceName>(const SpSimTime&   simTime,
                                                    [paramList],
                                 const char         trigger = NULL,
                                 const char*        data = NULL,
                                 int                dataBytes = 0);
```

| Parameter | Description |
|---|---|
| interfaceName | The interface name was defined by macro `DEFINE_HANDLER_INTERFACE`. |
| simTime | This parameter specifies the time at which the scheduled handler event will be invoked. This time must be at the present time or future time (i.e. it cannot be in the past). |
| objHandle | This parameter uniquely specifies the simulation object on which handlers will be invoked. There are convenience functions provided in the SPEEDES framework which can look up object handles for simulation objects in SPEEDES. Please see Section 3.3 for additional information on class `SpObjHandle`. |
| paramList | A comma delimited list of the parameters that are to be passed to the simulation object method (e.g. in the above example data parameters passed to `MyMethod` should be an integer and double). |
| trigger | This optional parameter specifies a trigger string, which allows additional filtering to occur when handlers are invoked. The trigger used must match the triggers specified when the handlers were registered using `AddHandler` or `SubscribeHandler`. |
| data | This optional parameter allows users to send a character buffer (can be binary or a character string) to the invoked handlers. |
| dataBytes | This parameter specifies the size of the optional data, if sent, in bytes. |

Table 7.7: Function SCHEDULE_HANDLER API

For example, the scheduling function for Example 7.6 would be `SCHEDULE_HANDLER_MyInter-faceDefinition`. Of course, in order for this function to work, the simulation object must have used method `AddHandler` or `SubscribeHandler` to register `MyHandlerName`.

## 7.4 Tips, Tricks, and Potholes

- While handler events are powerful, do not assume they are always a better event paradigm than point-to-point events. Point-to-point events have the advantage that schedulers know exactly what code will be invoked each time they schedule an event. Handlers offer no such assurance, since handlers can be added and removed dynamically at any point in simulated time. Also, use of undirected event handlers typically increases the number of events processed (i.e. internal SPEEDES events).

Chapter 8

# The Process Model

The process model extends normal C++ methods into specialized algorithms called “processes”. When so transformed, an algorithm inside such a method evolves over a simulated time span, rather than occurring at a single instant in simulation time.

In SPEEDES, a “process” is a point-to-point event that uses special macros that allow execution of code to be exited at at any point and reentry to the code at the exit point at some later simulated time. This is done without losing local variable state or algorithmic context.

Traditional discrete-event models are often called “event-based”, while models written mainly using processes are often called “process-based”. Writing a process-based simulation is often simpler, more intuitive, and easier to maintain than writing the same simulation in the event-based paradigm. This is because processes are able to model a thread of control as a single algorithm, while event-based models must jump from one piece of code to another whenever simulation time advances.

The event-based paradigm maps well to passive models. Events serve as callbacks to modify a passive model’s state when something happens. For event-based passive models, saving state data to preserve context from one event to another suffices.

However, active models have another type of “state”: the state of each process used to self-propagate their state themselves, such as a heat-sensing missile. In addition to preserving state data context, process-based models also preserve the algorithmic context of their self-propagating processes. This enables developers to write algorithms that map naturally to the processes that animate those models. Thus, it often makes sense to design active models using the process model.

In practice, it usually makes sense to combine both event-based semantics and process-based semantics in a single simulation. SPEEDES supports not only mixing both paradigms in the same simulation, but also mixing event-based semantics and process-based semantics within a single process. Since a SPEEDES process is an abstraction that uses point-to-point events as its implementation, processes and normal events work together using the common SPEEDES event-based simulation engine.

## 8.1 Process Model API

The following sections describe the API for the process model, which includes:

1. Process model initializers.
2. Wait reentry points (similar to UNIX’s “`sleep`” command).

3. Semaphore reentry points.

4. Ask reentry points.

### 8.1.1 Required Process Model Initializers

To use the process model, an event must be written that contains process model initializer constructs and one or more process model reentry point constructs. Process model reentry points mark where in the user algorithm the process model will jump to upon process model reentry. All process model events must contain, at a minimum, the following three structural macros:

```
P_VAR;
P_BEGIN(numReentryLabels);
P_END;
```

`P_VAR` is always the first macro in the event. `P_BEGIN` marks the beginning of process model user-code algorithm. Macro parameter `numReentryLabels` is an integer number that defines the number of process model reentry point constructs to be used in this event. For example, if the process model contains two reentry points, then 2 would be the parameter used in macro `P_BEGIN`. Macro `P_END` marks the end of the process model event and is usually placed at the bracket prior to the method exit.

Process models can also define local state or stack variables. These variables are defined between the macros `P_VAR` and `P_BEGIN`. To define a local state variable, the following macro is used:

```
P_LV(type, name);
```

| Parameter | Description |
|---|---|
| type | Any primitive base-type (e.g. `int`, `double`, etc.), class or pointer. It cannot be a rollbackable type (e.g. `RB_int`). Instead, all types placed here, will be turned into a rollbackable type automatically, since the process model saves all of this data in a rollbackable manner. While rollbackable types cannot be used here, pointers to such types can be used. Most often, these rollbackable types are encapsulated in a class for which a pointer type is created and set to point to this class instance. Also, if a class is used here as the type, then the class can be made up of primitive base-types or pointers to rollbackable types. The class cannot contain pointers to non-rollbackable data. |
| name | Local state variable name. |

Table 8.1: Macro P_LV API

Process model reentry point macros are the final constructs necessary to implement process model functionality. All process model reentry point macros have one item in common, which is the first parameter for the macro. The first parameter must be an integer, which is unique for each process model construct. This id is a structural parameter that allows the macro to create a unique and identifiable process model reentry point. This enables the process model to be reentrant by allowing the code to restart execution at the exit point (i.e. previous execution point in the code). Wherever the process model exits and reenters, users need to place a unique integer beginning with 1 and ending with the total number of reentry labels. The code shown in Example 8.1 illustrates this, as well as the other process model initializer macros.

```
#ifndef MyClass_H
#define MyClass_H

#include "SpDefineEvent.H"
#include "SpProc.H"
#include "RB_SpBinaryTree.H"

class MyClass {
  public:
    void MyMethod(int param1, double param2) {
      /*
       * Process model variable definitions start here.  Define all
       * process model state variables between P_VAR and P_BEGIN
       * using macro P_LV.
       */
      P_VAR;
      P_LV(double, a);            // double state variable
      P_LV(RB_SpBinaryTree*, b);  // RB_SpBinaryTree pointer state
                                  //  variable
      int i;                      // Local non-state variable
      /*
       * P_BEGIN marks the end of the variable definition area and
       * the start of the user-defined process model algorithm.
       */
      P_BEGIN(2);
      /*
       * User-defined algorithm is added between P_BEGIN and P_END.
       * This example should contain 2 process model reentry points.
       * P_END marks the end of the user-defined process model algorithm.
       */
      P_END;
   }
};

DEFINE_LOCAL_EVENT_2_ARG(MyProcess, MyClass, MyMethod, int, double);
#endif
```

Example 8.1: Basic Process Model Form

### 8.1.2 Wait (a.k.a. Sleep) Process Model Reentry Points

The most basic constructs in the process model are the `WAIT` and `WAIT_UNTIL` reentry point constructs. These constructs mimic the UNIX `sleep` command. Specifically, `WAIT` will wait the specified amount of time prior to process model continuation (as measured by GVT, not wall clock time). `WAIT_UNTIL` waits until the specified GVT has been reached before process model continuation. The API for these constructs is shown below:

```
  WAIT(reentryLabel, duration);
  WAIT_UNTIL(reentryLabel, simTime);
```

| Parameter | Description |
|---|---|
| reentryLabel | Integer id for the appropriate process model reentry label. For example, if `P_BEGIN` defines the number of reentry labels to be 3, then there should be 3 process model reentry constructs whose labels are 1, 2 and 3. |
| duration | Specifies the amount of time, in seconds, that will elapse (i.e. GVT, not wall clock time) before process will continue. |
| simTime | Specifies the simulation time when processing will continue. For example, if GVT is 10.0 seconds and simTime is set to 100.0, then the code directly following `WAIT_UNTIL` will be executed (i.e. committed) when GVT reaches 100.0 seconds. If the time specified is in the past of the current simulation time, then the `WAIT_UNTIL` construct is ignored. |

Table 8.2: Macro WAIT and WAIT_UNTIL API

To illustrate these process model constructs, let us revisit the car and stop light example described in Chapter 6. The code shown in Examples 6.2 and 6.3 shows the original stop light simulation object design. This object will be redesigned to use the process model `WAIT` and `WAIT_UNTIL` constructs. The new design for the stop light simulation object now only requires one event, as compared to the two events in the previous design. Therefore, in the definition file, methods `Red` and `Green` have been replaced with method `Signal`. Example 8.2 shows the new definition code for the stop light simulation object.

```
// S_StopLight.H
#ifndef S_StopLight_H
#define S_StopLight_H

#include "SpSimObj.H"
#include "SpDefineSimObj.H"
#include "SpDefineEvent.H"

class S_StopLight : public SpSimObj {
  public:
    S_StopLight() {};
    virtual ~S_StopLight() {};

    virtual void Init();
    void         Signal();

  protected:
  private:
    RB_int TimesChanged;
};

DEFINE_SIMOBJ(S_StopLight, 1, SCATTER);
DEFINE_SIMOBJ_EVENT_0_ARG(StopLight_Signal, S_StopLight, Signal);
#endif
```

Example 8.2: Process Model Sleep Stop Light Object Definition File

The implementation for method `Signal` is shown on lines 16 through 44 in Example 8.3. Line 23 shows the `WAIT` construct being used with a wait of 30.0 seconds. When processing continues after the `WAIT`, an event is scheduled on each car object, notifying the cars that the light has turned red. Line 35 shows the `WAIT_UNTIL` construct being used with a `simTime` of “now” plus 30.0 seconds (i.e. sleep for 30.0 seconds). After the WAIT_UNTIL, an event is scheduled on each car instance indicating that the light has turned green. Lines 26 through 28 output the values of the process model local state variable and the simulation object state variable. The two state variables contain the number of times the light has turned red. These two values should always be equal. Other changes made to the implementation file are the plugging in of the correct event (line 10) and the scheduling of the first StopLight_Signal event (line 14).

```
 1  // S_StopLight.C
 2  #include "SpGlobalFunctions.H"
 3  #include "SpMainPlugIn.H"
 4  #include "RB_ostream.H"
 5  #include "SpProc.H"

 6  #include "S_StopLight.H"
 7  #include "S_Car.H"

 8  void PlugInStopLight() {
 9    PLUG_IN_SIMOBJ(S_StopLight);
10    PLUG_IN_EVENT(StopLight_Signal);
11  }

12  void S_StopLight::Init() {
13    TimesChanged = 0;
14    SCHEDULE_StopLight_Signal(0.0, SpGetObjHandle());
15  }

16  void S_StopLight::Signal() {
17    P_VAR;
18    P_LV(int, changes);
19    int i;
20    P_BEGIN(2);
21    changes = 0;
22    for (;;) {
23      WAIT(1, 30.0);
24      ++changes;
25      ++TimesChanged;
26      RB_cout << "Event StopLight_Signal local state variable= " << changes
27              << " should be equal to class state variable= "
28              << TimesChanged << endl;
29      for (i = 0; i < 4; ++i) {
30        SpObjHandle objHandle = SpGetObjHandle("S_Car_MGR", i);
31        SCHEDULE_Car_Stop(SpGetTime(), objHandle,
32                          (char *) "Red Light On",
33                          strlen("Red Light On") + 1);
34      }

35      WAIT_UNTIL(2, SpGetTime() + 30.0);
36      for (i = 0; i < 4; ++i) {
37        SpObjHandle objHandle = SpGetObjHandle("S_Car_MGR", i);
38        SCHEDULE_Car_Go(SpGetTime(), objHandle,
39                        (char *) "Green Light On",
40                        strlen("Green Light On") + 1);
41      }
42    }
43    P_END;
}
```

Example 8.3: Process Model Sleep Stop Light Object Implementation File

In order to execute this example, the main and car simulation objects introduced in Section 6.3 are needed. Method `S_Car::Stop` was modified by removing its input argument. The define event macro used for turning the method into an event was changed to use the 0 argument macro.

### 8.1.3 Semaphore Process Model Reentry Points

The process model `WAIT_FOR` and `WAIT_FOR_RESOURCE` constructs work with classes called semaphores. These constructs break out of their waits based on the setting of a semaphore or after the specified wake-up time has expired. The API for the semaphore process model constructs is:

```
WAIT_FOR(reentryLabel, semaphore, timeOut);
WAIT_FOR_RESOURCE(reentryLabel, semaphore, amount, timeOut);
```

| Parameter | Description |
|---|---|
| reentryLabel | Integer id for the appropriate process model reentry label. For example, if `P_BEGIN` defines the number of reentry labels to be 3, then there should be 3 process model reentry constructs whose labels are 1, 2 and 3. |
| semaphore | This parameter must be of type `SpLogicalSem` or `SpCounterSem` for `WAIT_FOR` and of type `SpResourceSem` for `WAIT_FOR_RESOURCE`. When set, these semaphores cause the appropriate process model construct to wake-up and continue processing. |
| timeOut | This parameter specifies when the process model `WAIT_FOR` construct may break out of its wait and continue processing. If the value for this parameter is greater than 0, then the `WAIT_FOR` construct waits this amount of time or until the semaphore is set, whichever is shorter. If the parameter is negative, there is no timeout period (i.e. the only way to continue processing is for the semaphore to be set). Finally, a value of 0 has no meaning for `WAIT_FOR` (i.e. if 0 is used, then the `WAIT_FOR` construct is ignored and processing continues). |
| amount | This parameter, only found in `WAIT_FOR_RESOURCE`, specifies how much of the `SpResourceSem` resource is desired (i.e. being waited for). Once that amount of resource becomes available or the timeout condition occurs, the process continues. If the resource amount desired becomes available before the timeout condition, that amount is subtracted from the resource. Currently, there is no built-in way to determine which of these conditions caused the process to continue, unless the timeout condition is negative, in which case continuing implies the resource did become available. |

Table 8.3: Macro WAIT_FOR and WAIT_FOR_RESOURCE API

The classes `SpLogicalSem`, `SpCounterSem`, and `SpResourceSem` are an integral part of the process model `WAIT_FOR` construct. Semaphores allow other events to interrupt the process model during its wait period. Process models use the `WAIT_FOR` construct to wait for either:

- A semaphore to be set, which causes processing to continue.
- The timeout time to expire, which causes processing to continue.

When using the `WAIT_FOR` construct, users may specify a maximum simulated time interval to wait for the semaphore to be set. A description of each semaphore is described in Table 8.4.

| Semaphore | Description |
|---|---|
| SpLogicalSem | This semaphore can have one of two values, true or false. These values are represented by a non-zero value for true and a zero value for false. Objects of this type may be used as booleans (or technically, integers) in equations and assignments. |
| SpCounterSem | This semaphore is initialized to a non-negative value. This semaphore returns 0 when its value is greater than zero and returns 1 once the counter reaches 0. When zero, processes wait for this semaphore. When non-zero, the processes are released. Objects of this type may be used as integers in equations and assignments. |
| SpIntSem and SpDoubleSem | These semaphores are resources that are initialized to non-negative values (integer for `SpIntSem` or double for `SpDoubleSem`) that represents the amount of resource available. These semaphores work in conjunction with `WAIT_FOR_RESOURCE`, "amount" parameter. The amount parameter specifies how much of the resource semaphore is required in order for processing to continue. If the amount requested is available, then the processing continues. |

Table 8.4: List of Semaphores

Logical and counter semaphores contain a method called `IsSet`. When processing continues after a `WAIT_FOR` process model construct, users can call the method to determine if the semaphore was set, or if the process model breakout was due to a timeout. If `IsSet()` returns 0, then the semaphore was not set and the timeout was reached.

Let us continue enhancing the car and stop light example to show the process model `WAIT_FOR` construct usage with semaphores and timeouts. Example 8.4 shows the new definition file for the car simulation object. Changes made to the definition file are:

1. Added `#include "SpProcSem.H"` (line 7).
2. Changed method name from `StopCar` to `Controller` (line 15).
3. Deleted the attributes `XPos`, `LastTimeStopped`, `StopState`, `Cancel_Handle`, and `Event_Scheduled_Time`.
4. Added two `SpLogicalSem` semaphores called `Red` and `Green` (line 21 and 22).
5. Changed `DEFINE_SIMOBJ_EVENT_0_ARG(Car_StopCar, S_Car, StopCar)` to `DEFINE_SIMOBJ_EVENT_0_ARG(Car_Controller, S_Car, Controller)` (line 27).

```
1   // S_Car.H
2   #ifndef S_Car_H
3   #define S_Car_H

4   #include "SpSimObj.H"
5   #include "SpDefineSimObj.H"
6   #include "SpDefineEvent.H"
7   #include "SpProcSem.H"

8   class S_Car : public SpSimObj {
9     public:
10      S_Car() {};
11      virtual ~S_Car() {};

12      virtual void Init();
13      void         Stop();
14      void         Go();
```

```
    void         Controller();

  protected:
  private:
    double            Velocity;           // Car Velocity
    RB_int            TimesStopped;       // Number of times Car
                                          //  Stopped
    SpLogicalSem      Red;                // Set when light is red
    SpLogicalSem      Green;              // Set when light is green
};

DEFINE_SIMOBJ(S_Car, 4, SCATTER);
DEFINE_SIMOBJ_EVENT_0_ARG(Car_Stop, S_Car, Stop);
DEFINE_SIMOBJ_EVENT_0_ARG(Car_Go, S_Car, Go);
DEFINE_SIMOBJ_EVENT_0_ARG(Car_Controller, S_Car, Controller);
#endif
```

Example 8.4: Semaphore Car Object Definition File

The code shown in Example 8.5 shows the changes made to the car implementation file. The following list explains the changes made in the implementation file, as compared to the event version described in Section 6.3:

1. Added `#include "SpProc.H"` (line 6).
2. Replaced `PLUG_IN_EVENT(Car_StopCar)` with `PLUG_IN_EVENT(Car_Controller)` (line 11).
3. Deleted the initialization attributes `XPos`, `LastTimeStopped`, `StopState`, `Cancel_Handle`, and `Event_Scheduled_Time`.
4. Cleared semaphore attributes `Red` and `Green` (line 18 and 19).
5. Replaced method `Stop` with code that sets semaphore `Red` (line 22 though 24).
6. Replaced method `Go` with code that sets semaphore `Green` (line 25 though 27).
7. Renamed method `StopCar` to `Controller` (line 28).

```
1  // S_Car.C
2  #include "SpGlobalFunctions.H"
3  #include "SpMainPlugIn.H"
4  #include "RB_ostream.H"
5  #include "SpSchedule.H"
6  #include "SpProc.H"

7  #include "S_Car.H"

8  static int QuarterMile = 5280 / 4; // In feet

9  void PlugInCars() {
10   PLUG_IN_SIMOBJ(S_Car);
11   PLUG_IN_EVENT(Car_Controller);
12   PLUG_IN_EVENT(Car_Go);
13   PLUG_IN_EVENT(Car_Stop);
14 }
```

```
void S_Car::Init() {
  Velocity      = (SpGetSimObjGlobalId() * 10.0 + 11.0) / 3600 * 5280;
  TimesStopped  = 0;
  Red.Unset();
  Green.Unset();

  SCHEDULE_Car_Controller(0.0, SpGetObjHandle());
}

void S_Car::Stop() {
  Red.Set();
}

void S_Car::Go() {
  Green.Set();
}

void S_Car::Controller() {
  P_VAR;

  P_LV(double, XPos);
  P_LV(double, LastTimeStopped);
  P_LV(int,    StopState);
  double       currentXPos;
  int          nextStopLightPos;
  double       stopLightArrivalTime;

  P_BEGIN(2);

  XPos            = 0.0;
  LastTimeStopped = 0.0;
  StopState       = 1;
  for (;;) {
    Red.Unset();
    WAIT_FOR(1, Red, -1.0);
    /*
     * Light turned Red.  Calculate the distance the car is from the
     * stop light and the amount of time that it will take for the
     * car to arrive at the light.
     */
    if (StopState == 0) {
      LastTimeStopped = SpGetTime() - 30.0;
      StopState       = 1;
    }
    currentXPos          =
         (XPos + Velocity * (SpGetTime() - LastTimeStopped));
    nextStopLightPos     =
         (int) ((((int) currentXPos) / QuarterMile + 1) * QuarterMile);
    stopLightArrivalTime =
           (nextStopLightPos - currentXPos) / Velocity;
    Green.Unset();
    WAIT_FOR(2, Green, stopLightArrivalTime);
    /*
     * Either the light has turned Green (Car never made it to the
     * light before the light turned red), or the WAIT_FOR
63      * timed out which indicates we have arrived at the stop light
64      * and the light was red so we must stop.
65      */
66     if (Green.IsSet() == 0) {
67       StopState = 0;
68       ++TimesStopped;
69       XPos = XPos + (Velocity * (SpGetTime() - LastTimeStopped));
70       LastTimeStopped = SpGetTime();
71       RB_cout << "Car #" << SpGetSimObjGlobalId()
72               << " is stopping at " << SpGetTime()
73               << " at X coordinate of " << XPos
74               << ". Car has stopped " << TimesStopped << " times."
75               << " LastTimeStopped= " << LastTimeStopped
76               << endl;
77     }
78     else {
79       /*  Light is green.  Therefore do not stop the car */
80     }
81   }
82   P_END;
83 }
```

Example 8.5: Semaphore Car Object Implementation File

Method `Controller` now contains the entire algorithm for handling the car position and state. Lines 30 though 35 define all of the variables necessary for this method. Notice that the car simulation object state variables `XPos`, `LastTimeStopped`, and `StopState`, which were deleted from the simulation object definition, are now process model local state variables (this was not required, but provided a nice example for local state variables). Line 41 clears the `Red` semaphore, before entering the `WAIT_FOR` construct. On line 42 we enter the process model `WAIT_FOR` construct waiting for semaphore `Red` to be set. Since the timeout defined here is negative, processing will not continue until the semaphore is set.

Once the `Red` semaphore is set, processing continues on line 48. If `StopState` is equal to 0, then the car had been stopped at a red stop light, so variable `LastTimeStopped` is set to the time that the light last turned green. This keeps this example consistent with the previous car and stop light example simulations. If the stop light turned red and the car was not stopped at the stop light, then the distance from the light is calculated along with the time that it will take for the car to arrive at the next stop light (lines 52 through 57). This time is used for the timeout time in the process model `WAIT_FOR` construct. Line 58 clears the `Green` semaphore.

On line 59, the process model `WAIT_FOR` construct is entered. The car will arrive at the stop light at the time calculated previously. When either the `Green` semaphore is set or the timeout value reached, processing will continue on line 66. At this time, a check is done to determine if the semaphore was set (light turned green before the car arrived at the red light), or if the timeout was reached (indicates the car arrived at the red light and needs to stop). If the car arrived at the light and the light was still red, then lines 66 though 77 update the car's position.

Although the output from this example and the car and stop light example described in Section 6.3 are identical, there are interesting differences. The car and stop light simulation modeled in Chapter 6 was implemented using only events. The model code was spread out over three different events, which can add to model complexity. The process model example still contained three events, but all of the logic required to model the car's position and state is contained in one event. The use of the process model can simplify model logic at times.

### 8.1.4 Ask Process Model Reentry Points

The third and last type of process model construct is the ASK. This process model construct allows users to send data, via input parameters, to another simulation object method, regardless of what node the simulation object resides on. The receiving simulation object responds to the sender via return output parameters. The ASK and the response to the ASK are implemented in two separate events.

The process model ASK construct communicates with (i.e. asks) a user-defined event method written on any simulation object. To employ the ASK in a process, the user must first use a macro that turns a method into an event that can be used by the process model ASK construct. The API for this event is very similar to previously discussed event definition macros and is shown below:

```
DEFINE_ASK_EVENT_<numParam>_ARG(askEventName,
                                 className,
                                 methodName,
                                 [paramList])
```

| Parameter | Description |
|---|---|
| numParam | The number of parameters used in the method being converted into an ask event (valid range is 0 to 8). |
| askEventName | Any user-defined string representing the name of the event (legal characters for string names include alphanumeric and underscore characters). |
| className | The name of the simulation object class that contains the method that is to be turned into an ask event. |
| methodName | The name of the method that is to be turned into an ask event. |
| paramList | A comma delimited list of the parameter types found in the method. |

Table 8.5: Macro DEFINE_ASK_EVENT API

The ASK response method (i.e. the method defined with the macro shown above) is invoked by using the process model ASK construct. The API for this macro is as shown:

```
ASK_<numParam>_ARG(reentryLabel,
                   simTime,
                   objHandle,
                   askEventName,
                   [paramList],
                   data,
                   dataBytes)
```

| Parameter | Description |
|---|---|
| numParam | The number of parameters used in the ASK event that is being invoked by this call (valid range is 0 to 8). |
| reentryLabel | Integer id for the appropriate process model reentry label. For example, if P_BEGIN defines the number of reentry labels to be 3, then there should be 3 process model reentry constructs whose labels are 1, 2, and 3. |
| simTime | This parameter specifies the simulation time at which the asked event will be executed. The further this time is ahead of GVT, the less likely that the object that the event is scheduled on will be rolled back. However, the response that is returned by the asked event will be scheduled at current GVT. Therefore, this object will be rolled back (provided there are additional events on this object). This side effect may be eliminated or reduced in future. |

| Parameter | Description |
|---|---|
| objHandle | Object handle of the object for which the asked event is scheduled. |
| askEventName | Any user-defined string representing the name of the event. (legal characters for string names include alphanumeric and underscore characters). |
| paramList | This is a list of parameters, representing the arguments to pass to the ask event method. |
| data | This parameter allows users to send additional data to the asked event for further processing. This data is sent in the form of a char buffer. If this parameter is not needed, set it to `(char *) NULL`. |
| dataBytes | Number of bytes sent in the data parameter (set to 0 if data is `NULL`). |

Table 8.6: Macro ASK API

Let us now show how to use the process model `ASK` construct by adding a Global Positioning System (GPS) object to the car and stop light example. The GPS object will simply define an `ASK` event that will return the input position to the caller. The code shown in Example 8.6 shows the definition file for the GPS object. Specifically, lines 16 and 17 use the `DEFINE_ASK_EVENT` macro to turn method `S_GPS::Position` into an `ASK` event.

```
 1  // S_GPS.H
 2  #ifndef S_GPS_H
 3  #define S_GPS_H

 4  #include "SpSimObj.H"
 5  #include "SpDefineSimObj.H"
 6  #include "SpDefineEvent.H"

 7  class S_GPS : public SpSimObj {
 8    public:
 9      S_GPS() {};
10      virtual ~S_GPS() {};

11      void Position(double inPos, double& outPos);

12    protected:
13    private:
14  };

15  DEFINE_SIMOBJ(S_GPS, 1, SCATTER);
16  DEFINE_ASK_EVENT_2_ARG(Gps_Ask_For_Position, S_GPS, Position,
17                         double, double);
18  #endif
```

Example 8.6: Ask GPS Object Definition File

The code shown in Example 8.7 shows the implementation file for the GPS object. Lines 4 through 7 define the plug-ins necessary for the GPS object. This function must be called from main with the other plug-ins. This is a simple object that returns the input position supplied by the caller (lines 8 through 10).

```
 1  // S_GPS.C
 2  #include "SpMainPlugIn.H"

 3  #include "S_GPS.H"

 4  void PlugInGPS() {
  PLUG_IN_SIMOBJ(S_GPS);
  PLUG_IN_EVENT(Gps_Ask_For_Position);
}

void S_GPS::Position(double inPos, double& outPos) {
  outPos = inPos;
}
```

Example 8.7: Ask GPS Object Implementation File

The final step for this example is to have the car simulation object ask the GPS object for its position. In order to accomplish this, the code shown in Example 8.5 is modified as follows, which will result in the code shown in Example 8.8:

1. Add `#include "S_GPS.H"` below line 7.
2. Add `double askPosition;` below line 35.
3. Change line 36 from `P_BEGIN(2)` to `P_BEGIN(3)`.
4. Add the following below line 70.

```
        askPosition = -1.0;
        ASK_2_ARG(3, SpGetTime(), SpGetObjHandle("S_GPS_MGR",
                                                  "S_GPS_MGR 0"),
                  Gps_Ask_For_Position, XPos, askPosition,
                  (char*) NULL, 0);
```

5. Add `<< ", askPosition= " << askPosition` below line 75.

```
// S_Car.C
#include "SpGlobalFunctions.H"
#include "SpMainPlugIn.H"
#include "RB_ostream.H"
#include "SpSchedule.H"
#include "SpProc.H"

#include "S_Car.H"
#include "S_GPS.H"

static int QuarterMile = 5280 / 4; // In feet

void PlugInCars() {
  PLUG_IN_SIMOBJ(S_Car);
  PLUG_IN_EVENT(Car_Controller);
  PLUG_IN_EVENT(Car_Go);
  PLUG_IN_EVENT(Car_Stop);
}

void S_Car::Init() {
  Velocity        = (SpGetSimObjGlobalId() * 10.0 + 11.0) / 3600 * 5280;
  TimesStopped    = 0;
  Red.Unset();
  Green.Unset();

  SCHEDULE_Car_Controller(0.0, SpGetObjHandle());
```

```
}

void S_Car::Stop() {
  Red.Set();
}

void S_Car::Go() {
  Green.Set();
}

void S_Car::Controller() {
  P_VAR;

  P_LV(double, XPos);
  P_LV(double, LastTimeStopped);
  P_LV(int,    StopState);
  double       currentXPos;
  int          nextStopLightPos;
  double       stopLightArrivalTime;
  double       askPosition;

  P_BEGIN(3);

  XPos            = 0.0;
  LastTimeStopped = 0.0;
  StopState       = 1;
  for (;;) {
    Red.Unset();
    WAIT_FOR(1, Red, -1.0);
    /*
     * Light turned Red.  Calculate the distance the car is from the
     * stop light and the amount of time that it will take for the
     * car to arrive at the light.
     */
    if (StopState == 0) {
      LastTimeStopped = SpGetTime() - 30.0;
      StopState       = 1;
    }
    currentXPos          =
         (XPos + Velocity * (SpGetTime() - LastTimeStopped));
    nextStopLightPos     =
         (int) ((((int) currentXPos) / QuarterMile + 1) * QuarterMile);
    stopLightArrivalTime =
           (nextStopLightPos - currentXPos) / Velocity;
    Green.Unset();
    WAIT_FOR(2, Green, stopLightArrivalTime);
    /*
     * Either the light has turned Green (Car never made it to the
     * light before the car arrived at the light), or the WAIT_FOR
     * timed out which indicates we have arrived at the stop light
     * and the light was red so we must stop.
     */
    if (Green.IsSet() == 0) {
      StopState = 0;
      ++TimesStopped;
      XPos = XPos + (Velocity * (SpGetTime() - LastTimeStopped));
      LastTimeStopped = SpGetTime();
      askPosition = -1.0;
      ASK_2_ARG(3, SpGetTime(),
                SpGetObjHandle("S_GPS_MGR", "S_GPS_MGR 0"),
                Gps_Ask_For_Position, XPos, askPosition,
                (char *) NULL, 0);
      RB_cout << "Car #" << SpGetSimObjGlobalId()
              << " is stopping at " << SpGetTime()
              << " at X coordinate of " << XPos
              << ". Car has stopped " << TimesStopped << " times."
              << " LastTimeStopped= " << LastTimeStopped
              << ", askPosition= " << askPosition
              << endl;
    }
    else {
      /*  Light is green.  Therefore do not stop the car */
    }
  }
  P_END;
}
```

Example 8.8: Ask Car Object Implementation File

When this example is executed, the cars “X coordinate” and “askPosition” will both output the same value.

## 8.2 Tips, Tricks, and Potholes

- Use the process model construct `WAIT`, since they are self-scheduled events that cannot cause rollbacks.
- Wherever possible, use semaphores to interrupt processes, rather than using “polling” or “time stepping” events. If a process cannot determine the next simulated time at which it needs to perform a calculation, have it `WAIT_FOR` a semaphore and let another event set that semaphore at the appropriate time. This alleviates the process from having to periodically “poll” the simulation to check if it needs to act.
- Schedule ask events as far in the future as possible in order to reduce rollbacks (this goes for all types of events as well).
- If asks are used to retrieve part or all of another simulation object’s state, consider the trade-offs between using asks versus using the object proxy system to access the other simulation object’s state. Asks use a “pull” method for accessing another simulation object’s state, while proxies use a “push” method. From a performance perspective, use the following rule to decide between these two strategies: If simulation object A’s data is updated more often than simulation object B accesses it, use asks when the simulation object B needs to access simulation object A’s data. If simulation object B needs to access simulation object A data more often than simulation object A updates that data, use object proxies on simulation object B to access simulation object A’s data.
- Use the process model `WAIT_FOR` construct to implement interruptible waits.
- Multiple `DEFINE` macros can be used on a common simulation object method. Thus, one method can be used as an ask event, a point-to-point event, a handler method, or any combination of these.

- Care must be taken when using `P_LV` to define used variables if the type specified contains pointers. The following code is incorrect because `SpList` contains pointers that point to non-rollbackable data. If this process is reentered, then the variable `list` will not be restored correctly:

  ```
  P_LV(SpList, list);
  ```

  That code could be corrected as follows:

  ```
  P_LV(RB_SpList*, list);
  ...
  list = RB_NEW_RB_SpList();
  ```

  As a second example, the following is correct, since `SpSimTime` has no pointers:

  ```
  P_LV(SpSimTime, time);
  ```

# Part IV

# Object Proxies

# Chapter 9

# Using Object Proxies

## 9.1 Introduction

In the previous chapters, simulation objects and different types of events have been introduced. If data needs to be shared between different simulation objects, then this data has to be sent in an event. While this could be done, this would be a tedious process. SPEEDES has a built-in capability called "Object Proxies" that allows simulation objects to "see" other simulation object's state data. It does this by automatically mirroring the public part of the state of the simulation object. That way, a simulation object holding a proxy can see the attributes of object represented by that proxy and base its processing on that information. For example, a radar simulation object, might have an airplane proxy containing the plane's position. This information could be used by the radar simulation object to change its behavior based on whether or not the airplane is within its radar range.

In the case of a traditional, single CPU simulation, the object proxy mechanism probably would not be particularly important, because all information is locally available. However, by using object proxies on a multiple CPU simulation, simulation objects gain read access to other simulation object's proxies, regardless of simulation object CPU (i.e. node) location. The proxy mechanism makes this transparent to the requesting simulation object. From the simulation object perspective, it is just as if the simulation is running on one node and the simulation object is reading global data from another simulation object, even though this object's physical location is on a different CPU (or computer, for that matter). Without the proxy, an object would have to:

1. Learn of the existence of a certain type of simulation object.
2. Determine node location of simulation object.
3. Query the node containing the simulation object asking for the current state of the desired simulation object.
4. Wait for the reply message to arrive.

Object proxies eliminate this process by automatically "pushing" updates to all proxy holders when the public state of the object is changed (and only when there are changes). Of course, "pushing" updates, rather than having simulation objects "pull" them as needed, involves a trade-off, since updates could be sent to simulation objects that do not actually need them. However, since proxies are only pushed to simulation objects that have previously expressed an interest for its data (i.e subscription), then this trade-off should be favorable. Also, the way SPEEDES implements proxies minimizes message traffic generated by sending proxy updates. Should this prove inadequate, SPEEDES provides additional filtering through its DDM services (see Chapter 11). Finally, the simplification to the code associated with proxies is in itself a big advantage when compared to the code users would need to write when not using the built-in proxy services.

## 9.2 Essential SPEEDES Object Proxy Terminology

**High Level Architecture (HLA) Simulation Object**: An HLA simulation object is defined to be a simulation object that inherits from class `S_SpHLA`. This class inherits from class `SpSimObj`. Therefore, it has all of the functionality previously discussed. In addition, new functionality has beed added which allows users read access to other HLA simulation object state variables.

**Attribute**: The elements of a simulation object's state that are to be made public for subscribers are called its attributes. For example, a ship simulation object could have a name, id number, position, course, and speed attributes (among others). Note that a simulation object's attributes are just the part of the object that it wishes to advertise, and not necessarily the entire state of the simulation object. In a typical case, the implementation of a simulation object class will have many internal data members that are not publicly announced, hence not part of its proxy.

**Publish**: When a simulation object publishes itself, it announces that it is making part of its state publicly readable to other simulation objects that are interested in this information.

**Subscribe**: This is the complement of publish, except done with respect to object classes. That is, when a simulation object subscribes to a class, it is asking SPEEDES to inform it of the existence of all such simulation objects instances, and to provide updates on the state of these simulation objects.

**Discover**: When a simulation object is created, all subscribers to that class are informed of the existence of the new simulation object. This is called discovering a simulation object. In SPEEDES, the discovery of a simulation object is implicit in the delivery of a proxy to the subscriber. The delivered proxy corresponds to the newly discovered simulation object.

**Reflect**: When a simulation object modifies one or more of its attributes, then subscribers to that class are informed of the new attribute values.

**Declaration Management (DM)**: This term refers to the basic proxy algorithm that matches publishers and subscribers. Proxies representing a given class are delivered to all subscribers of that class (or a superclass). Subscribers receive all attribute updates.

**Data Distribution Management (DDM)**: This term refers to a more advanced mechanism than that of DM above, in that DDM filters out attribute updates that are not needed or requested. For example, a radar object would not need to know about the current position of a plane located 8000 miles away and the use of DDM would allow automatic filtering of these out of range objects (see Chapter 11 for additional detail on DDM).

## 9.3 Object Proxy Usage Overview

The purpose of this section is to outline the procedure required to use proxies in SPEEDES. This will provide a context within which to understand the more detailed instructions in later sections. The following four steps describe the essence of proxy usage:

1. Define the simulation object's public attributes in the required file called `Objects.par`. Typically, each class is specified by a name and a list of its attribute names, along with a type designation for each attribute. Also, classes may specify desired class subscriptions at this point. The advantage of this is that the subscriptions can be changed via a text file (no-recompilation of source code).

2. Write classes corresponding to the simulation object classes defined in `Objects.par`. This means that the classes contain the attributes specified in `Objects.par`. SPEEDES provides special attribute classes for this purpose. Also, these simulation objects must inherit from a built-in SPEEDES class called `S_SpHLA` (hence the name "HLA" simulation object), which allows the proxy system to function.

3. Publish the simulation object instances. This occurs automatically if step 2 has been done correctly. To accomplish this, the simulation object's constructor must "declare" its name (i.e. this name must be identical to that found in `Objects.par`) and "define" its attributes. These two operations hook the simulation object instance and its associated attributes into the proxy system.

   When an HLA simulation object is constructed, SPEEDES sends a proxy to all subscribers of the simulation object's class. Also, when attributes are altered, SPEEDES keeps track of the changes and sends update messages to proxy holders so that the proxies mirrors the new attribute values. This automatic update mechanism is performed by the special attribute classes mentioned in step 2 above.

4. Subscribe to simulation object classes. SPEEDES will do this automatically if the subscription classes have been listed in the object class definition in `Objects.par` (step 1 above). For example, if the `Objects.par` had defined classes called Ship, Submarine, and Airplane and the Ship was subscribing to Submarines, then each Ship simulation object instance would receive a proxy of each Submarine. Likewise, since the Ship did not subscribe to the Airplane class, then it will not receive any Airplane proxies. SPEEDES provides mechanisms for examining the attributes contained in the proxies. Using these mechanisms, a subscriber can "see" the state of remote objects, and base its actions on that information.

   Also, there are methods available for manually subscribing and unsubscribing to object classes. This allows greater control over subscriptions than the class-wide method described above. That is, individual objects can decide whether to subscribe to a class and for how long.

## 9.4 Object Proxy Usage Detailed Description

The following sections elaborate on the four basic steps outlined above. The intent is to provide enough detail to begin using object proxies right away. Although there is some overlap, the first several sections deal primarily with the publisher's side of things. That is, the process of defining public object classes and creating instances that subscribers can see. The later section discuss the details of how a subscriber discovers objects, maintains lists of proxies, and how the proxies themselves can be queried for information.

### 9.4.1 Defining Object Classes in File Objects.par

SPEEDES reads in the `Objects.par.` file during simulation initialization. This allows SPEEDES to know how to build object proxies matching the objects that will be published and subscribed to in the simulation. Therefore, simulation object publications and subscriptions can only be made for classes defined in `Objects.par`. Figure 9.1 below shows a example `Objects.par` file containing the definition of five simulation object classes named `Entity`, `FixedEntity`, `GroundRadar`, `Airplane`, and `Airport`. The example shows several different types of features and attribute types available for use when designing the `Objects.par` file.

```
// The characters "//" indicate the start of a comment
// Base class for all objects
Entity {
  define int     EntityID         // Proxies can contain integers
  define string  EntityName       // Proxies can contain strings
  define logical Alive            // Proxies can contain booleans
}

// Base class for immobile objects
FixedEntity {
  reference INHERIT  Entity       // Inherits attributes from Entity
  define    position ObjLocation  // Fixed position declaration
}

// Stationary radar class
GroundRadar {
  reference INHERIT   FixedEntity // GroundRadar is a fixed entity
  reference SUBSCRIBE Airplane    // GroundRadar "subscribes" to
                                  //  Airplanes
  define    double    ScanTime    // Proxies can contain doubles
}

// Generic airplane class
Airplane {
  reference INHERIT          Entity   // Airplane is an entity
  define    dynamic_position Position // Dynamic position declaration
}

Airport {
  reference INHERIT FixedEntity       // Airport is a fixed entity
  define    object  MyRadar           // Airport has a radar
  define    list    ListOfPlanes      // Airport maintains a list
                                      //  of airplanes
  define    int     HourlyFlights[24] // Array of 24 integers
}
```

Figure 9.1: Objects.par Example File

The first class definition described is for `Entity`. The syntax is similar to that of a C++ class. It consists of a class name followed by the class description enclosed in braces. The `Entity` class description contains a list of three attribute declarations. This class definition tells SPEEDES that an `Entity` proxy consists of an `int` attribute called `EntityID`, a `string` attribute called `EntityName`, and a `logical` attribute called `Alive` (the `logical` type is like the C++ `bool` type and the `string` type is similar to a `char` array). SPEEDES provides many attribute types to choose from when designing the `Objects.par` file and HLA simulation objects (see Chapter 10). Note that C++ style comments are allowed (i.e. all text after the double slash is ignored).

So far, all that has been done is to define a class called `Entity` which has three attributes called `EntityID` (an integer), `EntityName` (a string), and `Alive` (a boolean). Class `FixedEntity` illustrates how a class can inherit from another class and add addition attributes to its definition. To have an HLA class inherit from another HLA class, add the keywords `reference INHERIT` to that class definition, as shown below:

```
reference INHERIT className
```

The `className` specified must match exactly (i.e. a case-sensitive string comparison) the name of a class previously defined in the `Objects.par` file. For this example, this means that `FixedEntity` contains the three attributes specified in `Entity`. The next line in the file adds an additional attribute called `ObjLocation` to `FixedEntity`. The `position` type is a SPEEDES attribute type denoting a fixed location (as opposed to a dynamic position attribute, which describes the location of a moving object over time). A position attribute can be specified in a variety of coordinate systems (more about this in Section 10.2.4).

Next, examine the `GroundRadar` declaration. This object inherits from `FixedEntity`. Therefore, it contains all the attributes that a `FixedEntity` does. The next line illustrates how class definitions can statically subscribe to other objects by using the following construct:

```
reference SUBSCRIBE className
```

In this case, `reference SUBSCRIBE` indicates that all `GroundRadar` simulation objects are subscribed to `Airplane` simulation objects. Therefore, a proxy for each `Airplane` instance will be delivered to each `GroundRadar` simulation object. Notice, that the `Airplane` class definition has not been defined yet. Definition of the class objects prior to subscription is not a requirement when creating an `Objects.par` file. Once again, the `className` specified must match, exactly, the name of a class defined somewhere in `Objects.par` file. Finally, the `GroundRadar` declaration adds another attribute, `ScanTime`, which is a double precision floating point number.

The next object definition `Airplane`, inherits from `Entity`. `Airplane` adds one attribute called `Position`, which is of type `dynamic_position`. Attributes of this type contain a function, $f(t)$, that returns that position (e.g. latitude, longitude, altitude) at a given time (i.e. position = $f(\text{time})$). How this function is specified will be discussed in greater detail in Section 10.2.4.

The idea of a dynamic position is that the subscribing object can use this attribute to figure the current location of an `Airplane` by itself, and therefore, does not need to get a series of location updates. This works as long as the `Airplane` sticks to its scripted motion plan. If it deviates from the plan, it sends out an update of the `Position` attribute so that subscribers obtain a new motion function, $g(t)$, that reflects the new motion plan. There are many built-in dynamic attribute types available in SPEEDES, such as the `DYNAMIC_DOUBLE_ATTRIBUTE`, which specifies a double precision number that varies over time, and the `DYNAMIC_INT_ATTRIBUTE`, which does likewise for an integer. Of course, you can design your own dynamic types if the built-in SPEEDES types are inadequate.

The final object declaration is the `Airport`, which inherits from `FixedEntity`. There are a few things to note, beginning with the first attribute declaration, `define object MyRadar`. The `object` type tells SPEEDES that the attribute will correspond to one of the classes defined in `Objects.par`. In this case, the attribute will represent a `GroundRadar`, hence the name `MyRadar`. This is all the information required by SPEEDES to set up the object proxies. Therefore, it is unnecessary to specify the exact object type. The `object` type attribute is essentially a recursive definition saying that this attribute will itself be represented by a proxy. In the simulation object corresponding to `Airport`, there will be a data member that inherits from `OBJECT_ATTRIBUTE`, one of the attribute types provided by SPEEDES. A description on how to set this up is discussed later.

The next attribute, `ListOfPlanes`, is of type `list`. This type allows the user to accumulate an ordered collection of objects. That is, one can add and remove items to create sequences of arbitrary length, but each `list` item must inherit from `OBJECT_ATTRIBUTE`. In this case, the intent is to keep track of all of the `Airplane` objects at the airport, hence it is called `ListOfAirplanes`. Section 10.1.5 will describe in greater detail how to use this attribute.

The final attribute declaration, `define int HourlyFlights[24]`, is shown as an example of how to declare an array attribute. `HourlyFlights` is an array of 24 integers (indexed 0 to 23), that keeps track of the number of flights departing the airport during each hour of the day. Any of the attribute types can be made into an array simply by appending `[N]` onto the end of the attribute name (where `N` is a positive integer). For example, the following line defines `Vec` to be an array of 100 doubles:

```
define double Vec[100]
```

As another example to define an array of twelve strings, write:

```
define string Months[12]
```

The following rules summarize the syntax of the `Objects.par` file. Although a somewhat informal notation is used, this should be sufficient to specify the format without introducing significant ambiguity. The basic notation rules are:

- The only meta-character is the asterisk ("*") which means "zero or more occurrences of the immediately preceding item".
- All text in Courier font (e.g. `define`) is to be used literally.
- Items in italics represent text patterns that are described in one of the rules, and should be replaced by a suitable instance of the pattern.

`Objects.par` file rules:

1. An `Objects.par` file consists of a list of zero or more *ObjectClassDefinitions*:

   *ObjectClassDefinition**

2. *ObjectClassDefinition* defines an object class, and is in the following form:

   *ObjectClassName* {
      *ClassSpecification*
   }

3. *ObjectClassName* is a case-sensitive string consisting of one or more allowable characters, beginning with a letter (not a numeral or an underscore). Allowable characters include all the letters (upper and lower case), the numerals 0 through 9, and the underscore character, "_".

4. *ClassSpecification* consists of a series of the following lines, in the order shown, where *InheritanceLine* is optional (it is only used if the class inherits from another class in `Objects.par`):

   *InheritanceLine*
   *SubscriptionLine**
   *AttributeSpecificationLine**

(a) *InheritanceLine* is a line in the following form:
`reference INHERIT` *ObjectClassName*

(b) *SubscriptionLine* is a line in the following form:
`reference SUBSCRIBE` *ObjectClassName*

(c) *AttributeSpecificationLine* is a line in one of the following two forms (where *ArraySize* is an integer greater than 0):
`define` *AttributeType AttributeName*
`define` *AttributeType AttributeName*`[`*ArraySize*`]`
Where *AttributeType* is one of the following: `int`, `logical`, `double`, `string`, `object`, `list`, `binary_buffer`, `position`, `dynamic_int`, `dynamic_logical`, `dynamic_double`, or `dynamic_position`.

### 9.4.2 Implementing Object Definitions Contained in Objects.par

After the object classes have been defined in file `Objects.par`, then the appropriate SPEEDES constructs that match these object classes have to be implemented. The simulation classes "match" in the sense that they contain all the attributes defined in the `Objects.par` and are of the correct type. The easiest way to describe this further is through the use of examples.

Consider the definition for class `Entity` shown in Figure 9.1 and repeated below for convenience.

```
// Base class for all objects
Entity {
  define int     EntityID         // Proxies can contain integers
  define string  EntityName       // Proxies can contain strings
  define logical Alive            // Proxies can contain booleans
}
```

The code shown in Example 9.1 shows the implementation for this simulation object.

```
// S_Entity.H
#ifndef S_Entity_H
#define S_Entity_H

#include "S_SpHLA.H"

class S_Entity : public S_SpHLA {
  public:
    S_Entity(char* objClassName = "Entity") :
      S_SpHLA(objClassName) {}

    virtual ~S_Entity() {}

    void Init() {
      DEFINE_ATTRIBUTE(EntityID,   "EntityID");
      DEFINE_ATTRIBUTE(EntityName, "EntityName");
      DEFINE_ATTRIBUTE(Alive,      "Alive");
      SetEntityID(-1);
      SetEntityName(NULL);
      SetAliveStatus(1);
    }
    void SetEntityID(int id) {EntityID = id;}
    int  GetEntityID()       {return EntityID;}

    void        SetEntityName(char* n) {EntityName = n;}
    const char* GetEntityName()        {return EntityName;}

    void SetAliveStatus(int objIsAlive) {
      if (objIsAlive != 0) {
        Alive = LOGICAL_TRUE;
      }
      else {
        Alive = LOGICAL_FALSE;
      }
    }

    int ObjIsAlive() {
      return (Alive == LOGICAL_TRUE);
    }

  private:
    INT_ATTRIBUTE     EntityID;   // Unique object id
    STRING_ATTRIBUTE  EntityName; // Object name
    LOGICAL_ATTRIBUTE Alive;      // Object Status
};
#endif
```

Example 9.1: Entity Class Definition

This example has several items of interest, including:

1. The class inherits from `S_SpHLA`. This enables simulation object instances to participate in the proxy system. Objects that inherit from `SpSimObj` do not participate in the proxy system.

2. The name of the class (i.e. `S_Entity`) does not need to be the same as the class name in the `Objects.par` file. The `S_Entity` constructor takes a `char *` argument, which it passes on to the `S_SpHLA` constructor. The name of this argument has to be defined in a class listed in `Objects.par`. The `S_SpHLA` constructor publishes the simulation object instance allowing subscribers to discover it. This concept should be followed through in derived classes such as `S_FixedEntity`. That is, the class constructor should take a `char *` argument and call the constructor of the parent class. This will result in code similar to what is shown in Example 9.2.

   Notice that the constructor for `S_Entity` has a default argument (i.e. `objClassName = "Entity"`). This is necessary for the `DEFINE_SIMOBJ` macros to work. The simulation define macros produce code that creates objects using zero-argument constructors. Therefore, a default argument is essential for using HLA objects with the `DEFINE_SIMOBJ` macros.

3. The class contains attributes that correspond to the attributes listed in the `Objects.par` definition. Specifically, the data member `INT_ATTRIBUTE EntityID` corresponds to `EntityID`, data member `STRING_ATTRIBUTE EntityName` corresponds to `EntityName`, and `LOGICAL_ATTRIBUTE Alive` corresponds to `Alive`. As with the class name, the data members do not need to have the same name as the attributes listed in `Objects.par`.

   The types `INT_ATTRIBUTE`, `STRING_ATTRIBUTE`, and `LOGICAL_ATTRIBUTE` are three of the built-in attribute classes provided by SPEEDES. Some of these attribute classes have been designed to mimic C++ types. For example, an `INT_ATTRIBUTE x` can be used in expressions just like an ordinary integer (e.g. `int x = y * 3 + 27`). The reason they must be used is that they automate the sending of updates to subscribers. Any change to `INT_ATTRIBUTE x (e.g. x = y + 14, x--, ++x,` etc.) will cause an update message to be sent to all subscribers, altering their proxies to reflect the new value assigned to `x`. The way this works is by overloading the assignment operators so that they cause updates to be distributed (in addition to performing a normal assignment). All of the built-in proxy attributes are shown and described in Chapter 10.

4. The `Init` method calls macro `DEFINE_ATTRIBUTE` for each attribute definition. There must be one `DEFINE_ATTRIBUTE` call for each attribute in the class. This macro's first argument is the data member and its second argument is the corresponding attribute name from the `Objects.par`. Once again, the name used in the second argument must be identical to the name found in the `Objects.par` file. This macro call is needed to associate the data member with the attribute name, and to hook it into the proxy system.

   When the subscriber receives the proxy representing an instance of `S_Entity`, the current attribute values are extracted using the attribute names given to `DEFINE_ATTRIBUTE`. The proxy has a method called `Find` that returns a pointer to the attribute with the given name:

```
INT_ATTRIBUTE* EntityId = (INT_ATTRIBUTE *)
  EntityProxy.Find("EntityID");
```

```
// S_FixedEntity.H
#ifndef S_FixedEntity_H
#define S_FixedEntity_H

#include S_Entity.H

class S_FixedEntity : public S_Entity {
  public:
    S_FixedEntity(char* objClassName = "FixedEntity") :
      S_Entity(objClassName) {}
    virtual ~S_FixedEntity() {}

    void Init() {
        DEFINE_ATTRIBUTE(ObjLocation, "ObjLocation");
    }

  private:
    POSITION_ATTRIBUTE_TYPE ObjLocation;
}
#endif
```

Example 9.2: Fixed Entity Class Definition

### 9.4.3 Publishing Objects

HLA objects, by default, are self-publishing. Users need not do anything out of the ordinary to have simulation objects publish themselves for properly implemented HLA objects.

The three basis steps to follow when implementing HLA simulation objects are:

1. Define the class and its public attributes in the `Objects.par` file.

2. Implement a class that inherits from class `S_SpHLA`. Its constructor should pass the class name defined in `Objects.par` to the `S_SpHLA` constructor. This announces the existence of the instance to the proxy system. If the macro `DEFINE_SIMOBJ` is called, make sure that the class contains a default constructor (i.e., a constructor which can be called without passing arguments).

3. Use the SPEEDES-provided attribute classes when implementing the HLA simulation objects. These classes are rollbackable and automate attribute updates. In the simulation object's `Init` method, use the `DEFINE_ATTRIBUTE` macro on each attribute. All of the built-in attribute types available in the SPEEDES framework are described in Chapter 10.

### 9.4.4 Subscribing to Object Classes

HLA simulation objects can "subscribe" to other HLA simulation objects. Any subscriptions to an HLA simulation object causes SPEEDES to give all of the published attributes to the subscriber. There are two methods which HLA simulation objects can use to make subscriptions

The easiest way to subscribe to a class is to add `reference SUBSCRIBE` to the appropriate class(es) definition in `Objects.par`. The example shown below shows the `GroundRadar` subscribing to all `Airplane` simulation objects. Using a `SUBSCRIBE` line means that every instance of a `GroundRadar` object will automatically subscribe to the `Airplane` class.

```
Airplane {
  define double Speed
  define int    IntVec[10]
}

GroundRadar {
  reference SUBSCRIBE Airplane // <-- SUBSCRIBE line
  define    double    ScanTime
}
```

Setting the subscriptions via `Objects.par` allows users to configure the inter-simulation object subscription requirements at simulation initialization. Subscriptions can be created or removed during run time by using the following methods found on class `S_SpHLA`.

```
int Subscribe(char*   objClass)
int UnSubscribe(char* objClass)
```

Therefore, by using these methods at the appropriate time, a user can control "when" a simulation object starts or stops subscribing to another class of objects.

Let us look at the details of finding a proxy and examining it. The subscriber class must also inherit from `S_SpHLA`, just like the publisher. From the `S_SpHLA` class, the subscriber gains access to method `GetRemoteObjectProxies`. This method, which takes no arguments, returns a pointer to an `RB_queue`, which is a linked list (see the "SPEEDES API Reference Manual" for additional details regarding `RB_queue`s). The items in the `RB_queue` are pointers to `F_SpProxyItem`, a container holding a proxy pointer (`SpObjProxy *`). The following example illustrates how to find a proxy on the proxy list.

```
// Return proxy of Airplane whose global id is given. If
// not found, return NULL.

SpObjProxy* S_GroundRadar::GetPlaneProxy(int globalId) {
  int i;
  // Get list of delivered Airplane proxies
  RB_queue* planeProxies = GetRemoteObjectProxies();

  int numProxies = planeProxies->get_length(); // Number of proxies
                                               //  in list
  // Search through list for the proxy with right global id
  F_SpProxyItem* pItem = (F_SpProxyItem *) planeProxies->get_top();
  for(i = 0; i < numProxies; ++i) {
    SpObjProxy* planeProxy = pItem->GetObjProxy();
    if (planeProxy->GetProxySimObjGlobalId() == globalId) {
      return planeProxy;
    }
    pItem = (F_SpProxyItem *) pItem->get_link();
  }
  return NULL; // not found
}
```

The above example shows the proper way of accessing each proxy on the simulation object remote proxy list, as well as looping through the proxy list. The primary points of interest are:

- Use the method `get_top` to access the first item in the list, and `get_length` to find the number of items in the list.
- The proxy pointer is retrieved from each `F_SpProxyItem *` item using the `GetObjProxy` method.
- The next item in the `RB_queue` is retrieved by calling `get_link` on the current item.
- **Caution:** The remote proxy list is an `RB_queue`, which is not necessarily `NULL`-terminated. To search the list, use methods `get_top` and `get_length` in a loop as shown above, counting the number of list items visited.

After the subscriber has found a given proxy in the list returned as `GetRemoteObjectProxies`, the proxy may be examined. To locate an attribute by name, use the following method located on class `SpObjProxy`.

```
BASE_ATTRIBUTE* Find(const char* name, int arrayIndex = 0)
```

The parameter `name` specifies the string name of the attribute being searched for. If the data being searched for is an array, then an index for an array element can be supplied. The following short example shows how to retrieve both non-array and array data values from the proxy.

```
GroundRadar* radar = GetRadar();

SpObjProxy*  planeProxy = radar->GetPlaneProxy();

DOUBLE_ATTTRIBUTE* planeSpeed =
  (DOUBLE_ATTTRIBUTE *) planeProxy->Find("Speed");

// Get the 7th item from IntVec[10] attribute.
// Indexing is 0..9
INT_ATTTRIBUTE* planeSpeed =
  (INT_ATTTRIBUTE *) planeProxy->Find("IntVec", 6);
```

Both of the above versions of `Find` search an array of attributes, and locate the one with the correct name using string compares. This inefficient method should be replaced by the use of `GetReference` to obtain the attribute's reference (i.e. its integer index) into the proxy array. This will allow faster look ups. The `Find` method has been overloaded to work with the attribute reference (for both scalar and array attributes).

```
  GroundRadar* radar = GetRadar();

  SpObjProxy*  planeProxy = radar->GetPlaneProxy();

  // Get reference to "Speed" attribute
  int speedRef = planeProxy->GetReference("Speed");

  // Find attribute by reference:
  DOUBLE_ATTTRIBUTE* planeSpeed =
    (DOUBLE_ATTTRIBUTE*) planeProxy->Find(speedRef);

  // ***** This also works with array attributes *****

  // Get reference to "IntVec" attribute
  int intVecRef = planeProxy->GetReference("IntVec");

  // Get the 7th item from IntVec[10] attribute.
  // Find attribute by reference:
  INT_ATTTRIBUTE* planeSpeed =
    (INT_ATTTRIBUTE*) planeProxy->Find(intVecRef, 6);
```

You may be wondering why we did not just save a pointer to the attribute and use it for even faster access to the attributes. The problem with this approach is that the pointer could become invalid as the simulation progresses (even though the reference remains valid). When the proxy is updated, the attribute storage is replaced, not just overwritten, so a saved attribute pointer might be pointing to deleted memory. Therefore, for faster access, use `GetReference`, not saved attribute pointers.

As shown above, the `Find` method is the most general way to retrieve the attributes contained within a proxy. In addition to `Find`, there are several type-specific methods for accessing attributes. For example, the `GetInt` method can be used to get the current value of an integer attribute, and like `Find`, it can locate attributes by name or by reference. Unlike `Find`, it returns the current value of the attribute, rather than a pointer to the attribute. The `SpObjProxy` class provides type-specific accessors for all the attribute types (e.g. `GetString`, `GetList`, `GetLogical`, etc.). See the "SPEEDES API Reference Manual" for addition information on these interfaces.

#### 9.4.4.1 Associating Proxies With Objects

In the previous example, the simulation object global ids were used to locate the proxy corresponding to a particular object. Proxies contain other important data that can be used to identify the object it represents. The following `SpObjProxy` methods are available for this purpose.

- `int GetProxySimObjGlobalId()`
  Returns the global id of the object represented by this proxy.
- `int GetProxySimObjLocalId()`
  Returns the local id of the object represented by this proxy.
- `int GetProxySimObjMgrId()`
  Returns the type id of the object represented by this proxy.
- `int GetProxyNode()`
  Returns the node number on which the object represented by this proxy is located.
- `char* GetProxyName()`
  Returns the name of the object represented by this proxy. This corresponds to the name assigned to the object by the method `SetName` on `S_SpHLA`. Note that the object must be named only once, when it is created in method `Init`.

### 9.4.5 Attribute Level Subscription

In the previous section, we explained how to subscribe to object classes using the DM system. Using this method, the subscriber discovers all object instances whose classes are (or inherit from) the subscribed class. Also, the subscriber receives an update event whenever any of the attributes in a discovered object are altered. That is, the subscriber has subscribed to all attributes published by the object.

Clearly, this is not always desirable. A class might, for example, publish twenty attributes, but a given subscriber might be interested in only five of these. It would be advantageous if that object could subscribe to only those five attributes. This would mean that the subscriber would receive update events only when one of these five attributes changed, and would not be interrupted by extraneous updates. The importance of this capability becomes apparent when one realizes that these extraneous update events could occur in the subscriber's past, thus inducing a rollback of all processing done by events whose time stamp is greater than the update, and that have already acted on the subscriber.

For this reason, SPEEDES provides the ability to subscribe to attributes on an individual basis. This means that a simulation object can subscribe to any subset of the attributes published by a given class X (including the empty subset of zero attributes, which results in objects being discovered, but not updated). This subset is called the attribute subscription for class X. It acts as a filter that removes all update events, except for those containing attributes in the subset. This filter can be dynamically changed throughout the course of the simulation.

In order to use attribute level subscription, an object must subscribe to a published object class (or to a DDM space; see Chapter 11). The procedure for doing this was detailed in section 9.4.4. When this has been done, the attribute subscription for the class (i.e. the filtering subset) can be specified. This is accomplished using the `SubscribeByAttribute` method of the `S_SpHLA` class. This method specifies the current attribute subscription for a given class. Its interface is as follows:

```
void SubscribeByAttribute(char*  objClass,
                          int    numAtts,
                          char** attNames)
```

| Parameter | Description |
|---|---|
| objClass | Specifies the name of the class for which the attribute subscription is desired. This is the same class name used in `Objects.par`. |
| numAtts | Specifies the number of attributes in the subscription filter. |
| attNames | Specifies an array of attribute names to subscribe to. there must be `numAtts` of these names, each of which should exactly match an attribute name for the class in `Objects.par`. |

Table 9.1: Subscribe By Attribute API

`SubscribeByAttribute` creates a new attribute subscription, and replaces the current one, if any. Let us consider an example. Assume the `Objects.par` file contained the following class declaration:

```
class Blob {
  define int    K1
  define int    K2
  define int    K3
  define int    K4

  define double X1
  define double X2
  define double X3

  define string S1
  define string S2

  define list   L1
}
```

In order to subscribe to `Blob` class attributes `K1`, `X2`, `S2`, and `L1`, the following code could be used (in the `Init` method, for example):

```
// Must subscribe to class Blob (if not already done):
Subscribe("Blob");

const int numAtts = 4;
char*     attNames[numAtts];

attNames[0] = "K1";
attNames[1] = "X2";
attNames[2] = "S2";
attNames[3] = "L1";

// Set current filter to subset { K1, X2, S2, L1 }. This object will
// only receive updates when K1, X2, S2 or L1 is changed.
SubscribeByAtttribute("Blob", numAtts, attNames);
```

Other methods provided by `S_SpHLA` for altering attribute subscriptions are:

```
void SubscribeAllAttributes(char* objClass)

void UnSubscribeAllAttributes(char* objClass)
void AddAttributeSubscription(char*  objClass,
                              int    numAtts,
                              char** attNames)

void RemoveAttributeSubscription(char*  objClass,
                                 int    numAtts,
                                 char** attNames)

int CurrentlySubscribedToAtt(char* objClass,
                             char* attName)
```

The parameter definitions for each method are the same as those specified in Table 9.1. A description for each method is:

- `SubscribeAllAttributes`:
  Changes the current subscription to contain all attributes for the class denoted by parameter `objClass`. This is the default behavior when one does a DM subscribe and there does not yet exist a subscription set. (Which, incidentally, means that it is slightly more efficient to do `SubscribeByAtttribute` first, and then do the `Subscribe`, assuming both calls occur within the same event. This circumvents the need for `Subscribe` to create a full set only to have it immediately overridden by `SubscribeByAtttribute`.)

- `UnSubscribeAllAttributes`:
  Changes the current subscription to the empty set. This means that instances of class `objClass` will be discovered, but no updates will be received (assuming the object has subscribed to `objClass`).

- `AddAttributeSubscription`:
  Inserts additional attributes into the current subscription. The parameters are exactly like those from `SubscribeByAtttribute`, except that `numAtts` and `attNames` specify a set of attributes to be added to the current subscription (instead of replacing the current subscription). The resulting subscription set will be the union of the existing set and the set of additional attributes specified by the call. It is not an error to add attributes that are already in the subscription set; however, if there is any question about what the current subscription is, it is probably easiest to simply use `SubscribeByAttribute`, which defines the entire set, rather than making changes relative to the existing one.

- `RemoveAttributeSubscription`:
  This method is the opposite of `AddAttributeSubscription`. It removes all attributes specified in the call from the current subscription set. It is not an error to remove an attribute that is not a member of the current subscription set; such cases are simply ignored. If, however, an attempt is made to remove attributes before a subscription has been created, a warning is printed.

- CurrentlySubscribedToAtt:
  Returns 1 if if the specified `objClass` and `attName` currently exist in the current subscription set, otherwise it returns 0. As usual the parameters must exactly match (i.e. case-sensitive) a class name and an attribute name, respectively, from the `Objects.par` file.

#### 9.4.5.1 Effects of Attribute Level Subscription

As was mentioned above, the purpose of attribute level subscription is to prevent superfluous update events from reaching the subscriber. To complete the explanation, there is a feature and two issues related to updating that need to be described.

First of all, the `SpObjProxy` class (and descendants) contain an “accessor guard” feature that prevents a subscriber from accessing attributes that are not currently in the attribute subscription set. The reason for this is that such attributes could very well be out of date, meaning that the data they contain might not correctly reflect the current state of the object they represent. This makes perfect sense, by not subscribing to an attribute, the object is asking not to receive updates for that attribute. Thus, the attribute’s value could be an outdated one.

There are two aspects to this guard feature. First, an attempt to reference the attribute will return an “error” value. For example, if an object is not subscribed to `STRING_ATTRIBUTE ObjName`, then calling the `SpObjProxy` method `GetString` returns `NULL`:

```
ShipProxyType *shipProxy = GetShipProxy(); // Grab proxy

// shipName will be NULL if not subscribed to ObjName:
const char *shipName = shipProxy->GetString("ObjName");
```

The above example illustrates the problem with this technique: how can you tell whether the result is an “error” value or the actual attribute value? It might, for example, be perfectly legitimate for an object to have a `NULL` name.

In general, an object should not be accessing attributes to which it is not currently subscribed. If it does, then needs to be able to distinguish between an incorrect attribute data (i.e. an error) and an actual attribute value. Method `LastAttribRefWasLegal` on classes `SpObjProxy` provides this capability. This method is the second aspect of the proxy guard feature, and is shown below:

```
ShipProxyType *shipProxy = GetShipProxy(); // Grab proxy

// shipName will be NULL if not subscribed to ObjName:
const char *shipName = shipProxy->GetString("ObjName");

// Was the GetString call OK?
if (!shipProxy->LastAttribRefWasLegal()) {
  RB_cout << "Cannot access ship name now" << endl;
}
```

As you might suspect, method `LastAttribRefWasLegal` refers only to the very last attribute reference, so it should be called right away, and certainly before making any function calls that might be hiding proxy attribute references.

In order to alert the user, SPEEDES returns an error value of -1e20 if an attribute is accessed which is not subscribed to. Nevertheless, it is probably best to keep track of attribute subscriptions and take care not to access unsubscribed attributes; or else use `LastAttribRefWasLegal` to make sure your reference was a legal one.

The following list provides the user with what to expect whenever an access is made to an unsubscribed attribute:

- **Pointers**: Any proxy accessor returning a pointer will return `NULL`. The most important of these are `GetAttribute` and `Find`. Others examples are: `GetBinaryBuffer`, `GetList`, `GetListName`, `GetDynamicObject`, `GetObjProxy`, and `GetString`.

- **Doubles**: Any proxy accessor that returns one or more doubles will return a double (or an array of doubles) whose value is -1e20. Examples include `GetFloat`, `GetPosition` (which returns an array of three doubles), `GetDynamicPosition`, `GetDynamicFloat`, etc.

- **Integers**: Any proxy accessor that returns an integer will return the hexadecimal number `0x8000000` (which equals 134,217,728). Examples include `GetInt` and `GetDynamicInt`.

- **Booleans**: Any proxy accessor that returns a boolean will return 0 whenever an unsubscribed referenced is made. Examples include `GetLogical` and `GetDynamicLogical`.

Finally, we wish to direct your attention to two subtle points of the attribute level subscription system. The system was designed to streamline the distribution of information by minimizing unnecessary update events on subscriber objects. But because it is not practical to entirely eliminate such events, it is possible that a small number of them will get through.

Another consequence of this streamlining is the opposite: it is possible for a subscriber not to receive an update event even though the simulation object has changed state and its proxy has been altered to reflect that. This may sound puzzling at first, but is easily explained.

The subscriber holds a pointer to a list of time stamped proxies representing the changing state of the simulation object. When a subscriber requests a proxy, it receives the proxy reflecting the object's state at the subscriber's current time. So, if an update for an object is in the subscriber's future, there is no need to bother interrupting him with an update event. The proxy list is just quietly updated. If, in the future, the subscriber needs to reference the state of the object, the list simply returns the proxy corresponding to that time. The only case in which it is necessary to schedule an update event on the subscriber is when the subscriber's logical time is ahead of the update time. Since he may have referenced superceded data in a proxy, the subscriber must be rolled back to the time of the update; hence the need to schedule an event on the subscriber.

#### 9.4.5.2 Attribute Level Subscription and DDM

This section is included here for completeness. It depends on an understanding of DDM, which is discussed in Chapter 11. Readers not yet acquainted with DDM or who do not intend to use it, can safely skip this section.

The relationship between DDM and attribute level subscribing is simple: the latter acts as just another level of filtering, but applied only to object updates, not to object discoveries. With DDM, an object subscribing to a particular space discovers and receives updates for only those objects that fall within the parameters defining the space; all other objects are filtered out. By subscribing to a set of attributes, this object also filters out updates that do not pertain to attributes contained in the subscription set. The latter has no effect the discover/undiscover aspect of DDM, only the updating.

To use individual attribute subscribing in conjunction with DDM, simply create an attribute subscription set for each object class requiring this mode of update filtering (using the `S_SpHLA` methods described above). This can be done either before or after the DDM operations, and can be changed during the course of the simulation. The result is an independent layer of filtering that allows only those update events concerning attributes in the subscription set corresponding to the updated object's class. Of course, if no such subscription set exists, all updates get through.

### 9.4.6 Free Object Proxy

The final step to perform before using the object proxy system is to create the required implementation of a class `SpFreeObjProxy` constructor. This file must be compiled and linked with your simulation. This is required because SPEEDES uses free lists to streamline memory allocation. `SpFreeObjProxy` implements the proxy free lists. Example 9.3 shows the implementation of `SpFreeObjProxy` in its simplest form.

```
// Implementation for SpFreeObjProxy with no user-defined
// proxy classes:

#include "SpFreeObjProxy.H"

SpFreeObjProxy::SpFreeObjProxy(int n) {
  set_ntypes(n);
}
```

Example 9.3: Free Object Proxy Implementation

Section 9.5.2 will describe how to define your own proxy classes, which inherit from `SpObjProxy`. The reason for creating proxy classes is to provide subscribers with additional functionality beyond what is available in a generic proxy. If you choose to define your own proxy classes, it will be necessary to insert these proxies into the proxy framework. This is not shown in the example shown above. Example 9.15 shows how to plug user-defined proxies into the proxy framework.

For now, the important thing to note is that `SpFreeObjProxy.C` must be written, compiled, and linked with your simulation, and that the option exists to create specialized proxy classes. These specialized proxies correspond to classes defined in file `Objects.par`. Subscribers to these classes will receive the specialized proxy instead of a generic one.

## 9.5 Proxies in Use (Examples)

In this section, three code examples are examined in detail. The first two examples highlight the differences between an implementation that uses object proxies and one that does not. The first shows how attribute updates can be distributed to interested parties without the use of object proxies. The second implements the same scenario, but uses object proxies to automate attribute updates, thus demonstrating the advantages provided by this feature. The third example is a variation of the second, illustrating the attribute level subscription capabilities of object proxies.

### 9.5.1 Non-Proxy Example: Manually Pushing Attribute Updates Without Proxies

The code shown in Examples 9.4 through 9.8 creates a small simulation consisting of three radars and two airplanes. The radars are very powerful and "see" everything within a long range. Each radar maintains a list of all objects that they have detected.

In general, there are two basic schemes for distributing the data around to the objects that need it. The first choice is a "pull" mechanism, in which the objects that need data request or "pull" the data from the objects that have it. This can result in poor performance if the object that contains the data is continually getting rolled back due to requests for data in its past. Similarly, the requester could get rolled back as well, if it moves forward in time (because of events scheduled on it) before the reply event has arrived.

The second choice is a “push” mechanism where the objects that have the data send or “push” it automatically to the objects that want the data. This has the disadvantage of possibly sending too much data. However, this results in higher performance as long as the amount of extraneous data pushed is limited and since the pusher is not rolled back. The example described below uses the “push” technique to distribute their location to the radar objects. Examples 9.4 and 9.5 show the definition and implementation code for the F-15 aircraft.

```
// S_F15.H
#ifndef S_F15_H
#define S_F15_H

#include "SpSimObj.H"
#include "SpDefineSimObj.H"
#include "SpDefineEvent.H"

class S_F15 : public SpSimObj {
  public:
    S_F15() {}
    virtual ~S_F15() {}

    void Init();
    void Move(double latDelta, double lonDelta, double altDelta);

  private:
    RB_double Lat;
    RB_double Lon;
    RB_double Alt;
};

DEFINE_SIMOBJ(S_F15, 2, SCATTER);
DEFINE_SIMOBJ_EVENT_3_ARG(F15Move, S_F15, Move,
                          double, double, double);
#endif
```

Example 9.4: F-15 Definition (Manual Version)

```
// S_F15.C
#include "RB_SpRandom.H"
#include "S_F15.H"
#include "S_Radar.H"

void S_F15::Init() {
  double       x[3];
  RB_SpRandom* rand = SpGetRandom();
  double       time = rand->GenerateDouble(0, 100.0);

  rand->GenerateVector(x, 6378.145);
  Lat = x[0];
  Lon = x[1];
  Alt = x[2];
  rand->GenerateVector(x, time * 0.5);
  SCHEDULE_F15Move(time, SpGetObjHandle(), x[0], x[1], x[2]);
}

void S_F15::Move(double latDelta, double lonDelta, double altDelta) {
  int          i;
  double       x[3];
  SpSimTime    schedTime = SpGetTime();
  RB_SpRandom* rand      = SpGetRandom();
  double       time      = rand->GenerateDouble(0, 100.0);

  Lat += latDelta;
  Lon += lonDelta;
  Alt += altDelta;
  for (i = 0; i < S_Radar::GetNumRadars(); ++i) {
    SpObjHandle objHandle = SpGetObjHandle("S_Radar_MGR", i);
    SCHEDULE_RadarF15Move(SpGetTime(), objHandle,
                          SpGetObjHandle(), Lat, Lon, Alt);
  }
  rand->GenerateVector(x, time * 0.5);
  schedTime += time;
  SCHEDULE_F15Move(schedTime, SpGetObjHandle(), x[0], x[1], x[2]);
}
```

Example 9.5: F-15 Implementation (Manual Version)

The implementation file shows the usual `Init` method and a user-defined method called `Move`. Method `Init` initializes its position and schedules method `Move` with its current position as input parameters. When method `Move` executes, it updates it current position and schedules an event on each Radar notifying the radar of its new position (i.e. “push”). It then schedules an event for itself so that the F-15 can “move” its position again.

Examples 9.6 and 9.7 show the definition and implementation code for the Radars.

```
// S_Radar.H
#ifndef S_Radar_H
#define S_Radar_H

#include "SpSimObj.H"
#include "SpDefineSimObj.H"
#include "SpDefineEvent.H"

class S_Radar : public SpSimObj {
  public:
    S_Radar() {}
    virtual ~S_Radar() {}

    void Init() {}
    void Scan();
    void F15Move(SpObjHandle objHandle,
                 double lat, double lon, double alt);
    static int GetNumRadars() {return 3;}

  private:
    RB_SpList ListOfObjectsInTrack;
};
DEFINE_SIMOBJ(S_Radar, S_Radar::GetNumRadars(), SCATTER);
DEFINE_SIMOBJ_EVENT_0_ARG(RadarScan, S_Radar, Scan);
DEFINE_SIMOBJ_EVENT_4_ARG(RadarF15Move, S_Radar, F15Move,
                          SpObjHandle, double, double, double);
#endif
```

Example 9.6: Radar Definition (Manual Version)

```
// S_Radar.C
#include "RB_SpDefineClass.H"
#include "RB_ostream.H"
#include "S_Radar.H"

class ObjHandleAndPos {
  public:
    SpObjHandle handle;
    RB_double   lat;
    RB_double   lon;
    RB_double   alt;
    int operator ==(SpObjHandle& rhs) {
      return ((handle.GetSimObjMgrId() == rhs.GetSimObjMgrId()) &&
              (handle.GetNodeId()      == rhs.GetNodeId())      &&
              (handle.GetSimObjMgrId() == rhs.GetSimObjMgrId()));
    }
};
RB_DEFINE_CLASS(ObjHandleAndPos);

void S_Radar::Scan() {
  ObjHandleAndPos* obj =
    (ObjHandleAndPos *) ListOfObjectsInTrack.GetFirstElement();
  while (obj  != NULL) {
    RB_cout << GetName() << " sees object " << obj->handle.GetNodeId()
            << ", " << obj->handle.GetSimObjMgrId()
            << ", " << obj->handle.GetSimObjLocalId()
            << ", " << obj->lat
            << ", " << obj->lon
            << ", " << obj->alt << endl;
    obj = (ObjHandleAndPos *) ListOfObjectsInTrack.GetNextElement();
  }
  if (ListOfObjectsInTrack.GetNumElements() != 0) {
    SCHEDULE_RadarScan(SpGetTime() + 10.0, SpGetObjHandle());
  }
}

void S_Radar::F15Move(SpObjHandle objHandle,
                      double lat, double lon, double alt) {
  int              newObjectFlag = 0;
  ObjHandleAndPos* oHandle;

  /*
   * Search through the list of aircraft tracks looking for
   * the correct aircraft that is being tracked.
   */
  oHandle = (ObjHandleAndPos *) ListOfObjectsInTrack.GetFirstElement();
  while (oHandle != NULL) {
    if (*oHandle == objHandle) {
      break;
    }
    oHandle = (ObjHandleAndPos *) ListOfObjectsInTrack.GetNextElement();
  }
  /*
   * If we have a track on this aircraft, then oHandle will not be
   * equal to NULL.  If this is the first time we have seen this
```

```
     * track then create a new data element for which to save the
     * aircraft track parameters.
     */
    if (oHandle == NULL){
      oHandle         = RB_NEW_ObjHandleAndPos();
      oHandle->handle = objHandle;
      newObjectFlag   = 1;
    }
    /*
     * Update the track information with the aircrafts new position.
     */
    oHandle->lat = lat;
    oHandle->lon = lon;
    oHandle->alt = alt;
    /*
     * If the radar scan event is not running then start it.
     */
    if (ListOfObjectsInTrack.GetNumElements() == 0) {
      SCHEDULE_RadarScan(SpGetTime(), SpGetObjHandle());
    }
    /*
     * Add track to list if this is a new track.
     */
    if (newObjectFlag == 1) {
      ListOfObjectsInTrack.Insert(oHandle);
    }
}
```

Example 9.7: Radar Implementation (Manual Version)

With every F-15 position update, the F-15 schedules event `RadarF15Move` notifying the Radar of its current object id and position. Event `RadarF15Move` gets the old track report off of the list `ListOfObjectsInTrack` and, if not found, creates a new track report. The track report is updated with the F-15 unique object id and position and added to the track list. If this is the first object that this Radar has "seen", then it starts the Radar scan cycle, event `RadarScan`.

On a side note, this example also shows how to use macro `RB_DEFINE_CLASS` (see Section 5.1 for definition). This allows the Radar to rollbackably create and delete the track data. Since the position data changes over time and since the Radar could be rolled back, the position data is maintained with `RB_double` types. Since the object id is initialized only once, there is no need to worry about roll backs, hence the usage of the non-rollbackable type of `SpObjHandle`.

The final piece required for this example is `main`, which is shown in in Example 9.8

```
// Main.C
#include "S_F15.H"
#include "S_Radar.H"
#include "SpMainPlugIn.H"

int main(int argc, char** argv) {
  PLUG_IN_SIMOBJ(S_F15);
  PLUG_IN_SIMOBJ(S_Radar);

  PLUG_IN_EVENT(RadarF15Move);
  PLUG_IN_EVENT(RadarScan);
  PLUG_IN_EVENT(F15Move);
  ExecuteSpeedes(argc, argv);
}
```

Example 9.8: main Function (Manual Version)

Implementing a push system for data distribution can be quite efficient and an excellent way of distributing data for smaller simulations. However, as the number of objects begins to grow, managing the lists of subscribers can turn into a large and complex task. The SPEEDES built-in object proxy mechanism provides simple interfaces for automating the process of distributing data and for filtering the lists of recipients to only those objects that currently need the information.

### 9.5.2 Proxy Example: Automatically Pushing Attribute Updates With Proxies

The following example implements the previous "push" example using the built-in proxy mechanism. This automates the F-15 data delivery to the Radar simulation objects. First, an `Objects.par` file is needed to name the subscriber class, the publisher class, and to specify the public attributes. This is shown in Figure 9.2.

```
F15 {
  define double Latitude
  define double Longitude
  define double Altitude
}

Radar {
  reference SUBSCRIBE F15
}
```

Figure 9.2: Objects.par File (Proxy Version)

Notice that the F-15's position attributes are defined here. This will allow the Radar simulation object access to the F-15's position. The Radar also "subscribes" to the F-15s with line `reference SUBSCRIBE F15` in its definition. Now all F-15 simulation object proxies will be delivered to and will be available for the Radar to use.

Examples 9.9 and 9.10 show the definition and implementation code for the F-15 simulation objects.

```
// S_F15.H
#ifndef S_F15_H
#define S_F15_H

#include "S_SpHLA.H"
#include "SpDefineSimObj.H"
#include "SpDefineEvent.H"

class S_F15 : public S_SpHLA {
  public:
    S_F15() : S_SpHLA("F15") {}
    virtual ~S_F15() {}

    void Init();
    void Move(double latDelta, double lonDelta, double altDelta);

  private:
    DOUBLE_ATTRIBUTE Lat;
    DOUBLE_ATTRIBUTE Lon;
    DOUBLE_ATTRIBUTE Alt;
};

DEFINE_SIMOBJ(S_F15, 2, SCATTER);
DEFINE_SIMOBJ_EVENT_3_ARG(F15Move, S_F15, Move,
                          double, double, double);
#endif
```

Example 9.9: F-15 Definition (Proxy Version)

```
// S_F15.C
#include "RB_SpRandom.H"
#include "S_F15.H"

void S_F15::Init() {
  DEFINE_ATTRIBUTE(Lat, "Latitude");
  DEFINE_ATTRIBUTE(Lon, "Longitude");
  DEFINE_ATTRIBUTE(Alt, "Altitude");

  double       x[3];
  RB_SpRandom* rand = SpGetRandom();
  double       time = rand->GenerateDouble(0, 100.0);

  rand->GenerateVector(x, 6378.145);
  Lat = x[0];
  Lon = x[1];
  Alt = x[2];
  rand->GenerateVector(x, time * 0.5);
  SCHEDULE_F15Move(time, SpGetObjHandle(), x[0], x[1], x[2]);
}

void S_F15::Move(double latDelta, double lonDelta, double altDelta) {
  double       x[3];
  RB_SpRandom* rand      = SpGetRandom();
  SpSimTime    schedTime = SpGetTime();

  Lat += latDelta;
  Lon += lonDelta;
  Alt += altDelta;
  double time = rand->GenerateDouble(0, 100.0);
  rand->GenerateVector(x, time * 0.5);
  schedTime += time;
  SCHEDULE_F15Move(schedTime, SpGetObjHandle(), x[0], x[1], x[2]);
}
```

Example 9.10: F-15 Implementation (Proxy Version)

The definition for the F-15 is very similar to before. Notice that the F-15 simulation object now inherits from class `S_SpHLA`, rather than `SpSimObj`. The only other change is that this simulation object's attribute types of `RB_double` have changed to `DOUBLE_ATTRIBUTE`.

The F-15 implementation has changed a little bit more than the definition file. Notice that the constructor now passes the string "F15" to its base class `S_SpHLA`, which is the same string name used in `Objects.par`. As explained previously, this is a requirement. The `Init` method for this object is the same as before, except that the attributes are defined using the `DEFINE_ATTRIBUTE` macro. This example shows that the simulation object attribute name (e.g. `Lat`) does not have to match the string and `Objects.par` file attribute name. Event `F15Move` now updates its position and reschedules itself at the appropriate time so that it can continue to update its position. Notice that the F-15s do not know about the Radar. The F-15s are no longer scheduling an event on the Radars, indicating that they have changed their position. The Radars can now access the F-15s data via object proxies.

Since the F-15 has published attributes, subscribing objects will receive an `SpObjProxy` instance representing the published object. Users can define classes that inherit from the `SpObjProxy` class to create customized proxies geared to the needs of the particular classes in the simulation. One reason to do this is to provide methods that simplify the retrieval of the attribute values. The code shown in Example 9.11 illustrates what a user-defined proxy definition file looks like.

```
1  // F15Proxy.H
2  #ifndef F15Proxy_H
3  #define F15Proxy_H

4  #include "SpObjProxy.H"

5  class F15Proxy : public SpObjProxy {
6    public:
7      F15Proxy() {
8        LongitudeReference = GetReference("Longitude", "F15");
9      }
10     double GetLatitude()  {return *(DOUBLE_ATTRIBUTE *) Find("Latitude");}
11     double GetLongitude() {return GetFloat(LongitudeReference);}
12     double GetAltitude()  {return GetFloat("Altitude");}

13   private:
14     static int LongitudeReference;
15 };
16 #endif
```

Example 9.11: F-15 Object Proxy Definition

This proxy example shows three different methods for retrieving attributes from the proxy. Line 10 shows how to use the `SpObjProxy` method `Find(char*)`. Since `Find` returns a pointer to a base class, the return value must be casted to the correct type (a `DOUBLE_ATTRIBUTE*` in this case) before dereferencing.

Line 11 shows that the attributes can be looked up using type specific calls. This call is passed the integer reference (i.e. a unique id) for the attribute of interest, which, in this case, is the reference value for the `Longitude` attribute. The integer reference is constant for each proxy attribute and can be calculated once. This is shown on lines 7 through 9 with the use of the `GetReference` call. Notice that the `GetReference` name takes as inputs both the attribute and object string names, as defined in `Objects.par`.

Finally, line 12 shows how to look them up by using type specific calls and passing in the name of the attribute sought. This method of looking up attribute values is inherently slower than the integer reference version, since string compares are used.

Examples 9.12 and 9.13 show the new definition and implementation files for the Radar simulation object.

```
// S_Radar.H
#ifndef S_Radar_H
```

```
#define S_Radar_H

#include "S_SpHLA.H"
#include "SpDefineSimObj.H"
#include "SpDefineEvent.H"

class S_Radar : public S_SpHLA {
  public:
    S_Radar() : S_SpHLA("Radar") {}
    virtual ~S_Radar() {}

    void Init();
    void Scan();
};

DEFINE_SIMOBJ(S_Radar, 3, SCATTER);
DEFINE_SIMOBJ_EVENT_0_ARG(RadarScan, S_Radar, Scan);
#endif
```

Example 9.12: Radar Definition (Proxy Version)

```
// S_Radar.C
#include "RB_SpDefineClass.H"
#include "RB_ostream.H"
#include "F_SpProxyItem.H"
#include "S_Radar.H"
#include "F15Proxy.H"

void S_Radar::Init() {
  SCHEDULE_RadarScan(0.0, SpGetObjHandle());
}

void S_Radar::Scan() {
  int       i;
  RB_queue* aircraftProxy;

  aircraftProxy = GetRemoteObjectProxies();
  F_SpProxyItem* pItem = (F_SpProxyItem *) aircraftProxy->get_top();
  for (i = 0; i < aircraftProxy->get_length(); ++i){
    F15Proxy* mProxy = (F15Proxy *) pItem->GetObjProxy();
    RB_cout << GetName() << " sees object " << mProxy->GetProxyNode()
            << ", " << mProxy->GetProxySimObjMgrId()
            << ", " << mProxy->GetProxySimObjLocalId()
            << ", " << mProxy->GetLatitude()
            << ", " << mProxy->GetLongitude()
            << ", " << mProxy->GetAltitude()
            << endl;
    pItem = (F_SpProxyItem *) pItem->get_link();
  }
  SCHEDULE_RadarScan(SpGetTime() + 10.0, SpGetObjHandle());
}
```

Example 9.13: Radar Implementation (Proxy Version)

As with the F-15 simulation object, the Radar simulation object now inherits from `S_SpHLA`. The Radar constructor initializes the `S_SpHLA` constructor with the appropriate string for its class name, which is “Radar” in this case. Also notice that method `F15Move`, attribute `ListOfObjectsInTrack`, and event `RadarF15Move` have all been removed. This is because there is no need for the F-15 to manually “Push” or send its data to the Radar. The proxy mechanism will automatically delivery the data (i.e. positions) to the Radar for use.

The Radar implementation file has changed quite a bit more. Class `ObjHandleAndPos`, which was used to save the track data in the track list, has been deleted. This information is all available in the proxy data. Hence, there is no need for the Radar simulation object to save the data. Method `F15Move` has also been removed. Method `Init` has been added so that it can start event `RadarScan` at $t = 0.0$. Event `RadarScan` (i.e. method `Scan`) has also changed. Previously, this event went through each track in its own internal track list and printed the contents of each track. Now, instead of iterating through its own internal track list (which does not exist anyway), it now iterates through its proxy list (i.e. returned via `GetRemoteObjectProxies`). Since it subscribes to F-15 types, all F-15 simulation objects will be on this list (provided automatically by the proxy mechanism). Anytime a F-15 attribute changes, these changes are automatically propagated to each Radar. Therefore, each Radar always has the correct position for each F-15, regardless of attribute access time.

Function `main` also changed slightly, as shown in Example 9.14

```
// Main.C
#include "SpMainPlugIn.H"
#include "S_F15.H"
#include "S_Radar.H"
#include "F15Proxy.H"

void PlugInHLA();

int main(int argc, char **argv)
{
  PLUG_IN_SIMOBJ(S_F15);
  PLUG_IN_SIMOBJ(S_Radar);

  PLUG_IN_EVENT(RadarScan);
  PLUG_IN_EVENT(F15Move);

  PlugInHLA();                  // Required plugin with proxies

  ExecuteSpeedes(argc, argv);
}

int F15Proxy::LongitudeReference;
```

Example 9.14: main Function (Proxy Version)

In `main`, the `RadarF15Move` event does not need to be plugged in anymore (and indeed must not, since it is no longer defined) and a new call to `PlugInHLA` has been added. The latter call registers the events that provide the built-in proxy mechanism. For convenience, the definition of `MissileProxy::LongitudeReference` has been added to this file.

The final task to be completed for this example is to implement class `SpFreeObjProxy`. This is shown in Example 9.15.

```
// SpFreeObjProxy.C
#include "SpFreeObjProxy.H"
#include "F15Proxy.H"
```

```
4  enum {
5    F15_PROXY_ID,
6    N_FREE_PROXIES
7  };

8  void* newF15Proxy(int quanity) {
9    return new F15Proxy[quanity];
10 }

11 SpFreeObjProxy::SpFreeObjProxy(int n) {
12   set_ntypes(n);
13   set_type("F15", F15_PROXY_ID, newF15Proxy, sizeof(F15Proxy), 4);
14 }
```

Example 9.15: SpFreeObjProxy Implementation

This class manages the free lists for allocating proxies on each node. First, each proxy type must be assigned an unique integer id (lines 4 through 7). Then, an allocation function must be written for each proxy class (lines 8 through 10). As shown above, this function simply allocates the specified number of proxies.

The last step is to implement the constructor for class `SpFreeObjProxy` (lines 11 through 14). Specifically, line 12, `set_ntypes`, is always the first line in this class. This informs class `SpFreeObjProxy` how many proxy items are to be added. Line 13 adds the user-defined proxy for the F-15 to `SpFreeObjProxy` with method `set_type`. The API for this method is shown below:

```
set_type(char*        name,
         int          id,
         void*        f(int n),
         unsigned int size,
         int          nCreate)
```

| Parameter | Description |
|---|---|
| name | Name of the proxy. This string name must be identical to the name specified in `Objects.par`. |
| id | Unique integer id of this proxy type. Must be a number between 0 and the number of proxies added. |
| f(int n) | Function that returns an array of n proxies of this type. |
| size | Size in bytes of a proxy of this type. |
| nCreate | Number of proxies of this type to create at a time. |

Table 9.2: Free Object Proxy set_type API

### 9.5.3 Proxy Example: Using Attribute Level Subscription

In this section, the previous example will be varied slightly in order to demonstrate the attribute level subscription capabilities. Suppose the F-15 object included an additional "Temperature" attribute intended to model the heat given off by the engines. This might be used to estimate the plane's vulnerability to detection by infrared sensors, but is not relevant for Radar objects, since they cannot detect heat. Thus, a Radar object should not subscribe to the Temperature attribute, since this might cause the scheduling of useless update events on it. Such events would result in wasted computations and could roll back events that have already been processed on the Radar object (if the update time stamp is earlier than these events). Therefore, the Radar code will be modified to ensure that it does not subscribe to the Temperature attribute of the F-15 class. It will, however, discover all F-15 instances and receive all F-15 position updates.

The first modification is to Objects.par, in order to add the new attribute to the F-15 class. This is shown in Figure 9.3.

```
F15 {
  define double Latitude
  define double Longitude
  define double Altitude

  define double Temperature
}

// Radars still subscribe to the F15 class, but will remove
// the "Temperature" attribute from their attribute subscription set.
Radar {
  reference SUBSCRIBE F15
}
```

Figure 9.3: Objects.par File (Attribute Subscription Version

The code shown in Example 9.16 shows the new definition for the F-15 class. A new data member, `Temperature`, has been added, as well as a new method, called `ChangeTemperature`. Note that method `ChangeTemperature` has been turned into an event called `F15ChangeTemperature`. It will be scheduled every 20.0 seconds to increment the `Temperature` attribute. Thus, there will be a lot of update events generated by these changes. If the Radar objects do not remove the `Temperature` attribute from their attribute subscription, they will receive many unnecessary update events.

```
// S_F15.H
#ifndef S_F15_H
#define S_F15_H

#include "S_SpHLA.H"
#include "SpDefineSimObj.H"
#include "SpDefineEvent.H"

class S_F15 : public S_SpHLA {
  public:
    S_F15() : S_SpHLA("F15") {}
    virtual ~S_F15() {}

    void Init();
    void Move(double latDelta, double lonDelta, double altDelta);
    void ChangeTemperature();

  private:
    DOUBLE_ATTRIBUTE Lat;
    DOUBLE_ATTRIBUTE Lon;
    DOUBLE_ATTRIBUTE Alt;
    DOUBLE_ATTRIBUTE Temperature; // <-- New attribute
};

DEFINE_SIMOBJ(S_F15, 2, SCATTER);
DEFINE_SIMOBJ_EVENT_3_ARG(F15Move, S_F15, Move,
                          double, double, double);
DEFINE_SIMOBJ_EVENT_0_ARG(F15ChangeTemperature, S_F15, ChangeTemperature);
#endif
```

Example 9.16: F-15 Definition (Attribute Subscription Version)

The code shown in Example 9.17 shows the implementation for the F-15. In `Init`, there is an extra `DEFINE_ATTRIBUTE` call for the new attribute. Also, `Temperature` is initialized and the first `F15ChangeTemperature` event is scheduled. Finally, method `ChangeTemperature` is defined.

```
// S_F15.C
#include "RB_SpRandom.H"
#include "RB_ostream.H"
#include "S_F15.H"

void S_F15::Init() {
  DEFINE_ATTRIBUTE(Lat,         "Latitude" );
  DEFINE_ATTRIBUTE(Lon,         "Longitude");
  DEFINE_ATTRIBUTE(Alt,         "Altitude");
  DEFINE_ATTRIBUTE(Temperature, "Temperature");

  double       x[3];
  RB_SpRandom* rand = SpGetRandom();
  double       time = rand->GenerateDouble(0, 100.0);

  rand->GenerateVector(x, 6378.145);
  Lat = x[0];
  Lon = x[1];
  Alt = x[2];
  rand->GenerateVector(x, time * 0.5);
  SCHEDULE_F15Move(time, SpGetObjHandle(), x[0], x[1], x[2]);

  /*
   * Initialize Temperature and start chain of
   * F15ChangeTemperature events.
   */
  Temperature = 100.0;
  SCHEDULE_F15ChangeTemperature(0.0, SpGetObjHandle());
}

void S_F15::Move(double latDelta, double lonDelta, double altDelta) {
  double       x[3];
  RB_SpRandom* rand      = SpGetRandom();
  SpSimTime    schedTime = SpGetTime();

  Lat += latDelta;
  Lon += lonDelta;
  Alt += altDelta;
  double time = rand->GenerateDouble(0, 100.0);
  rand->GenerateVector(x, time * 0.5);
  schedTime += time;
  SCHEDULE_F15Move(schedTime, SpGetObjHandle(), x[0], x[1], x[2]);
}

// Increment the Temperature attribute and reschedule ChangeTemperature
void S_F15::ChangeTemperature() {
  double currTime = SpGetTime();
  Temperature++;

  // Change Temperature every 20.0 secs.
  SpSimTime nextTime = currTime + 20.0;

  SCHEDULE_F15ChangeTemperature(nextTime, SpGetObjHandle());

  int numUpdates = int(Temperature - 100.0);
  RB_cout << "F15#" << GetSimObjGlobalId() << ": New temp= "
          << int(Temperature) << " #Updates done= " << numUpdates
          << " at t= " << currTime << endl;
}
```

Example 9.17: F-15 Implementation (Attribute Subscription Version)

Next, we will look at the changes to the Radar class. This is where the key revisions occur, since the Radar does the subscribing and is, therefore, the place where attribute subscriptions take place. Example 9.18 shows the Radar class definition, which contains only two changes: the addition of the `CheckForSubscriptionChange` and the `EqualTime` methods. The first of these performs various changes to the attribute subscription at specified times in order to demonstrate the attribute level subscription capabilities. The second method is just a utility that checks if two simulation times are "close enough" to be considered equal.

```
// S_Radar.H
#ifndef S_Radar_H
#define S_Radar_H

#include "S_SpHLA.H"
#include "SpDefineSimObj.H"
#include "SpDefineEvent.H"

class S_Radar : public S_SpHLA {
  public:
    S_Radar() : S_SpHLA("Radar") {}
    virtual ˜S_Radar() {}

    void Init();
    void Scan();

  private:
    void CheckForSubscriptionChange(double currTime);
    int  EqualTime(double t0, double t1);
};

DEFINE_SIMOBJ(S_Radar, 3, SCATTER);
DEFINE_SIMOBJ_EVENT_0_ARG(RadarScan, S_Radar, Scan);
#endif
```

Example 9.18: Radar Definition (Attribute Subscription Version)

Finally, the code shown in Example 9.19 shows the implementation for the F-15 simulation object.

```
// S_Radar.C
#include "RB_SpDefineClass.H"
#include "RB_ostream.H"
#include "F_SpProxyItem.H"
#include "S_Radar.H"
#include "F15Proxy.H"

void S_Radar::Init() {
  const int numAtts = 3;
  char*     attNames[numAtts];

  attNames[0] = "Latitude";
  attNames[1] = "Longitude";
  attNames[2] = "Altitude";

  // Make attribute subscription set. { Latitude, Longitude, Altitude }
  SubscribeByAttribute("F15", numAtts, attNames);

  SCHEDULE_RadarScan(0.0, SpGetObjHandle());
}

void S_Radar::Scan() {
  int       i;
  RB_queue* aircraftProxy;
  double    currTime;

  currTime = double(SpGetTime());
  /*
   * If we are not subscribed to F15 "Latitude" attribute,
   * the radar is ON; otherwise, it is OFF.
   */
  int radarOn = CurrentlySubscribedToAtt("F15", "Latitude");

  if (radarOn != 0) {
    aircraftProxy = GetRemoteObjectProxies();
    F_SpProxyItem* pItem = (F_SpProxyItem *) aircraftProxy->get_top();
    for (i = 0; i < aircraftProxy->get_length(); ++i){
      F15Proxy* mProxy = (F15Proxy *) pItem->GetObjProxy();
      RB_cout << "t= " << currTime << ": "
              << GetName() << " sees object " << mProxy->GetProxyNode()
              << ", " << mProxy->GetProxySimObjMgrId()
              << ", " << mProxy->GetProxySimObjLocalId()
              << ", " << mProxy->GetLatitude()
              << ", " << mProxy->GetLongitude()
              << ", " << mProxy->GetAltitude();

      double temp = mProxy->GetFloat("Temperature");
      /*
       * Above attribute reference was "legal" if subscribed
       * to Temperature.
       */
      if (mProxy->LastAttribRefWasLegal()) {
        RB_cout << ", Temp= " << temp;
      }
      else {
        /*
         * Proxy disallows access to unsubscribed attributes; thus
         * GetFloat() should have returned an "error" value (e.g. -1e20).
         */
        RB_cout << " Temp not subscribed to";
      }
      RB_cout << endl;

      pItem = (F_SpProxyItem *) pItem->get_link();
    }
  }
  else { // Radar is OFF
    RB_cout << "t= " << currTime << ": "
            << GetName() << " radar is OFF." << endl;
  }
  /*
   * Check if it is time for a subscription change. If so, make
   * the change.
   */
  CheckForSubscriptionChange(currTime);

  SCHEDULE_RadarScan(SpGetTime() + 10.0, SpGetObjHandle());
}

void S_Radar::CheckForSubscriptionChange(double currTime) {
  const double FirstChangeTime = 1000.0; // Time of 1st change
  const double LastChangeTime  = 2400.0; // Time of last change
  const int    numAtts = 1;              // #Attributes added or removed
  char* attNames[numAtts];               // Names of attribs added or
                                         //  removed

  // Return if outside interval of changes:
  if ((currTime < (FirstChangeTime - 1.0)) ||
      (currTime > (LastChangeTime + 1.0))) {
    return;
  }

  /*
   * ------- CHANGE #1: Subscribe to Temperature attribute -------
   * Add Temperature attribute to subscription. We will get more update
   * events, but will have correct current Temperature:
   */
  if (EqualTime(currTime, FirstChangeTime)) {
    attNames[0] = "Temperature";

    AddAttributeSubscription("F15", numAtts, attNames);
    RB_cout << "**** " << GetName() << " subscribed to Temperature "
            << "attribute at t= "  << currTime << " ****" << endl;
    return;
  }

  /*
   * ------- CHANGE #2: Unsubscribe to Temperature attribute -------
   */
  if (EqualTime(currTime, 1200.0)) {
    attNames[0] = "Temperature";

    RemoveAttributeSubscription("F15", numAtts, attNames);
    RB_cout << "**** " << GetName() << " UN-subscribed to Temperature "
            << "attribute at t= " << currTime << " ****" << endl;
    return;
  }

  /*
   * ------- CHANGE #3: Turn OFF Radar -------
   * Unsubscribe all attributes so Radar does not "see" any updates:
   */
  if (EqualTime(currTime, 1500.0)) {
    UnSubscribeAllAttributes("F15");
    RB_cout << "**** " << GetName() << " UN-subscribed ALL "
            << "attributes at t= " << currTime << " ****" << endl;
    return;
  }

  /*
   * ------- CHANGE #4: Turn Radar back ON -------
   * Subscribe to {Latitude, Longitude, Altitude} in a different way:
   */
  if (EqualTime(currTime, LastChangeTime)) {
    SubscribeAllAttributes("F15");

    attNames[0] = "Temperature";
```

```
119     RemoveAttributeSubscription("F15", numAtts, attNames);
120     RB_cout << "**** " << GetName() << " RE-subscribed to (lat,lon,alt) "
121             << "attributes at t= " << currTime << " ****" << endl;
122     return;
123   }
124 }

125 /*
126  * Return 1 if t0 and t1 are close enough to be considered equal
127  * simulation time:
128  */
129 int S_Radar::EqualTime(double t0, double t1) {
130   return(fabs(t1 - t0) < 0.01); // close enough for our purposes
131 }
```

Example 9.19: S_Radar Implementation (Attribute Subscription Version)

First, take note of the `Init` method. To avoid `Temperature` updates, `SubscribeByAttribute` is invoked so that only attributes `Latitude`, `Longitude`, and `Altitude` are subscribed. It is still necessary to subscribe to the F-15 class, and this is done automatically because of the `reference SUBSCRIBE F15` line in `Objects.par`. Had that line not been there, it would have been necessary to call `Subscribe("F15")`, since `SubscribeByAttribute` does not subscribe to the class, it just defines the attribute subscription filter.

Next, examine the `Scan` method. As in the previous version, `Scan` schedules itself periodically, and prints out data contained in its proxy list. There are several changes in the simulation, but the basic concept of the revised version is this: initially, the Radar object is subscribed to attributes `Latitude`, `Longitude`, and `Altitude`. Then it adds in the `Temperature` attribute for an interval. Then it turns the Radar off for an interval (i.e. unsubscribes all attributes). Finally, it resubscribes to the original three attributes for the rest of the simulation. All the subscription changes are performed in the method `CheckForSubscriptionChange`. This method checks to see if the current time is one of the times in its schedule of subscription changes. If so, it performs the change.

When the radar is turned off, it unsubscribes to all attributes. Thus, we can determine if the radar is on by seeing if attribute `Latitude` is currently on the subscription list. This is done by using method `CurrentlySubscribedToAtt` on line 26. If it is on, then information from the proxy list is printed; otherwise, it prints a message saying that the radar is off.

On line 39, an attempt to access the `Temperature` attribute is made. If the object is currently subscribed to that attribute, the access will be "legal". This is checked by calling `LastAttribRef-WasLegal` on line 44. If the reference was legal, the current value of `Temperature` is printed. Otherwise, an the message "`Temp not subscribed to`" is printed.

The last method to be explained is `CheckForSubscriptionChange`. This method contains a schedule of four subscription changes to be performed. These demonstrate the use of various attribute level subscription methods.

Recall that `Init` subscribed to `Latitude`, `Longitude`, and `Altitude` attributes (and only these). The first subscription change (beginning on line 85) adds the `Temperature` attribute to the subscription using the `AddAttributeSubscription` method. This increases the number of update events scheduled on the subscriber, and also allows access to the `Temperature` attribute during the time interval when the object is subscribed to that attribute. Although only one attribute is added, `AddAt-tributeSubscription` permits an arbitrary number.

The second subscription change (beginning on line 95) unsubscribes the `Temperature` attribute by using `RemoveAttributeSubscription`. This returns the object's attribute subscription to the initial set of three attributes. Although only one attribute is removed, `RemoveAttributeSubscription` permits an arbitrary number.

The third subscription change (beginning on line 106) unsubscribes all F-15 attributes using method `UnSubscribeAllAttributes`. Throughout the interval during which it is not subscribed to any F-15 attributes, the subscriber will not receive any attribute update events, and cannot access any F-15 attributes in its proxies. It can, however, discover F-15 instances should any be dynamically created.

The fourth (and last) subscription change (beginning on line 100) resubscribes to attributes `Latitude`, `Longitude`, and `Altitude`, but does so in a different way than was done in `Init`. First, it subscribes to all F-15 attributes using `SubscribeAllAttributes`. Then, it removes the `Temperature` attribute with `RemoveAttributeSubscription`, thus achieving the desired attribute subscription.

## 9.6 Tips, Tricks, and Potholes

- Inheriting from `S_SpHLA` is required for all objects that wish to participate in the SPEEDES built-in publication or subscription services. However, it is not necessary to have all objects in a simulation inherit from `S_SpHLA` or from `SpSimObj` exclusively. There can be a mix of both types in a simulation, and they will work together. But only the objects inheriting from `S_SpHLA` have the ability to automatically share their attributes.

# Chapter 10

# Proxy Attributes

The previous chapter introduced HLA simulation objects and object proxies. This chapter discusses all of the built-in types that can be used as state variables when building HLA classes. The SPEEDES built-in attribute classes are fully rollbackable and provide automated proxy update capability. The previous chapter primarily showed these attributes from the publisher's perspective, but these attribute types are also used by the subscriber. Proxies delivered to subscribers contain these attributes. Therefore, it is necessary for subscribers to understand them, at least to the extent that is necessary to extract the received proxy data. This information is provided below with each attribute type description.

The attribute classes can be divided into two groups: static and dynamic attribute types. Static attributes are like the built-in data types in most programming languages. They are assigned a particular value (or values) and retain that state until reassigned. A dynamic attribute, on the other hand, contains a function, $f(t)$, specifying the values of the attribute over a period of time.

## 10.1 Static Attribute Types

These are the simple attribute types that provide basic capabilities similar to the built-in types available in C++. Besides automating updates, all the static attribute classes are rollbackable. Finally, all static attributes provide a virtual method called `outstream` for writing the attribute to an `ostream`. This means that each attribute type knows how to send a printable representation of itself to a file or the terminal and the fact that `outstream` is virtual means that it can be redefined in derived classes, if necessary.

### 10.1.1 Integer Numbers

Class `INT_ATTRIBUTE` is for attributes that take on integer values. It mimics the C++ `int` type and can be used in expressions just like an `int`. For example:

```
INT_ATTRIBUTE k = 12;
INT_ATTRIBUTE m = 41;

k = 3 * k + 5 * m - 27;
k++;
--m;
k += 117;
```

Specifically, the following operators have been overloaded to make the `INT_ATTRIBUTE` behave just like a C++ integer: `int(), =, ++` (prefix and postfix), `--` (prefix and postfix), `+=, -=, *=, /=, %=, >>=, <<=, ^=, &=`, and `|=`.

### 10.1.2 Floating Point Numbers

Class `DOUBLE_ATTRIBUTE` is for attributes that take on double precision floating point values. It mimics the C++ `double` type and can be used in expressions just like a `double`. For example:

```
DOUBLE_ATTRIBUTE x = 3.14;
DOUBLE_ATTRIBUTE y = -11.347;
double           z = 33.33;

x *= (4.77 / y) + z;
y  = (x + 4.2) * z;
z += x * x - y / 1.2;
```

Specifically, the following operators have been overloaded to make the `DOUBLE_ATTRIBUTE` behave like a C++ double: `double(), =, ++` (prefix and postfix), `--` (prefix and postfix), `+=, -=, *=`, and `/=`.

### 10.1.3 Booleans

Class `LOGICAL_ATTRIBUTE` is for boolean attributes, similar to the C++ `bool` type. It mimics the `bool` type when used in expressions. SPEEDES defines the `enum` values `LOGICAL_FALSE` and `LOGICAL_TRUE` for use with the `LOGICAL_ATTRIBUTE`, which can take on the values of 0 and 1, respectively. For example:

```
LOGICAL_ATTRIBUTE ready(LOGICAL_TRUE);
LOGICAL_ATTRIBUTE willing;
LOGICAL_ATTRIBUTE able = LOGICAL_TRUE;

if (ready != LOGICAL_TRUE) {
  cout << "Program is NOT ready to accept input" << endl;
}

willing = able || ready;
if (ready && willing && able) {
  getBusy();
}
```

Operators `int()` and = have been overloaded for class `LOGICAL_ATTRIBUTE`.

### 10.1.4 Character Strings

Class `STRING_ATTRIBUTE` is used for character strings, similar to a `char` array in C++. This class allocates its own memory to store the assigned string. Thus, in the following example, the overloaded = operator allocates storage and does a string copy into this storage; it does not merely copy a `char` pointer. This frees the user from having to keep track of the string storage assigned to a `STRING_ATTRIBUTE`.

```
  // Characters are copied into attribute.
  STRING_ATTRIBUTE BlimpName = "Hindenburg";
```

Available operators include `=, ==, !=,` and `const char*()`. The last of these converts the `STRING_ATTRIBUTE` into a `const char*` pointing to the allocated storage, thus allowing many of the usual string operations available in C++. The only operation that automatically sends updates and is rollbackable is the = operator. Note that casting away the `const` of the `const char*()` operator allows many additional C++ string operations. However, doing this results in non-rollbackable modifications to the `STRING_ATTRIBUTE` and, therefore, is not recommended.

### 10.1.5 Objects as Attributes

HLA simulation objects can have attributes which consists of another object (non-simulation). These attribute classes can then be composed of other proxy attributes. These user-defined attribute classes must inherit from `OBJECT_ATTRIBUTE` class. By using this class, users can create objects, which can then become an attribute of a HLA simulation object just like an `INT_ATTRIBUTE`, `DOUBLE_ATTRIBUTE`, etc.

The process of defining a child class of `OBJECT_ATTRIBUTE` is very similar to the process of defining a publishable HLA simulation object.

1. Describe the class in `Objects.par` and create a corresponding C++ class. This class will inherit from `OBJECT_ATTRIBUTE` (not `S_SpHLA`).
2. Declare the attribute type name by using the `SetClassName` method in the class constructor. As an alternative, the class name can be specified as a constructor input to `OBJECT_ATTRIBUTE`.
3. Use macro `DEFINE_ATTRIBUTE` to define all of the proxy attributes in the class.
4. Implement the virtual method `GetSize` for this class and all classes that may inherit from this new object class. Method `GetSize` should return the size of the class in bytes.

As an example, suppose we had a Car HLA simulation object and it contains an Engine, which will be an `ENGINE_ATTRIBUTE` inheriting from `OBJECT_ATTRIBUTE`. The `ENGINE_ATTRIBUTE` itself will have proxy attributes on it (e.g. `NumCylinders, RPMs, Temperature`), which could contain additional object attributes. This allows for an indefinite number of recursive layers, in which an object could contain an object attribute, which itself contains an object attribute, etc. all of which must inherit from `OBJECT_ATTRIBUTE`. Figure 10.1 shows the `Objects.par` file for this Car and Engine example. The Garage simulation object has been added in order to show how a subscriber object accesses the all of the proxies that it receives.

```
// Define the Car's attributes, including an Engine sub-object:
Car {
  define string ModelName
  define double Speed
  define object Engine
}

// Define the Engine's attributes:
Engine {
  define int    NumCylinders
  define int    RPMs
  define double Displacement
  define double Temperature
}

Garage {
  reference SUBSCRIBE Car
}
```

Figure 10.1: Objects.par File for OBJECT_ATTRIBUTE Example

Example 10.1 shows the code for the new `ENGINE_ATTRIBUTE`.

```
// ENGINE_ATTRIBUTE.H
#ifndef ENGINE_ATTRIBUTE_H
#define ENGINE_ATTRIBUTE_H

#include "SpObjProxy.H"
#include "SpExportAttribute.H"

class ENGINE_ATTRIBUTE : public OBJECT_ATTRIBUTE {
  public:
    ENGINE_ATTRIBUTE() : OBJECT_ATTRIBUTE("Engine") {
      if (SpPoDataBase::DoNotAllocateMemoryInConstructor == 1) {
        return;
      }
      /*
       * Hook attributes into proxy system.
       * Second argument is Objects.par name, not data member name!
       */
      DEFINE_ATTRIBUTE(NumCylinders, "NumCylinders");
      DEFINE_ATTRIBUTE(RevsPerMin,   "RPMs");
      DEFINE_ATTRIBUTE(EngineSize,   "Displacement");
      DEFINE_ATTRIBUTE(Temperature,  "Temperature");

      NumCylinders = 6;
      RevsPerMin   = 0;
      EngineSize   = 3.4;
      Temperature  = 20.2;
    }

    // NOTE: GetSize *MUST* be defined as follows.
    virtual int GetSize(void) {return sizeof(*this);}

    // Other methods can be defined.
    void   SetRevsPerMin(double rmp)   {RevsPerMin = rmp;}
    int    GetRevsPerMin(void)         {return RevsPerMin;}

    void   SetTemperature(double temp) {Temperature = temp;}
    double GetTemperature(void)        {return Temperature;}

  private:
    INT_ATTRIBUTE    NumCylinders;
    INT_ATTRIBUTE    RevsPerMin;
    DOUBLE_ATTRIBUTE EngineSize;
    DOUBLE_ATTRIBUTE Temperature;
};
#endif
```

Example 10.1: ENGINE_ATTRIBUTE Definition

This example shows the basic elements required when designing a user-defined attribute. The first step is that the attribute must inherit from `OBJECT_ATTRIBUTE`. When defining the attribute, pass its name must be passed `OBJECT_ATTRIBUTE` constructor. The name used here must be identical to the name used for the object definition in `Objects.par`. In this example, the string "Engine" is used. The next three lines are required, in order for the new attribute to be initialized correctly. Part of checkpoint/restart allocates and deallocates objects. Since this phase is not part of the proxy initialization, steps need to be taken to ensure that the proxy code is not executed, which is the purpose of `SpPoDataBase::DoNotAllocateMemoryInConstructor`.

The `DEFINE_ATTRIBUTE` is identical to what is used when defining attributes on an HLA object. Each attribute in the new object attribute requires this. Once again, the first argument in `DEFINE_ATTRIBUTE` is the name of the attribute found in this class. The second argument is the string name for this attribute, as defined in `Objects.par`. The only other required entry for a user-defined attribute is the definition of method `GetSize`.

Not shown in this example is the method `outstream` on `OBJECT_ATTRIBUTE`. The default method for `outstream` prints the data contained within the proxy. This includes the proxy name, the attribute names, and the attribute values. Since `outstream` is a virtual method, users can override their object attribute classes and customize their printed output.

The proxy definition file is fairly simple. It provides accessor methods to the engine proxy data for use by subscribers.

```
// EngineProxy.H
#ifndef EngineProxy_H
#define EngineProxy_H

#include "SpObjProxy.H"

class EngineProxy : public SpObjProxy {
  public:
    EngineProxy() {
      NumCylindersRef = GetReference("NumCylinders", "Engine");
      RPMsRef         = GetReference("RPMs",         "Engine");
      DisplacementRef = GetReference("Displacement", "Engine");
      TemperatureRef  = GetReference("Temperature",  "Engine");
    }
    int    GetNumCylinders() {return GetInt(NumCylindersRef);}
    int    GetRPMs()         {return GetInt(RPMsRef);}
    double GetDisplacement() {return GetFloat(DisplacementRef);}
    double GetTemperature()  {return GetFloat(TemperatureRef);}

  private:
    static int NumCylindersRef;
    static int RPMsRef;
    static int DisplacementRef;
    static int TemperatureRef;
};
#endif
```

Example 10.2: Engine Proxy Definition

Examples 10.3 and 10.4 show definition and implementation code for the Car simulation object (the methods for that car have been broken out into separate files so that additional methods can easily be added in later sections).

```
// S_Car.H
#ifndef S_Car_H
#define S_Car_H

#include "S_SpHLA.H"
#include "SpDefineSimObj.H"
#include "ENGINE_ATTRIBUTE.H"

class S_Car : public S_SpHLA {
  public:
    S_Car(char* objClassName = "Car") : S_SpHLA(objClassName) {}

    void   Init() {
      AddObjectAttributes();
    }
    void   AddObjectAttributes();

  private:
    STRING_ATTRIBUTE ModelName;
    DOUBLE_ATTRIBUTE Speed;
    ENGINE_ATTRIBUTE Engine;
};
DEFINE_SIMOBJ(S_Car, 2, SCATTER);
#endif
```

Example 10.3: Car HLA Simulation Object Definition

```
// S_Car_AddObjectAttibutes.C
#include "SpGlobalFunctions.H"
#include "S_Car.H"
#include "CarProxy.H"
#include "EngineProxy.H"

int CarProxy::ModelNameRef;
int CarProxy::SpeedRef;
int CarProxy::EngineRef;

int EngineProxy::NumCylindersRef;
int EngineProxy::RPMsRef;
int EngineProxy::DisplacementRef;
int EngineProxy::TemperatureRef;

void S_Car::AddObjectAttributes() {
  char name[64];

  DEFINE_ATTRIBUTE(ModelName, "ModelName");
  DEFINE_ATTRIBUTE(Speed,     "Speed");
  DEFINE_ATTRIBUTE(Engine,    "Engine");

  sprintf(name, "Car number %d", SpGetSimObjKindId());
  ModelName = name;
  Speed     = 30.0 * (1 + SpGetSimObjKindId());
  Engine.SetTemperature(20.1 * (1 + SpGetSimObjKindId()));
  Engine.SetRevsPerMin(1000 * (1 + SpGetSimObjKindId()));
}
```

Example 10.4: Car HLA Simulation Object AddObjectAttributes Method

The definition and implementation for the Car simulation object should look fairly familiar. The only new item introduced here is shown in Example 10.3, a user-defined attribute called ENGINE_ATTRIBUTE. Example 10.4 initializes all of the attributes found on the Car.

Example 10.5 shows the code for the user-defined proxy for the ENGINE_ATTRIBUTE.

```
// CarProxy.H
#ifndef CarProxy_H
#define CarProxy_H

#include "SpObjProxy.H"
#include "ENGINE_ATTRIBUTE.H"
#include "EngineProxy.H"

class CarProxy : public SpObjProxy {
  public:
    CarProxy() {
      ModelNameRef = GetReference("ModelName", "Car");
      SpeedRef     = GetReference("Speed",     "Car");
      EngineRef    = GetReference("Engine",    "Car");
    }
    const char*  GetModelName() {return GetString(ModelNameRef);}
    double       GetSpeed()     {return GetFloat(SpeedRef);}
    EngineProxy* GetEngine() {
      ENGINE_ATTRIBUTE* engineAtt = (ENGINE_ATTRIBUTE *) Find(EngineRef);
      return (EngineProxy *) engineAtt->GetObjProxy();
    }

  private:
    static int ModelNameRef;
    static int SpeedRef;
    static int EngineRef;
};
#endif
```

Example 10.5: Car Proxy Definition

As usual, the car proxy provides accessors for the different attributes on the car. No new concepts are introduced here except for the retrieval of the ENGINE_ATTRIBUTE. This can be confusing because the retrieval of a user-defined object attribute is a two-step process. The first step requires the use of the Find method with the type casting of its return value to its appropriate user-defined object attribute type (i.e. ENGINE_ATTRIBUTE in this case). Once the object attribute has been found, then method GetObjProxy must be called on that attribute in order to get its proxy (e.g. engine attribute proxy shown in Example 10.2).

Example 10.6 and 10.7 show the code for the Garage definitions and implementation files, respectively.

```
// S_Garage.H
#ifndef S_Garage_H
#define S_Garage_H
```

```

#include "S_SpHLA.H"
#include "SpDefineSimObj.H"
#include "SpDefineEvent.H"

class S_Garage : public S_SpHLA {
  public:
    S_Garage(char* objClassName = "Garage") : S_SpHLA(objClassName) {}

    void Init();
    void Display();
};
DEFINE_SIMOBJ(S_Garage, 1, SCATTER);
DEFINE_SIMOBJ_EVENT_0_ARG(Garage_Display, S_Garage, Display);
#endif
```

Example 10.6: Garage HLA Simulation Object Definition

```
// S_Garage.C
#include "F_SpProxyItem.H"
#include "S_Garage.H"
#include "CarProxy.H"
#include "EngineProxy.H"

void S_Garage::Init() {
  SCHEDULE_Garage_Display(0.0, SpGetObjHandle());
}

void S_Garage::Display() {
  int       i;
  RB_queue* remoteProxyList;  // Car Proxies in this case.

  RB_cout << SpGetTime() << endl;
  remoteProxyList = GetRemoteObjectProxies();
  F_SpProxyItem* proxyItem = (F_SpProxyItem *) remoteProxyList->get_top();
  for (i = 0; i < remoteProxyList->get_length(); ++i) {
    CarProxy*    carProxy = (CarProxy *) proxyItem->GetObjProxy();
    EngineProxy* engProxy = carProxy->GetEngine();
    RB_cout << carProxy->GetModelName()                      << endl
      << "Car:Speed=        " << carProxy->GetSpeed()        << endl
      << "Eng:Cylinders=    " << engProxy->GetNumCylinders() << endl
      << "Eng:Displacement= " << engProxy->GetDisplacement() << endl
      << "Eng:RPMs=         " << engProxy->GetRPMs()         << endl
      << "Eng:Temperature=  " << engProxy->GetTemperature()  << endl
      << endl;
    proxyItem = (F_SpProxyItem *) proxyItem->get_link();
  }
  RB_cout << endl;
  SCHEDULE_Garage_Display(SpGetTime() + 1000.0, SpGetObjHandle());
}
```

Example 10.7: Garage HLA Simulation Object Implementation

The Garage simulation object shows how a subscribing object accesses its received proxies. This object has one event called `Garage_Display` that executes at $t = 0.0$ and every 1000.0 seconds thereafter. As explained in the previous section, the subscribing object gains access to the objects that it subscribed to though its remote proxy list. Each proxy is pulled of the remote proxy list to be examined. For this example, the Garage only received Car proxies. If it had subscribed to other object types, then it would have to apply the appropriate type cast to each returned proxy. In this case, the returned proxy was type cast to a `CarProxy` pointer at which time the convenience methods provided for the Car proxy can be use to access its contents.

The final two files necessary to complete this example are shown in Examples 10.8 and 10.9.

```
// SpFreeObjProxy.C
#include "SpFreeObjProxy.H"
#include "CarProxy.H"
#include "EngineProxy.H"

enum {
  CAR_PROXY_ID,
  ENGINE_PROXY_ID,
  N_FREE_PROXIES
};

void* newCarProxy(int quanity) {
  return new CarProxy[quanity];
}
void* newEngineProxy(int quanity) {
  return new EngineProxy[quanity];
}

SpFreeObjProxy::SpFreeObjProxy(int n) {
  set_ntypes(n);
  set_type("Car", CAR_PROXY_ID, newCarProxy,
           sizeof(CarProxy), 4);
  set_type("Engine", ENGINE_PROXY_ID, newEngineProxy,
           sizeof(EngineProxy), 4);
}
```

Example 10.8: Free Object Proxy Implementation for Object Attribute Example

```
// Main.C
#include "SpMainPlugIn.H"
#include "S_Car.H"
#include "S_Garage.H"

void PlugInHLA();

int main(int argc, char **argv) {
  PLUG_IN_SIMOBJ(S_Car);
  PLUG_IN_SIMOBJ(S_Garage);

  PLUG_IN_EVENT(Garage_Display);

  PlugInHLA();                  // Required plugin with proxies

  ExecuteSpeedes(argc, argv);
}
```

Example 10.9: Main Implementation for Object Attribute Example

Since this example created two proxies, the implementation for `SpFreeObjProxy` must call method `set_type` for each proxy. The simulation object and event plug-ins are called by `main`.

### 10.1.6 Lists

Class `LIST_ATTRIBUTE` can be used to create unordered lists of objects. More precisely, this attribute class stores an unordered list of pointers to objects attributes, all of which must have inherited from the `OBJECT_ATTRIBUTE`. The following methods or operator overloads are available for navigating through a list and inserting or removing items from a list:

```
class LIST_ATTRIBUTE {
  public:
    int   GetNumElements();
    void* GetFirstElement();
    void* GetLastElement();
    void* operator ++ ();
    void* operator -- ();
    void  operator +=(OBJECT_ATTRIBUTE* item);
    void  operator -=(OBJECT_ATTRIBUTE* item);
};
```

- `GetNumElements`:
  Returns the number of items on the lists.
- `GetFirstElement`:
  Returns the first item on the list. The return value must be type casted to its appropriate object attribute type.
- `GetLastElement`:
  Returns the last item on the list. The return value must be type casted to its appropriate object attribute type.
- `++`:
  Gets the next element on the list. Its return value must be type casted to its appropriate object attribute type.
- `--`:
  Gets the previous item on the list. Its return value must be type casted to its appropriate object attribute type.
- `+=`:
  Inserts the item onto the list.
- `-=`:
  Removes the item from the list.

The list attribute is a non-ordered list of objects which have inherited from `OBJECT_ATTRIBUTE`. These lists can only be traversed one element at a time in either a forward or reverse direction. As each element is accessed, it must be type casted to its appropriate type.

The Car example has been modified to show how to create a list, modify and remove elements on the list, and how a subscriber can examine the contents of the list. A list of Fuses has been added to this example. Figure 10.2 shows the new `Objects.par` file.

```
// Define the Car's attributes, including an Engine sub-object:
Car {
  define string ModelName
  define double Speed
  define object Engine
  define list   FuseList
}

// Define the Engine’s attributes:
Engine {
  define int    NumCylinders
  define int    RPMs
  define double Displacement
  define double Temperature
}

// Define the Fuse’s attributes:
Fuse {
  define int     IdNumber
  define logical State
  define int     Amperage
}

Garage {
  reference SUBSCRIBE Car
}
```

Figure 10.2: Objects.par File for LIST_ATTRIBUTE Example

The Car object in `Objects.par` has a new attribute on it called `list`. Notice that this is a generic list of items (i.e. no type specification for items on the list). A new Fuse object has been added. The Fuse object contains three attributes. These Fuse objects will be created and added to the list.

Examples 10.10 and 10.11 show the code for the Fuse object and the Fuse proxy, respectively.

```
// Fuse.H
#ifndef Fuse_H
#define Fuse_H

#include "SpObjProxy.H"
#include "SpExportAttribute.H"

class Fuse : public OBJECT_ATTRIBUTE {
  public:
    Fuse() : OBJECT_ATTRIBUTE("Fuse") {
      if (SpPoDataBase::DoNotAllocateMemoryInConstructor == 1) {
        return;
      }
      /*
       * Hook attributes into proxy system:
       * Second argument is Objects.par name, not data member name!
       */
      DEFINE_ATTRIBUTE(IdNumber, "IdNumber");
      DEFINE_ATTRIBUTE(State,    "State");
      DEFINE_ATTRIBUTE(Amperage, "Amperage");

      IdNumber = -1;
      State    = LOGICAL_TRUE;
      Amperage = -2;
    }

    // NOTE: GetSize *MUST* be defined as follows:
    virtual int GetSize(void) {return sizeof(*this);}

    // Other methods can be defined:
    void   SetId(int id)         {IdNumber = id;}
    void   SetState(int inState) {State    = inState;}
    void   SetAmperage(int amp)  {Amperage = amp;}

  private:
    INT_ATTRIBUTE     IdNumber;
    LOGICAL_ATTRIBUTE State;         // 0 - Bad; 1 - Good
    INT_ATTRIBUTE     Amperage;
};
#endif
```

Example 10.10: Fuse Definition

```
// FuseProxy.H
#ifndef FuseProxy_H
#define FuseProxy_H

#include "SpObjProxy.H"

class FuseProxy : public SpObjProxy {
  public:
    FuseProxy() {
      IdNumberRef = GetReference("IdNumber", "Fuse");
      StateRef    = GetReference("State",    "Fuse");
      AmperageRef = GetReference("Amperage", "Fuse");
    }
    int GetId()       {return GetInt(IdNumberRef);}
    int GetState()    {return GetLogical(StateRef);}
    int GetAmperage() {return GetInt(AmperageRef);}

  private:
    static int IdNumberRef;
    static int StateRef;
    static int AmperageRef;
};
#endif
```

Example 10.11: Fuse Proxy Definition

As stated earlier, any object that is inserted onto a `LIST_ATTRIBUTE` must be derived from an `OBJECT_ATTRIBUTE`. As shown above, the Fuse object is following this rule. As usual, the name of the Fuse object is passed to the `OBJECT_ATTRIBUTE` constructor, which is the same name found in the `Objects.par` file. The three attributes defined in the `Objects.par` are also defined in this class definition.

The proxy for the Fuse is shown in Example 10.11. This proxy definition is not presenting any new proxy concepts. It simply is providing convenience accessor methods for the proxy data to subscriber(s).

Changes to the Car object are shown in Examples 10.12 through 10.14.

```
// S_Car.H
#ifndef S_Car_H
#define S_Car_H

#include "S_SpHLA.H"
#include "SpDefineSimObj.H"
#include "SpDefineEvent.H"
#include "ENGINE_ATTRIBUTE.H"

class S_Car : public S_SpHLA {
  public:
    S_Car(char* objClassName = "Car") : S_SpHLA(objClassName) {}

    void   Init() {
      AddObjectAttributes();
      AddListAttributes();
    }
    void   AddObjectAttributes();
    void   AddListAttributes();
    void   RemoveFuseItem();

  private:
    STRING_ATTRIBUTE  ModelName;
    DOUBLE_ATTRIBUTE  Speed;
    ENGINE_ATTRIBUTE  Engine;
    LIST_ATTRIBUTE    FuseList;
};
DEFINE_SIMOBJ(S_Car, 2, SCATTER);
DEFINE_SIMOBJ_EVENT_0_ARG(Car_RemoveFuseItem, S_Car, RemoveFuseItem);
#endif
```

Example 10.12: Car HLA Simulation Object Definition

```
#include "SpGlobalFunctions.H"
#include "S_Car.H"
#include "Fuse.H"
#include "CarProxy.H"
#include "FuseProxy.H"

int CarProxy::FuseListRef;

int FuseProxy::IdNumberRef;
int FuseProxy::StateRef;
int FuseProxy::AmperageRef;

void S_Car::AddListAttributes() {
  int          i;
  Fuse*        fuseObj;
  RB_SpRandom* random = SpGetRandom();

  DEFINE_ATTRIBUTE(FuseList, "FuseList");

  for (i = 0; i < 5; ++i) {
    fuseObj = new Fuse;
    fuseObj->SetId(i);
    fuseObj->SetState(random->GenerateInt(0, 1));
    fuseObj->SetAmperage(5 * ((i + 1) + SpGetSimObjKindId()));
    FuseList += fuseObj;
  }
}
```

Example 10.13: Car HLA Simulation Object AddListAttributes Method

```
// S_Car_RemoveFuseItem.C
#include "SpGlobalFunctions.H"
#include "S_Car.H"
#include "Fuse.H"

void S_Car::RemoveFuseItem() {
  int          i = 0;
  Fuse       *fuseItem    = (Fuse *) FuseList.GetFirstElement();
  Fuse       *tmpFuseItem;
  while (fuseItem != NULL) {
    if (3 == ++i) {
      tmpFuseItem = fuseItem;
    }
    if (5 == i) {
      fuseItem->SetAmperage(100);
    }
    fuseItem = (Fuse *) ++FuseList;
  }
  FuseList -= tmpFuseItem;
}
```

Example 10.14: Car HLA Simulation Object RemoveFuseItem Method

The Car class definition has two changes worth mentioning. The first is that the `FuseList` attribute has been added to this class. The second change is the two methods added called `AddListAttributes` and `RemoveFuseItem`. Method `AddListAttributes` adds the `FuseList` attribute to the proxy framework and initializes the list. Method `RemoveFuseItem` will be turned into an event by the `DEFINE_SIMOBJ_EVENT` macro. This method is used to show how to remove an item from the list and how to modify an item on the list.

Example 10.13 shows the `DEFINE_ATTRIBUTE` being used to add attribute `FuseList` to the proxy framework. Afterwards, five Fuses are added to the list. Notice that each Fuse is allocated individually on the heap with its own `new` call. The `LIST_ATTRIBUTE` destructor runs through its list and calls `delete` on each pointer (not `delete[]`). Therefore, if any of the items in the list were allocated as part of an array (e.g. `new Fuse[n]`), or is not part of the heap (i.e. global, static, or local storage), then the proxy memory manager will become corrupted. This usually results in the application running incorrectly, perhaps even coring. Each Fuse is then initialized and added to the list by using the += list operator.

Method `RemoveFuseItem` has been turned into event `Car_RemoveFuseItem`, which will execute at $t = 500.0$. This method shows how to search through a list. List method `GetFirstElement` to get the first item on the list. Then, each item on the list is examined in the `while` loop. The third item is removed by the list operator -= and the fifth item is modified with methods on the Fuse. The list operator ++ gets the next item on the list. The point of interest here is that the Car simulation object (publisher) uses the actual methods on the Fuse object attribute to modify its contents and not the method on the Fuse's object proxy.

Code for the subscriber (i.e. Garage) is shown in Example 10.15.

```
// S_Garage.C
#include "F_SpProxyItem.H"
#include "S_Garage.H"
#include "CarProxy.H"
#include "EngineProxy.H"
#include "Fuse.H"
#include "FuseProxy.H"

void S_Garage::Init() {
  SCHEDULE_Garage_Display(0.0, SpGetObjHandle());
}

void S_Garage::Display() {
  int       i;
  RB_queue* remoteProxyList;  // Car Proxies in this case.

  RB_cout << SpGetTime() << endl;
  remoteProxyList = GetRemoteObjectProxies();
  F_SpProxyItem* proxyItem = (F_SpProxyItem *) remoteProxyList->get_top();
  for (i = 0; i < remoteProxyList->get_length(); ++i) {
    CarProxy*       carProxy = (CarProxy *) proxyItem->GetObjProxy();
    EngineProxy*    engProxy = carProxy->GetEngine();
    LIST_ATTRIBUTE* fuseList = carProxy->GetFuseList();
    Fuse*           fuseItem;
    FuseProxy*      fuseProxy;
    RB_cout << carProxy->GetModelName()                      << endl
      << "Car:Speed=        " << carProxy->GetSpeed()        << endl
      << "Eng:Cylinders=    " << engProxy->GetNumCylinders() << endl
      << "Eng:Displacement= " << engProxy->GetDisplacement() << endl
      << "Eng:RPMs=         " << engProxy->GetRPMs()         << endl
      << "Eng:Temperature=  " << engProxy->GetTemperature()  << endl;
    fuseItem = (Fuse *) fuseList->GetFirstElement();
    while (fuseItem != NULL) {
      fuseProxy = (FuseProxy *) fuseItem->GetObjProxy();
      RB_cout
        << "Fuse:Id=          " << fuseProxy->GetId()       << endl
        << "Fuse:State=       " << fuseProxy->GetState()    << endl
        << "Fuse:Amperage=    " << fuseProxy->GetAmperage() << endl;
      fuseItem = (Fuse *) ++(*fuseList);
    }
    RB_cout << endl;

    proxyItem = (F_SpProxyItem *) proxyItem->get_link();
  }
  RB_cout << endl;
  SCHEDULE_Garage_Display(SpGetTime() + 1000.0, SpGetObjHandle());
}
```

Example 10.15: Garage HLA Simulation Object Implementation

The code for displaying the previously defined Car and Engine attributes remained the same. New code has been added for displaying the Fuse attribute. The first step is to get the pointer to the list which contains the Fuses. This is done with the proxy convenience method `GetFuseList`. Once the list is retrieved, the first item on the list is retrieved with list method `GetFirstElement`. Next, a `while` loop is entered, which allows the Garage to access each item in the list. Since the Fuse is an object attribute (i.e. a derived class from `OBJECT_ATTRIBUTE`), method `GetObjProxy` must be called on the Fuse attribute in order to get the Fuse proxy (i.e. `FuseProxy`). Once this is done, then the attributes on the Fuse's proxy can be accessed by using the Fuse's convenience accessor methods. It is worth noting here that the subscriber cannot access the Fuse's attributes directly with methods on the fuse. Users must get the proxy for the fuse.

Also, not shown here is the implementation for class `SpFreeObjProxy` or `main`. The Fuse proxy (i.e. `FuseProxy`) has been added to `SpFreeObjProxy` and event `Car_RemoveFuseItem` has been plugged into the SPEEDES framework in `main`.

When this example executes at $t = 0.0$ the Garage prints out all of the Car attributes, including the values for the five Fuses. At $t = 500.0$, the third Fuse is removed and the fifth item `Amperage` was changed to 100. At $t = 1000.0$ and all subsequent displays by the Garage, the Fuse list only contains four Fuse's. The `Amperage` for the fourth Fuse (was fifth) will now display 100.

### 10.1.7 Binary Buffer Data (Character Pointer to Non-String Data)

Class `BINARY_BUFFER_ATTRIBUTE` works on raw buffer of data. This attribute is ideal for use with images, compressed files, specially formatted message data, etc. The following methods are available on the `BINARY_BUFFER_ATTRIBUTE`:

```
void        CopyIntoBuffer(char* srcBuff, int srcBytes)
const char* GetBuffPtr()
int         GetBuffSize()
```

- `CopyIntoBuffer`:
  Copies raw byte data into the attribute. Parameter `srcBuff` is a pointer to data that contains `srcBytes` of data. It then allocates storage and copies the data into the new buffer. This method is rollbackable and sends automatic updates to subscribers. The code below shows an example of how to set data in a `BINARY_BUFFER_ATTRIBUTE`.

  ```
  BINARY_BUFFER_ATTRIBUTE secretMsgBuff;
  int                     msgBytes;

  // Encrypts msgData and returns number of bytes via msgBytes arg.
  char* msg = encryptSecretMsg(msgData, &msgBytes);

  // Rollbackably change attribute; distribute to subscribers.
  secretMsgBuff.CopyIntoBuffer(msg, msgBytes);
  ```

- `GetBuffPtr`:
  Returns a character pointer to the data in the buffer.

- `GetBuffSize`:
  Returns the size of the data in the buffer in bytes.

Subscribers retrieve the data with method `GetBuffPtr`. It is assumed that the receiver knows how to interpret the raw data. Therefore, the buffer pointer and buffer size are the only information that class `BINARY_BUFFER_ATTRIBUTE` provides. The code below shows an example of how to retrieve data in a `BINARY_BUFFER_ATTRIBUTE`.

```
BINARY_BUFFER_ATTRIBUTE* msgBuff;
```

```
msgBuf            = (BINARY_BUFFER_ATTRIBUTE *)
  spyObjProxy->Find("SecretMsgBuff");
char* encryptedMsg = msgBuf->GetBuffPtr();
int   msgBytes     = msgBuf->GetBuffSize();

char* plainText    = decryptSecretMsg(encryptedMsg, msgBytes);
```

The same warning regarding the `STRING_ATTRIBUTE` applies to this attribute: if the publisher obtains a pointer into the attribute's data and alters the data, these changes will not be distributed to subscribers, nor will they be rollbackable. The only operation that automatically performs these tasks is method `CopyIntoBuffer`.

### 10.1.8 Static Positions

Class `POSITION_ATTRIBUTE` provides storage for a fixed object's position (i.e. a position that does not move), typically described by a latitude, longitude, and altitude triplicate. By "fixed" we do not mean that this attributes value must remain constant and can never change throughout a simulation. They can be changed and all changes will be sent to the appropriate subscribers. However, if the object is going to have a continuous motion, it is better to use the `DYNAMIC_POSITION_ATTRIBUTE` (see Section 10.2.4).

The `POSITION_ATTRIBUTE` has three modes, corresponding to the three coordinate systems it can work with. The default is called `EARTH` coordinates. In this mode, you set the position by passing the latitude, longitude, and altitude into method `SetEARTH`. The units are radians for latitude and longitude, and kilometers (km) above sea level for altitude. `SetEARTH` is a rollbackable operation and, when used, causes attribute updates to be sent out to all subscribers. The `GetEARTH` method retrieves the current latitude, longitude, and altitude.

The current mode (i.e. coordinate system) can be retrieved from a `POSITION_ATTRIBUTE` by the method `GetPositionType`. The three possible return values are `EARTH`, `ECI`, or `ECR` (defined constants). As already defined, `EARTH` corresponds to a latitude, longitude, and altitude triplicate. `ECI` and `ECR` correspond to "Earth Centered Inertial" and "Earth Centered Rotating" coordinate systems. These are Cartesian coordinate systems whose origin is at the center of the earth. `ECI` is a coordinate system that is fixed in space (i.e. never moves) with its z axis pointing through the North Pole. Earth is then rotating around the z axis. Therefore, a fixed position on earth `ECI` coordinates are always changing as time increases. `ECR` is a coordinate system that is fixed to earth with its z axis also pointing through the North Pole. However, as the earth rotates, the `ECR` coordinate system rotates, as well. Therefore, a fixed position on a rotating earth never has its coordinates change as time advances. The units used in the `ECR` and `ECI` coordinate systems are km. `EARTH` is similar to `ECR` in that the coordinate system is fixed to earth.

The `POSITION_ATTRIBUTE` has methods for inserting or extracting the position in any of the three coordinate systems. The `SetEARTH` method takes latitude, longitude, and altitude arguments and inserts them into the attribute. It also sets the position type to `EARTH`. The `SetECR` and `SetECI` take x, y, and z arguments, inserts them into the attribute, and sets the position type to `ECR` or `ECI`, respectively. Positions can be extracted in any of the three forms, regardless of the current setting of the coordinate system or position type. If the requested coordinate system is different from the current form, a conversion is done automatically. Thus, `GetEARTH`, `GetECR`, or `GetECI`, can be invoked at anytime and the position will be retrieved in the correct frame of reference. The `GetECI` method requires a `time` argument (seconds past midnight) in order to do the conversion. See the SPEEDES API Reference Manual for additional details explanation). Obviously, it is most efficient to store and extract position in the same coordinate system, since then no conversion is needed. Also, whenever the current coordinate system is `ECI`, you must provide a time argument to `GetEARTH` or `GetECR` since the `EARTH` or `ECR` coordinate systems vary with time, relative to the `ECI` system. The units on the time argument are seconds past midnight.

The set and get methods on a `POSITION_ATTRIBUTE` are shown below:

```
  void SetEARTH(double lat, double lon, double alt)
  void SetECI(double position[3])
  void SetECR(double position[3])
  void GetEARTH(double& lat, double& lon, double& alt, double t = 0.0)
  void GetECI(double position[3], double t = 0.0)
  void GetECR(double position[3], double t = 0.0)
```

| Parameter | Description |
|---|---|
| lat | Latitude element for the simulation object position. |
| lon | Longitude element for the simulation object position. |
| alt | Altitude element for the simulation object position. |
| position | Simulation object position triplicate, corresponding to the $(x, y, z)$ position of the object. |
| t | Time past midnight for which the position is to be returned. |

Table 10.1: Static Position Get and Set Methods

The Car example has been expanded to include attribute types of `BINARY_BUFFER_ATTRIBUTE` and `POSITION_ATTRIBUTE`. Figure 10.3 shows the new `Objects.par` file.

```
// Define the Car's attributes, including an Engine sub-object:
Car {
  define string         ModelName
  define double         Speed
  define object         Engine
  define list           FuseList
  define binary_buffer  Message
  define position       StaticPosition
}

// Define the Engine's attributes:
Engine {
  define int    NumCylinders
  define int    RPMs
  define double Displacement
  define double Temperature
}

// Define the Fuse's attributes:
Fuse {
  define int     IdNumber
  define logical State
  define int     Amperage
}

Garage {
  reference SUBSCRIBE Car
}
```

Figure 10.3: Objects.par File for BINARY_BUFFER_ATTRIBUTE and POSITION_ATTRIBUTE

The `Objects.par` file shows two new attributes on the Car object called `Message` and `StaticPosition`. The new code to support these two attributes is shown in Examples 10.16 and 10.17.

```
// S_Car.H
#ifndef S_Car_H
#define S_Car_H

#include "S_SpHLA.H"
#include "SpDefineSimObj.H"
#include "SpDefineEvent.H"
#include "ENGINE_ATTRIBUTE.H"

class S_Car : public S_SpHLA {
  public:
    S_Car(char* objClassName = "Car") : S_SpHLA(objClassName) {}

    void   Init() {
      AddObjectAttributes();
      AddListAttributes();
      AddPositionAttributes();
    }
    void   AddObjectAttributes();
    void   AddListAttributes();
    void   AddPositionAttributes();
    void   RemoveFuseItem();

  private:
    STRING_ATTRIBUTE        ModelName;
    DOUBLE_ATTRIBUTE        Speed;
    ENGINE_ATTRIBUTE        Engine;
    LIST_ATTRIBUTE          FuseList;
    BINARY_BUFFER_ATTRIBUTE Message;
    POSITION_ATTRIBUTE      StaticPosition;
};
DEFINE_SIMOBJ(S_Car, 2, SCATTER);
DEFINE_SIMOBJ_EVENT_0_ARG(Car_RemoveFuseItem, S_Car, RemoveFuseItem);
#endif
```

Example 10.16: Car HLA Simulation Object Definition

```
// S_Car_AddPositionAttributes.C
#include "SpGlobalFunctions.H"
#include "S_Car.H"

#include "CarProxy.H"

int CarProxy::MessageRef;
int CarProxy::StaticPositionRef;

void S_Car::AddPositionAttributes() {
  char   tempString[] = "Test Binary Attribute Data";
  double position[3] = {1.0, 2.0, 3.0};

  DEFINE_ATTRIBUTE(Message,        "Message");
  DEFINE_ATTRIBUTE(StaticPosition, "StaticPosition");

  Message.CopyIntoBuffer(tempString, strlen(tempString) + 1);
  StaticPosition.SetECR(position);
}
```

Example 10.17: Car HLA Simulation Object AddPositionAttributes Method

The Car definition file adds attributes `Message` and `StaticPosition` as types `BINARY_BUFFER_ATTRIBUTE` and `POSITION_ATTRIBUTE`, respectively. Method `AddPositionAttributes` is new to this class. It is responsible for initializing the two new attributes.

Method `AddPositionAttributes` has been added to `Init` (not shown here) so that the new attributes are initialized. This method registers the attributes `Message` and `StaticPosition` to the proxy framework. The binary buffer is initialized with the string “`Test Binary Attribute Data`”. Since this is a string, the string attribute would have been more appropriate. However, for demonstration purposes, the binary attribute was initialized to a string. This method also initializes the Car’s static position to $(1.0, 2.0, 3.0)$.

Next, the Car’s proxy needs to be modified, as shown in Example 10.18.

```
// CarProxy.H
#ifndef CarProxy_H
#define CarProxy_H

#include "SpObjProxy.H"
#include "ENGINE_ATTRIBUTE.H"
#include "EngineProxy.H"

class CarProxy : public SpObjProxy {
  public:
    CarProxy() {
      ModelNameRef       = GetReference("ModelName",      "Car");
      SpeedRef           = GetReference("Speed",          "Car");
      EngineRef          = GetReference("Engine",         "Car");
      FuseListRef        = GetReference("FuseList",       "Car");
      MessageRef         = GetReference("Message",        "Car");
      StaticPositionRef  = GetReference("StaticPosition", "Car");
    }
    const char*  GetModelName() {return GetString(ModelNameRef);}
    double       GetSpeed()     {return GetFloat(SpeedRef);}
    EngineProxy* GetEngine() {
      ENGINE_ATTRIBUTE* engineAtt = (ENGINE_ATTRIBUTE *) Find(EngineRef);
      return (EngineProxy *) engineAtt->GetObjProxy();
    }
    LIST_ATTRIBUTE* GetFuseList() {return GetList(FuseListRef);}
    const char*     GetBinaryBuffer(int& size) {
      BINARY_BUFFER_ATTRIBUTE* msgBuff = (BINARY_BUFFER_ATTRIBUTE *)
        Find(MessageRef);
      size = msgBuff->GetBuffSize();
      return msgBuff->GetBuffPtr();
    }
    void             GetEARTH(double pos[3], double time = 0.0) {
      GetPosition(StaticPositionRef, pos, EARTH, time);
    }
    void             GetECR(double pos[3], double time = 0.0) {
      GetPosition(StaticPositionRef, pos, ECR, time);
    }
    void             GetECI(double pos[3], double time = 0.0) {
      GetPosition(StaticPositionRef, pos, ECI, time);
    }

  private:
    static int ModelNameRef;
    static int SpeedRef;
    static int EngineRef;
    static int FuseListRef;
    static int MessageRef;
    static int StaticPositionRef;
};
#endif
```

Example 10.18: Car Proxy Definition

This example illustrates how to extract binary data from a BINARY_BUFFER_ATTRIBUTE using its method GetBuffPtr. This example also has three methods for getting the object's static position, one for each coordinate system of EARTH, ECR, and ECI.

The last step shown here is to add code to the Garage simulation object so that it can print the values for its received message and position. The new implementation for the Garage is shown in Example 10.19.

```
// S_Garage.C
#include "F_SpProxyItem.H"
#include "S_Garage.H"
#include "CarProxy.H"
#include "EngineProxy.H"
#include "Fuse.H"
#include "FuseProxy.H"

void S_Garage::Init() {
  SCHEDULE_Garage_Display(0.0, SpGetObjHandle());
}

void S_Garage::Display() {
  int         i;
  RB_queue*   remoteProxyList;  // Car Proxies in this case.
  const char* msg;
  int         msgSize;
  double      position[3];

  RB_cout << SpGetTime() << endl;
  remoteProxyList = GetRemoteObjectProxies();
  F_SpProxyItem* proxyItem = (F_SpProxyItem *) remoteProxyList->get_top();
  for (i = 0; i < remoteProxyList->get_length(); ++i) {
    CarProxy*       carProxy = (CarProxy *) proxyItem->GetObjProxy();
    EngineProxy*    engProxy = carProxy->GetEngine();
    LIST_ATTRIBUTE* fuseList = carProxy->GetFuseList();
    Fuse*           fuseItem;
    FuseProxy*      fuseProxy;
    RB_cout << carProxy->GetModelName()                      << endl
      << "Car:Speed=        " << carProxy->GetSpeed()        << endl
      << "Eng:Cylinders=    " << engProxy->GetNumCylinders() << endl
      << "Eng:Displacement= " << engProxy->GetDisplacement() << endl
      << "Eng:RPMs=         " << engProxy->GetRPMs()         << endl
      << "Eng:Temperature=  " << engProxy->GetTemperature()  << endl;
    fuseItem = (Fuse *) fuseList->GetFirstElement();
    while (fuseItem != NULL) {
      fuseProxy = (FuseProxy *) fuseItem->GetObjProxy();
      RB_cout
        << "Fuse:Id=          " << fuseProxy->GetId()        << endl
        << "Fuse:State=       " << fuseProxy->GetState()     << endl
        << "Fuse:Amperage=    " << fuseProxy->GetAmperage()  << endl;
      fuseItem = (Fuse *) ++(*fuseList);
    }
    msg = carProxy->GetBinaryBuffer(msgSize);
    carProxy->GetECR(position);
    RB_cout << "Car:Message=      " << msg         << endl
            << "Car:Position[0]=  " << position[0] << endl
            << "Car:Position[1]=  " << position[1] << endl
            << "Car:Position[2]=  " << position[2] << endl;
    RB_cout << endl;

    proxyItem = (F_SpProxyItem *) proxyItem->get_link();
  }
  RB_cout << endl;
  SCHEDULE_Garage_Display(SpGetTime() + 1000.0, SpGetObjHandle());
}
```

Example 10.19: Garage HLA Simulation Object Implementation

New code for printing the binary buffer and position attribute data was added after the Fuse output section. Since these items are attributes of the Car, the Car proxy must be used when accessing these data values.

When the example executes, in addition to the previously explained output data, the message is printed out (i.e. “`Test Binary Attribute Data`”) and the `ECR` position of $(1.0, 2.0, 3.0)$.

## 10.2 Dynamic Attribute Types

As opposed to the static attribute type discussed in the preceding sections, dynamic attributes can be preprogrammed to take different values as simulation time advances. This allows dynamic attribute types to be specified as a function, $f(t)$, that defines the values attributes can take on over intervals of time (i.e. $AttributeValue = f(t), \forall t_{min} \leq t \leq t_{max}$).

There are dynamic attribute type equivalents for integers, booleans, doubles, and positions. Perhaps the most important of these attributes is the dynamic position attribute. If a simulation object updated its position at regular time intervals and the position consisted of $(x, y, z)$ components for position, velocity, and acceleration, then all subscribers would receive these updated position when they were made. This would cause publisher to have to constantly update their position, and subscribers would constantly receive these position updates causing them to rollback. If the simulation object path is known in advance, then the positions can be preprogrammed during initialization as a position equation as a function of time. This preprogrammed equation could then be delivered to all of the subscribers through the proxy delivery system. Now, whenever a subscriber wants to know the position of a simulation object for which it has its proxy, it can simply ask for the position at some point in time, thus not causing any rollbacks.

For example, consider an integer step function, $g(t)$, over the interval $[0.0, 5.0]$ defined as follows:

$$g(t) = \begin{cases} 10 & 0.0 < t < 1.8 \\ 15 & 1.8 < t < 3.2 \\ 50 & 3.2 < t < 5.0 \end{cases}$$

This equation specifies the value of $g(t)$ over three different time intervals. In general, users that create dynamic attributes determine the value (or equation) of the output data during specific time intervals. Once this is determined, the data for each time interval is added to the dynamic attribute.

### 10.2.1 Basic Concepts

The basic dynamic attribute consists of a list of values or an equation of values. When a request is made for the attribute at a certain time, the list is searched for the interval that is specified by the requested time, and the data value at this time is returned. For example, consider the equation $g(t)$ above. For this example, three entries (i.e. dynamic items) have to be made in order to specify all of the data for this equation. These values are 10 between the range of $0.0 < t < 1.8$, 15 between the range of $1.8 < t < 3.2$, and 50 between the range of $3.2 < t < 5.0$. Once the dynamic attribute is defined, then its value can be queried at a certain time. For example, if $g(t)$ is queried at 1.5, then the dynamic attribute would return 10 for its value.

All dynamic attribute's inheritance chain, consist of a class called `BASE_DYNAMIC_ATTRIBUTE`, which then inherits from `LIST_ATTRIBUTE`. Therefore, all dynamic attributes have the same API as list attribute. Dynamic attributes use the list attribute `+=` and `-=` operators to add and remove dynamic items from the list. Class `BASE_DYNAMIC_ATTRIBUTE` adds some additional functionality, which is shown below:

```
class BASE_DYNAMIC_ATTRIBUTE : LIST_ATTRIBUTE {
  public:
    SpDynItem* FindDynamicItem(double time);
    SpDynItem* FindDynamicItem(int id);
    double     GetStartTime();
    double     GetEndTime();
    void       GetTimeInterval(double& startTime, double& endTime);
    void       outstream(ostream& out, int indentLevel);
};
```

- `FindDynamicItem`:
  Returns a pointer to the dynamic item. The item returned will be based on if there is an interval defined for the `time` specified, otherwise, `NULL` is returned. If an integer id is supplied, then the item with this id is returned, otherwise, `NULL` is returned.
- `GetStartTime`:
  Returns the minimum start time of all dynamic items loaded onto the dynamic attribute list.
- `GetEndTime`:
  Returns the maximum end time of all dynamic items loaded onto the dynamic attribute list.

- `GetTimeInterval`:
  Returns the minimum and maximum times for which the attribute contains data. That is `start-Time` will be set to `GetStartTime`, and `endTime` will be set to `GetEndTime`.

- `outstream`:
  This virtual method prints out the list of dynamic items contained on the dynamic attribute list. The `indentLevel` argument controls the indentation level (five spaces per level).

Dynamic items are added to the dynamic attribute list. Each built-in dynamic attribute has one or more dynamic items which can be used to build the behavior of the dynamic attribute. All dynamic items inherit from the `SpDynItem`. This class essentially consists of a start time, end time, and an integer id. All dynamic items have the following API.

```
class SpDynItem {
  public:
    virtual void   SetStartTime(double startTime);
            double GetStartTime();
    virtual void   SetEndTime(double endTime);
            double GetEndTime();
            int    GetCountId();
};
```

- `SetStartTime`:
  Sets this dynamic item's start time to `startTime`.

- `GetStartTime`:
  Returns the start time for this dynamic item.

- `SetEndTime`:
  Sets this dynamic item's end time to `endTime`.

- `GetEndTime`:
  Returns the end time for this dynamic item.

- `GetCountId`:
  Returns the id for the dynamic item.

As shown in Section 10.1.6, items to be inserted on lists, in general, are allocated from memory with `new` and deallocated with `delete`. Allocating and deallocating memory during run time can be an expensive operation. In order to minimize these affects, all dynamic attributes are preallocated and saved on a dynamic attributes free list (i.e. `SpFreeDynAttributes`). Free lists contain two functions that are used to retrieve and return items from the free list called:

```
C_ITEM* RB_FREE_NEW    (C_FREE_LIST* freelist, int     type)
void    RB_FREE_DELETE (C_FREE_LIST* freelist, C_ITEM* object)
```

- `RB_FREE_NEW`:
  This function returns an item off of the `freelist` as specified by `type`. In the case of the dynamic attribute free list, it returns a dynamic attribute (which was derived from `SpDynItem` which was derived from `C_ITEM`).

- `RB_FREE_DELETE`:
  Returns the dynamic attribute back to the free list.

Dynamic attributes are built by placeing dynamic attribute items onto the attribute. Dynamic attribute items are retrieved and returned from `SpFreeDynAttributes` using `RB_FREE_NEW` and `RB_FREE_DELETE`, respectively. Table 10.2 shows a list of built-in dynamic attribute items and their respective enumeration identifier. The first column specifies the available types, which are discuss in more detail in the following sections. The second column specifies what dynamic attribute types are available in SPEEDES. These items are initialized and inserted onto the dynamic attribute. The third column specifies the name to use with the dynamic attribute free list in order to retrieve the desired dynamic attribute item.

| Attribute Type | Dynamic Attribute Item | Free List Identifier type |
|---|---|---|
| integer | DYNAMIC_INT_CONSTANT | DYNAMIC_INT_CONSTANT_ID |
| double | DYNAMIC_DOUBLE_CONSTANT | DYNAMIC_DOUBLE_CONSTANT_ID |
| boolean | DYNAMIC_DOUBLE_CONSTANT | DYNAMIC_LOGICAL_CONSTANT_ID |
| one dimensional functions | DYNAMIC_POLY_1 | DYNAMIC_POLY_1_ID |
| | DYNAMIC_POLY_2 | DYNAMIC_POLY_2_ID |
| | DYNAMIC_POLY_3 | DYNAMIC_POLY_3_ID |
| | DYNAMIC_POLY_4 | DYNAMIC_POLY_4_ID |
| | DYNAMIC_POLY_5 | DYNAMIC_POLY_5_ID |
| | DYNAMIC_POLY_6 | DYNAMIC_POLY_6_ID |
| | DYNAMIC_POLY_7 | DYNAMIC_POLY_7_ID |
| | DYNAMIC_POLY_8 | DYNAMIC_POLY_8_ID |
| | DYNAMIC_POLY_9 | DYNAMIC_POLY_9_ID |
| | DYNAMIC_POLY_10 | DYNAMIC_POLY_10_ID |
| | DYNAMIC_SPLINE_3 | DYNAMIC_SPLINE_3_ID |
| | DYNAMIC_SPLINE_6 | DYNAMIC_SPLINE_6_ID |
| | DYNAMIC_COMPLEX_EXPONENTIAL | DYNAMIC_COMPLEX_EXPONENTIAL_ID |
| | DYNAMIC_EXPONENTIAL | DYNAMIC_EXPONENTIAL_ID |
| | DYNAMIC_EXTRAPOLATE | DYNAMIC_EXTRAPOLATE_ID |
| | SPLINE3_MOTION | SPLINE3_MOTION_ID |
| | SPLINE6_MOTION | SPLINE6_MOTION_ID |
| position | POLY_1_MOTION | POLY_1_MOTION_ID |
| | POLY_2_MOTION | POLY_2_MOTION_ID |
| | POLY_3_MOTION | POLY_3_MOTION_ID |
| | POLY_4_MOTION | POLY_4_MOTION_ID |
| | POLY_5_MOTION | POLY_5_MOTION_ID |
| | POLY_6_MOTION | POLY_6_MOTION_ID |
| | POLY_7_MOTION | POLY_7_MOTION_ID |
| | POLY_8_MOTION | POLY_8_MOTION_ID |
| | POLY_9_MOTION | POLY_9_MOTION_ID |
| | POLY_10_MOTION | POLY_10_MOTION_ID |
| | GREAT_CIRCLE | GREAT_CIRCLE_ID |
| | RHUMB_LINE | RHUMB_LINE_ID |
| | Elliptical | Elliptical_ID |
| | CIRCULAR_ORBIT | CIRCULAR_ORBIT_ID |
| | CONSTANT_MOTION_ID | CONSTANT_MOTION_ID |
| | LOITER_MOTION_ID | LOITER_MOTION_ID |
| | EXTRAPOLATE_MOTION_ID | EXTRAPOLATE_MOTION_ID |

Table 10.2: Dynamic Attribute Identifiers

### 10.2.2 Dynamic Integers, Doubles and Booleans

When an integer, double, or boolean data values change in a predetermined or scripted fashion over time, then these behaviors can be modeled by using one of the built-in types of `DYNAMIC_INT_ATTRIBUTE`, `DYNAMIC_DOUBLE_ATTRIBUTE`, or `DYNAMIC_LOGICAL_ATTRIBUTE`. The basic method for using any of these attributes is the same. Therefore, only the procedure for using an integer is described below and if another type is desired (i.e. double or boolean), simply exchange “`INTEGER`” with “`DOUBLE`” or “`LOGICAL`”.

1. Add an attribute whose type is `dynamic_int` to the appropriate class definition in file `Ob-`

jects.par. For doubles or booleans, add dynamic_double or dynamic_logical, respectively (e.g. lines 3, 4, and 5 in Figure 10.4).

2. Create a DYNAMIC_INT_ATTRIBUTE type attribute on the simulation object.

3. Use DEFINE_ATTRIBUTE to associate the simulation object with the proxy framework in the simulation object's Init method.

4. Use type DYNAMIC_INT_CONSTANT_ID to retrieve a DYNAMIC_INT_CONSTANT from the free dynamic attribute list.

5. Initialize the dynamic attribute by:

   (a) Set the data value for DYNAMIC_INT_CONSTANT. Use method Set.

   (b) Set the start time for DYNAMIC_INT_CONSTANT. Use method SetStartTime.

   (c) Set the end time for DYNAMIC_INT_CONSTANT. Use method SetEndTime.

6. Insert the DYNAMIC_INT_CONSTANT onto the DYNAMIC_INT_ATTRIBUTE list with += operator.

7. Repeat steps 4 through 6 for each data point to be added to the dynamic integer.

For example, consider the previous $g(t)$ example repeated below for convenience:

$$g(t) = \begin{cases} 10 & 0.0 < t < 1.8 \\ 15 & 1.8 < t < 3.2 \\ 50 & 3.2 < t < 5.0 \end{cases}$$

The built-in type of DYNAMIC_INT_ATTRIBUTE would be a perfect fit for this type of equation. Three DYNAMIC_INT_CONSTANT items are retrieved from the free dynamic attribute list (i.e. FreeDynamicAttributes). Each DYNAMIC_INT_CONSTANT is initialized with its appropriate magnitude, start, and end times. Insert the DYNAMIC_INT_CONSTANT into the DYNAMIC_INT_ATTRIBUTE list using the += operator.

Notice at time 1.8 and 3.2, that there is a discontinuity in $g(t)$. What would happen if users asked for the value of $g(t)$ at $t = 1.8$. The current behavior of the dynamic attribute is undefined at this point. In the case of step functions, this is problematic. In other cases, such as motion types, it is possible and often wise to define the dynamic attribute values such that the end point of one dynamic attribute item is the starting point of the next dynamic attribute item.

Also, what happens if the time intervals overlap or there are gaps? Behavior in regions whose intervals overlap is undefined and the returned data value is indeterminate. Attributes whose definitions contain gaps will produce an error if access is attempted in a gap.

As with static attributes, proxies convenience methods can be created to assist the subscribers when accessing the data on their received proxies. Accessing the remote proxy list and retrieving each proxy on the list is identical to what has already been described. Examples showing how to use the attribute types of DYNAMIC_INT_ATTRIBUTE, DYNAMIC_DOUBLE_ATTRIBUTE, and DYNAMIC_LOGICAL_ATTRIBUTE are shown below.

The Car and Garage simulation objects have been modified to show the usage of DYNAMIC_INT_ATTRIBUTE, DYNAMIC_DOUBLE_ATTRIBUTE, and DYNAMIC_LOGICAL_ATTRIBUTE. The new Objects.par file is shown in Figure 10.4.

```
Car {
  define string          ModelName
  define dynamic_int     Speed
  define dynamic_double  RadioFrequency
  define dynamic_logical IgnitionState
}

Garage {
  reference SUBSCRIBE Car
}
```

Figure 10.4: Objects.par File for Dynamic Base Type Example

Each of the previously discussed dynamic attribute is represented here. This example builds dynamic attributes in 300.0 second intervals. The Garage simulation object examines each car object's attributes, and prints out their values. The Garage does not examine the attributes at the interval transition points. Examples 10.20 through 10.22 show the definition and implementation for the Car simulation object.

```
// S_Car.H
#ifndef S_Car_H
#define S_Car_H

#include "S_SpHLA.H"
#include "SpDefineSimObj.H"
#include "SpDefineEvent.H"

class S_Car : public S_SpHLA {
  public:
    S_Car(char* objClassName = "Car") : S_SpHLA(objClassName) {}

    void   Init() {
      InitDynamicBaseTypes();
    }
    void   InitDynamicBaseTypes();
    void   ChangeDynamicBaseTypes();

  private:
    STRING_ATTRIBUTE          ModelName;
    DYNAMIC_INT_ATTRIBUTE     Speed;
    DYNAMIC_DOUBLE_ATTRIBUTE  RadioFrequency;
    DYNAMIC_LOGICAL_ATTRIBUTE IgnitionState;
};
DEFINE_SIMOBJ(S_Car, 2, SCATTER);
DEFINE_SIMOBJ_EVENT_0_ARG(Car_ChangeDynamicBaseTypes,
                          S_Car, ChangeDynamicBaseTypes);
#endif
```

Example 10.20: Car HLA Simulation Object Definition

The class definition creates three attributes, which will be used to illustrate how to create and use dynamic attributes. Method `InitDynamicBaseTypes` will be used to initialize the attributes on this class. Method `ChangeDynamicBaseTypes` is turned into event `Car_ChangeDynamicBaseTypes`. This event is used to illustrate how items on a dynamic attribute are removed and modified.

```
// S_Car_InitDynamicBaseTypes.C
#include "SpGlobalFunctions.H"
#include "SpFreeDynAttributes.H"
#include "RB_SpFrameworkFuncs.H"
#include "SpDynObjs.H"
#include "S_Car.H"
#include "CarProxy.H"

int CarProxy::ModelNameRef;
int CarProxy::SpeedRef;
int CarProxy::RadioFrequencyRef;
int CarProxy::IgnitionStateRef;

void S_Car::InitDynamicBaseTypes() {
  char                        name[64];
  int                         speed;
  double                      rFreq;
  double                      i;
  double                      stepSize = 300.0;
  DYNAMIC_INT_CONSTANT*       intItem;
  DYNAMIC_DOUBLE_CONSTANT*    doubleItem;
  DYNAMIC_LOGICAL_CONSTANT*   logicalItem;

  DEFINE_ATTRIBUTE(ModelName,      "ModelName");
  DEFINE_ATTRIBUTE(Speed,          "Speed");
  DEFINE_ATTRIBUTE(RadioFrequency, "RadioFrequency");
  DEFINE_ATTRIBUTE(IgnitionState,  "IgnitionState");

  sprintf(name, "Car number %d", SpGetSimObjKindId());
  ModelName = name;
  speed     = 30 * (1 + SpGetSimObjKindId());
  rFreq     = 100.0;
  for (i = 50.0; i < 3700.0; i = i + stepSize) {
    /*
     * Retrieve a dynamic integer from dynamic attribute free list.
     * Initialize its value.
     */
    intItem     = (DYNAMIC_INT_CONSTANT *)
      RB_FREE_NEW(FreeDynamicAttributes, DYNAMIC_INT_CONSTANT_ID);
    intItem->Set(speed % 60);
    intItem->SetStartTime(i);
    intItem->SetEndTime(i + stepSize);
    Speed += intItem;

    /*
     * Retrieve a dynamic double from dynamic attribute free list.
     * Initialize its value.
     */
    doubleItem  = (DYNAMIC_DOUBLE_CONSTANT *)
      RB_FREE_NEW(FreeDynamicAttributes, DYNAMIC_DOUBLE_CONSTANT_ID);
    doubleItem->Set((double) (((int) rFreq) % 100));
    doubleItem->SetStartTime(i);
    doubleItem->SetEndTime(i + stepSize);
    RadioFrequency += doubleItem;
    logicalItem = (DYNAMIC_LOGICAL_CONSTANT *)
      RB_FREE_NEW(FreeDynamicAttributes, DYNAMIC_LOGICAL_CONSTANT_ID);

    /*
     * Retrieve a dynamic logical from dynamic attribute free list.
     * Initialize its value.
     */
    logicalItem->Set(1);
    if (speed % 60 == 0) {
      logicalItem->Set(0);
    }
    logicalItem->SetStartTime(i);
    logicalItem->SetEndTime(i + stepSize);
    IgnitionState += logicalItem;

    speed = speed + 10;
    rFreq = rFreq + 10.0 + SpGetSimObjKindId();
  }

  SCHEDULE_Car_ChangeDynamicBaseTypes(1750.0, SpGetObjHandle());
}
```

Example 10.21: Car HLA Simulation Object InitDynamicBaseTypes Method

Method `Init` performs performs the usual initializations for simulation objects. In this case method `Init` calls method `InitDynamicBaseTypes`. Each dynamic attribute is pulled off of the free list (i.e. `FreeDynamicAttributes`) according to the specified type. Once a dynamic item has been retrieved, its value, start, and end times are initialized. Notice that there are no overlaps or gaps in the time intervals. This allows subscribers to access the values for this attribute anywhere within the valid range, except for exactly on an interval transition point. Also, notice that the first interval starts at time $t = 50.0$. The Garage will access these attributes at $t = 0.0$ and display the results for uninitialized dynamic attributes (i.e. $-1$, $-1e+20$, and $-1$, respectively).

```
// S_Car_ChangeDynamicBaseTypes.C
#include "SpGlobalFunctions.H"
#include "SpFreeDynAttributes.H"
#include "RB_SpFrameworkFuncs.H"
#include "SpDynObjs.H"
#include "SpList.H"
#include "S_Car.H"

void S_Car::ChangeDynamicBaseTypes() {
  double                    baseTime = SpGetTime();
  double                    endTime;
  double                    stepSize = 100.0;
  int                       currentSpeed = -1;
  DYNAMIC_INT_CONSTANT*     speedItem;
  DYNAMIC_LOGICAL_CONSTANT* ignitionItem;
  SpList                    speedItemsList;
  SpList                    ignitionItemsList;
  /*
   * Remove all of the items on the last whose time is after "now".
   * Save all of these items on a list.  We cannot remove the item
   * from the list immediately since this would invalidate internal
   * the dynamic pointers.  This would cuase a side affect of having
   * to restart the search.  Therefore all items to be removed will
   * be saved.
   */
  speedItem    = (DYNAMIC_INT_CONSTANT *)
    Speed.GetFirstElement();
  ignitionItem = (DYNAMIC_LOGICAL_CONSTANT *)
    IgnitionState.GetFirstElement();
  while (speedItem != NULL) {
    endTime = speedItem->GetEndTime();
    /*
     * Should this item be removed?
     */
    if (endTime > baseTime) {
      /*
       * If the current car velocity has not been locally saved,
       * then save it now.
       */
      if (currentSpeed == -1) {
        /*
         * Data values for dynamic attributes must be passed a time
         * so that the time within the correct interval can be
         * extracted.
         */
        currentSpeed = (*speedItem)(baseTime);
      }
      speedItemsList.Insert(speedItem);
      ignitionItemsList.Insert(ignitionItem);
    }
    speedItem    = (DYNAMIC_INT_CONSTANT *)     ++Speed;
    ignitionItem = (DYNAMIC_LOGICAL_CONSTANT *) ++IgnitionState;
  }
  /*
   * Now remove all the dynamic items off of the dynamic
   * attributes (list).
   */
  speedItem    = (DYNAMIC_INT_CONSTANT *)
    speedItemsList.GetFirstElement();
  ignitionItem = (DYNAMIC_LOGICAL_CONSTANT *)
    ignitionItemsList.GetFirstElement();
  while (speedItem != NULL) {
    Speed         -= speedItem;
    IgnitionState -= ignitionItem;
    RB_FREE_DELETE(FreeDynamicAttributes, speedItem);
    RB_FREE_DELETE(FreeDynamicAttributes, ignitionItem);
    speedItem    = (DYNAMIC_INT_CONSTANT *)
      speedItemsList.GetNextElement();
    ignitionItem = (DYNAMIC_LOGICAL_CONSTANT *)
      ignitionItemsList.GetNextElement();
  }
  /*
   * Now retreive new dynamic items, initializa them and add them
   * to the dynamic attribute.
   */
  while (baseTime <= 3600.0) {
    speedItem = (DYNAMIC_INT_CONSTANT *)
      RB_FREE_NEW(FreeDynamicAttributes, DYNAMIC_INT_CONSTANT_ID);
    speedItem->Set(currentSpeed % 60);
```

```
79      speedItem->SetStartTime(baseTime);
80      speedItem->SetEndTime(baseTime + stepSize);
81      Speed += speedItem;

82      ignitionItem = (DYNAMIC_LOGICAL_CONSTANT *)
83        RB_FREE_NEW(FreeDynamicAttributes, DYNAMIC_INT_CONSTANT_ID);
84      ignitionItem->Set(1);
85      if (currentSpeed % 60 == 0) {
86        ignitionItem->Set(0);
87      }
88      ignitionItem->SetStartTime(baseTime);
89      ignitionItem->SetEndTime(baseTime + stepSize);
90      IgnitionState += ignitionItem;

91      currentSpeed = currentSpeed - 10 - SpGetSimObjKindId();
92      if (currentSpeed < 0) {
93        currentSpeed = currentSpeed + 60;
94      }
95      baseTime = baseTime + stepSize;
96    }
97  }
```

Example 10.22: Car HLA Simulation Object ChangeDynamicBaseTypes Method

Method `ChangeDynamicBaseTypes` will execute at $t = 1750.0$. This method modifies attributes `Speed` and `IgnitionState`. Currently, these attributes contain data starting at time $t = 50.0$ to $t = 3650.0$. This method will remove all of the data from $t = 1800.0$ to $t = 3650.0$ and replace it with new data.

The first `while` loop (line 29) finds all of the dynamic items that are within the removal time interval. Notice that all items that match are saved on an `SpList`. If the dynamic items would have been removed immediately (operator `-=`), then the dynamic attribute list would have been immediately modified. This would have changed the next dynamic item retrieved off of the dynamic attribute list, giving the loop incorrect results. In short, after items are inserted or removed from a dynamic attribute list, then the search must restart at the beginning. With this approach, the items are not immediately deleted, but rather they are saved so that they can be deleted at a later time.

The second `while` loop (line 61) deletes all of the dynamic items previously saved in lists `speedItemList` and `ignitionItemList`. When this is done, there are no dynamic items on the dynamic attributes between the time of $1800.0$ to $3600.0$.

The third `while` loop (line 75) adds new `DYNAMIC_INT_CONSTANT` items in $100.0$ second intervals onto the `Speed` attribute. This time, the speed is decreased in each time interval. For anytime interval for which the speed is equal to $0$, then the `IgnitionState` attribute is set to off ($0$).

The code shown in Example 10.23 shows the Car proxy.

```
// CarProxy.H
#ifndef CarProxy_H
#define CarProxy_H

#include "SpObjProxy.H"

class CarProxy : public SpObjProxy {
  public:
    CarProxy() {
      ModelNameRef      = GetReference("ModelName",      "Car");
      SpeedRef          = GetReference("Speed",          "Car");
      RadioFrequencyRef = GetReference("RadioFrequency", "Car");
      IgnitionStateRef  = GetReference("IgnitionState",  "Car");
    }
    const char*  GetModelName() {return GetString(ModelNameRef);}
    int          GetSpeed(double time) {
      return GetDynamicInt(SpeedRef, time);
    }
    double       GetRadioFrequency(double time) {
      return GetDynamicFloat(RadioFrequencyRef, time);
    }
    int          GetIgnitionState(double time) {
      return GetDynamicLogical(IgnitionStateRef, time);
    }

  private:
    static int ModelNameRef;
    static int SpeedRef;
    static int RadioFrequencyRef;
    static int IgnitionStateRef;
};
#endif
```

Example 10.23: Car Proxy Definition

The proxy for the Car shows how to use methods `GetDynamicInt`, `GetDynamicFloat`, and `GetDynamicLogical` to retrieve data values from the dynamic attributes. As with the static attributes, an integer index is used to find the appropriate data value. However, unlike the static attributes, an additional parameter must be supplied, which is time. The dynamic attribute's proxy is searched for the data which falls within the specified time. The data value for this time is returned to the caller.

The code for the Garage simulation object, `SpFreeObjProxy`, and `main` are not shown here. The code for the Garage is exactly like the code previously shown. The Garage gets its remote proxy list, gets each proxy off the list, type casts the proxy to a Car, and then examines the proxy data. `SpFreeObjProxy` plugs the Car proxy into the proxy framework and main plugs the simulation objects and their events into the SPEEDES framework.

### 10.2.3 One Dimensional Functions

So far, dynamic integers, doubles, and logical have been discussed. The types all have one thing in common, their output values remain constant over their time interval. While this can useful, in reality their behavior can easily be modeled with their static attribute types counterparts.

However, there are more interesting built-in SPEEDES dynamic attributes that can be used to model continuous curves. These attributes allow users to access their data values at any point along their defined curve. The first type of functions explained below are for one-dimensional functions, which are a special type of `DYNAMIC_DOUBLE_ATTRIBUTE`. The last type of dynamic attribute to be discussed is a three-dimensional position vector. This type of attribute is discussed later in Section 10.2.4.

The procedure for creating a `DYNAMIC_DOUBLE_ATTRIBUTE` remains the same as discussed in Section 10.2.2. The dynamic item is pulled off the `FreeDynamicAttributes` free list. However, methods `Set`, `SetStartTime`, and `SetEndTime` are replaced with methods which are specific for each different type of function. These functions are explained in more detail in the following sections.

#### 10.2.3.1 Polynomial Items

The most basic dynamic item types are the polynomial types used to fit curves over a time interval. The built-in polynomial dynamic items range from first degree polynomial (i.e. linear), to a tenth degree polynomial. All these items inherit from the `DYNAMIC_POLY_N` class. The essential idea is that a $N^{th}$ degree polynomial can be determined from a list of $N + 1$ or more points. If there are $N + 1$ points, the polynomial is uniquely determined (e.g. two points determine a first degree polynomial, three points determine a second degree polynomial, etc). If there are more than $N+1$ points, a "best-fit" $N^{th}$ degree polynomial is calculated.

As with all dynamic attribute items, dynamic polynomials are retrieved from the dynamic attribute free list. Once a dynamic attribute item is retrieved, it is initialized with method `AddPoints`. The API for `AddPoints` is:

```
void AddPoints(double startTime,
               double endtime,
               double time,
               double x,
               double variance,
               ...
               )
```

| Parameter | Description |
|---|---|
| startTime | Interval start time. |
| endTime | Interval end time. |
| time | Time for point x. |
| x | Magnitude of data point at $t = time$. |
| variance | Sigma squared for that position must be greater than 0. The variance has to do with curve fitting, so it is ignored in all uniquely determined cases (i.e. $N + 1$ points provided for an $N^{th}$ degree polynomial), except that it must not be zero or negative. |
| ... | Additional $(time, x, variance)$ triplets. |

Table 10.3: Polynomial AddPoint Initialization Method

Method `AddPoints` is always terminated with `END_POLY`. This indicates that no additional data is to be input. The number of $(time, x, variance)$ triplets used in method `AddPoints` is one more that the order of the polynomial in use (i.e. N + 1). For example, `DYNAMIC_POLY_1` takes two triplets, `DYNAMIC_POLY_2` takes three triples, etc. After the polynomial item has been initialized with its input parameters, method `MakePoly` is called on the polynomial item which calculates the best fit line to the input data.

Consider the case of a uniquely determined polynomial. The simplest of these is the first degree polynomial, $f(t) = mt + b$, which is uniquely specified by two points. The `DYNAMIC_POLY_1` class is the type of dynamic item provided for linear equations. The basic procedure is to first load two points into the `DYNAMIC_POLY_1` object, and then have it construct a first-degree polynomial. Once this has been done, the linear function can be evaluated at any point within the interval associated with the item.

Example 10.24 shows an an example of how to use `DYNAMIC_POLY_1`. Attribute `Temperature`, whose type is `DYNAMIC_DOUBLE_ATTRIBUTE`, has been added to the Car simulation object with its appropriate `Objects.par` file entry of `Temperature` defined as a `dynamic_double`. The temperature of the Car will start at 10 degrees at $t = 0.0$ and will increase linearly to 20 degrees at $t = 1800.0$. Then, at time $t = 1800.0$, the car starts to warm up faster in a linear fashion until it reaches 100 degrees at time $t = 3600.0$. This will require that two `DYNAMIC_POLY_1` items be added to attribute `Temperature`.

```
// S_Car_InitPolynomial.C
#include "SpFreeDynAttributes.H"
#include "RB_SpFrameworkFuncs.H"
#include "SpDynObjs.H"
#include "S_Car.H"
#include "CarProxy.H"

int CarProxy::TemperatureRef;

void S_Car::InitPolynomial() {
  DYNAMIC_DOUBLE_CONSTANT* doubleItem;
  DYNAMIC_POLY_1*          polyItem;

  DEFINE_ATTRIBUTE(Temperature,  "Temperature");

  /*
   * Define an equation for temperature. The equation will be a simple
   * equation in the form of f(t) = mt + b.  Add two points for ranges
   * between 0.0 to 1800.0 seconds and 1800.0 to 3600.0.  Since end
   * points are undefined user can add a constant double at the end
   * points to that valid data can be retrieved at these points (i.e.
   * 0.0 and 3600.0).  The two equations are:
   * f1(t) = 1/180(t) + 10
   * f2(t) = 2/45(t)  + 60
   * At 1800 f1(t) = f2(t).  This corresponds to a straight line being
   * drawn between points (0.0, 10) to (1800.0 20) and (1800.0, 20) to
   * (3600.0, 100).  First add the constant double.
   */
  doubleItem  = (DYNAMIC_DOUBLE_CONSTANT *)
    RB_FREE_NEW(FreeDynamicAttributes, DYNAMIC_DOUBLE_CONSTANT_ID);
  doubleItem->Set(10.0);
  doubleItem->SetStartTime(-10.0);
  doubleItem->SetEndTime(0.0);
  Temperature += doubleItem;
  /*
   * Add f1(t) = 1/180t + 10
   */
  polyItem = (DYNAMIC_POLY_1 *)
    RB_FREE_NEW(FreeDynamicAttributes, DYNAMIC_POLY_1_ID);
  polyItem->AddPoints(0.0,    1800.0,
                      0.0,    10.0, 1.0,
                      1800.0, 20.0, 1.0,
                      END_POLY);
  polyItem->MakePoly();
  Temperature += polyItem;
  /*
   * Add f2(t) = 1/45t -60
   */
  polyItem = (DYNAMIC_POLY_1 *)
    RB_FREE_NEW(FreeDynamicAttributes, DYNAMIC_POLY_1_ID);
  polyItem->AddPoints(1800.0, 3600.0,
                      1800.0, 20.0,  1.0,
49                         3600.0, 100.0, 1.0,
50                         END_POLY);
51    polyItem->MakePoly();
52    Temperature += polyItem;
53    /*
54     * Add a constant 100 after time 3600.0
55     */
56    doubleItem  = (DYNAMIC_DOUBLE_CONSTANT *)
57      RB_FREE_NEW(FreeDynamicAttributes, DYNAMIC_DOUBLE_CONSTANT_ID);
58    doubleItem->Set(100.0);
59    doubleItem->SetStartTime(3600.0);
60    doubleItem->SetEndTime(3700.0);
61    Temperature += doubleItem;
62  }
```

Example 10.24: Car HLA Simulation Object InitPolynomial method

Lines 25 through 41 initialize the function for the temperature for the time interval of $0.0 < t < 1800.0$. Notice, that lines 25 through 30 are actually initializing a `DYNAMIC_DOUBLE_CONSTANT` and adding this to the `Temperature` attribute. Why do this? The valid range for the attribute will be from $0.0 < t < 3600.0$. Therefore, the `Temperature` attribute is undefined for $t < 0.0$ causing a discontinuity at $t = 0.0$. By adding `DYNAMIC_DOUBLE_CONSTANT` to the dynamic attribute, the discontinuity is removed since the dynamic items for both intervals contain the same value at $t = 0.0$. Lines 25 through 30 defines a temperature value of 10 from $-10.0 < t < 0.0$. Lines 34 through 41 initialize the `DYNAMIC_POLY_1` item and adds the item the attribute `Temperature`. Since both of these items have the same magnitude at $t = 0.0$, the discontinuity has been removed and queries at $t = 0.0$ can be performed.

Lines 45 through 61 initialize the function for the temperature for the time interval $1800.0 < t < 3600.0$. Since both this dynamic polynomial item and the previous interval dynamic polynomial item have the same value at $t = 1800.0$, there is not discontinuity at this data point. Also, the discontinuity at $t = 3600.0$ has been eliminated by adding the dynamic double constant item.

#### 10.2.3.2 Overdetermined Polynomial Items

What happens when a polynomial item of degree $N$ is given more than $N + 1$ data points? They will not necessarily all lie on the same curve, so the item must attempt to find the curve that "best" fits the data. The typical way of doing this is to find the "least square" $N^{th}$ degree polynomial. This minimizes the sum of the squared distance of each point from the curve.

The polynomial items have generalized this concept, so they can do a normal least square fit and more. As shown above, the `AddPoints` method accepts a list of $(time, x, variance)$ triplets (among other items). The $(time, x)$ pair specifies a point, and the $variance$ tells `MakePoly` how far to allow the polynomial to deviate from that point; the smaller the variance, the smaller the deviation of the polynomial from the point (note that the variance must be greater than 0). Thus, `MakePoly` determines the polynomial that minimizes the sum of the weighted squared distance of each point from the curve (where points whose variance is smaller are more heavily weighted). If each point has the same variance (e.g. all are 1.0), then this scheme reduces to an ordinary least squares fit.

#### 10.2.3.3 Exponential Items

This attribute defines equations of the form $f(t) = ae^{(L(t-t_0))}$, where $a$ is the amplitude parameter, $e$ is the transcendental real 2.71828..., $L$ is the parameter usually referred to as "lambda", and $t_0$ is the phase shift parameter, which equals the starting time of the interval associated with this attribute. Thus, by assigning the parameters $a$, $L$, and $t_0$, the exponential equation will be fully defined and then can be evaluated for arbitrary values of $t$. Since this attribute inherits from `SpDynItem`, its interval can be set using methods `SetStartTime` and `SetEndTime`. Since $t_0$ refers to the starting time, when you call `SetStartTime(x)`, $t_0$ will be assigned the value `x`. Use `SetAmplitude` to specify the $a$ parameter (and `GetAmplitude` to probe it). Use the `SetLambda` and `GetLambda` methods for the lambda parameter. Alternately, there is a `Set` method for assigning both the amplitude and lambda at once.

Once the parameters and the intervals have been set, the parentheses operator can be used to evaluate the function at time $t$. Also, the parentheses operator has been overloaded to accept two or three arguments. In the first case, the second argument is an `out` parameter that returns the first derivative of the exponential function, evaluated at the given time (always the first argument). In the second case, there are two `out` parameters, one that returns the first derivative, and one that returns the second derivative. Suppose a `DYNAMIC_EXPONENTIAL` pointer called `dynExp` had been retrieved from `SpFreeDynAttributes`. The following code shows an example of usage:

```
dynExp->SetStartTime(10.0);    // Sets t0 and interval starting time
dynExp->SetEndTime(50.0);      // Interval end time
dynExp->SetAmplitude(-3.5);    // Set amplitude parameter
dynExp->SetLambda(1.25);       // Set lambda parameter

double time = 12.22;
double x    = (*dynExp)(time); // Evaluate f(12.22)

double deriv1;                 // First derivative
double deriv2;                 // Second derivatives

time = 25.0;
// Returns f(25.0) and f'(25.0):
double y = (*dynExp)(time, deriv1);

time = 47.7;
// Returns f(47.7), f'(47.7), and f''(47.7):
double z = (*dynExp)(time, deriv1, deriv2);
```

#### 10.2.3.4 Other Dynamic Attribute Items

There are four other dynamic items that can be inserted into a `DYNAMIC_DOUBLE_ATTRIBUTE`. These items are `DYNAMIC_COMPLEX_EXPONENTIAL`, `DYNAMIC_EXTRAPOLATE`, `DYNAMIC_SPLINE_3`, and the `DYNAMIC_SPLINE_6`.

The `DYNAMIC_COMPLEX_EXPONENTIAL` item specifies the following equation over a given interval: $f(t) = ae^{(i\Omega(t-t_0)}$. In this equation, $a$ is the amplitude, $e$ is the transcendental real number 2.71828..., $i = \sqrt{-1}$, $\Omega$ is a parameter, and $t_0$ is the interval starting time. See the SPEEDES API Reference Manual for the details of how to use this item.

The `DYNAMIC_SPLINE_3` item creates a cubic polynomial for the given interval. Given the initial position and velocity, and the ending position and velocity, the cubic polynomial is uniquely determined over the entire interval. This polynomial is also known as a Hermite interpolant. See the SPEEDES API Reference Manual for the details of how to use this item.

The DYNAMIC_SPLINE_6 item creates a fifth-degree polynomial for the given interval. Given the initial position, velocity, and acceleration, and the ending position, velocity, and acceleration, a fifth-degree polynomial is uniquely determined over the entire interval. See the SPEEDES API Reference Manual for the details of how to use this item.

The DYNAMIC_EXTRAPOLATE item performs extrapolation of the position based on an initial position and velocity. Additionally, the initial acceleration and jerk (third derivative) can be specified as well. See the SPEEDES API Reference Manual for the details of how to use this item.

### 10.2.4 Dynamic Position Attributes

The DYNAMIC_POSITION_ATTRIBUTE specifies position as a function of time (equation of motion). As long as the object sticks to this scripted motion plan, subscribers can use this attribute to calculate the object's position, velocity, and acceleration for anytime within the interval. This reduces message traffic and also helps to spread the computational load across the number of execution nodes. If the object deviates from its motion plan, subscribers receive updates that provide new equations of motion.

The dynamic position attribute works similarly to DYNAMIC_DOUBLE_ATTRIBUTE: dynamic items are inserted into the attribute; each dynamic item specifies the equation of motion that applies over that interval. There are several different types of dynamic items to choose from, each of which defines a different motion type. Table 10.4 shows the built-in dynamic position items available:

| **Item** | **Description** |
|---|---|
| POLY_N_MOTION | Defines an $N^{th}$ order polynomial as a weighted least squares fit to at least $N+1$ points where $N = 1, 2, \ldots, 10$. |
| GREAT_CIRCLE | The shortest route between two points on the earth. |
| RHUMB_LINE | The shortest route between two points on the earth, proceeding with a fixed bearing. |
| CIRCULAR_ORBIT | Defines a circular orbit around the earth. |
| Elliptical | Defines an elliptical orbit around the earth. |
| EXTRAPOLATE_MOTION | Extrapolates the current position, velocity, acceleration, and jerk (set the latter two to zero if unknown) to find future positions. |
| LOITER_MOTION | Defines a constant altitude circular motion about a fixed point on the surface of the earth. |
| SPLINE3_MOTION | Defines a third order polynomial based on known initial and final position and velocity. |
| SPLINE6_MOTION | Defines a fifth order polynomial based on known initial and final position, velocity, and acceleration. |
| CONSTANT_MOTION | A fixed position over the interval. |

Table 10.4: Dynamic Positions Items

These dynamic items are similar to the POSITION_ATTRIBUTE in that there are three different coordinate systems that can be used: EARTH, ECI, or ECR. See Section 10.1.8 for descriptions of these coordinate systems. Refer to the SPEEDES API Reference Manual for the exact procedure on how to initialize each dynamic item. Item POLY_N_MOTION is similar to items DYNAMIC_POLY_N in that method AddPoints is used to initialize the dynamic item data points. Since POLY_N_MOTION items represent a three-dimensional position, each data point is represented by $(time, x, y, z, variance)$, as opposed to $(time, x, variance)$ in the one-dimensional case.

For a final example, a dynamic position has been added to the Car simulation. Figure 10.5 shows the new Objects.par file. The dynamic double added in the example described in Section 10.2.3.1 is also shown.

```
Car {
  define string           ModelName
  define dynamic_int      Speed
  define dynamic_double   RadioFrequency
  define dynamic_logical  IgnitionState
  define dynamic_double   Temperature
  define dynamic_position Position
}

Garage {
  reference SUBSCRIBE Car
}
```

Figure 10.5: Objects.par File for Dynamic Attribute Example

This `Objects.par` file shows the usage of `dynamic_double` and `dynamic_position`. Examples 10.25 and 10.26 show the Car definition file and the initialization for the dynamic position attribute.

```
// S_Car.H
#ifndef S_Car_H
#define S_Car_H

#include "S_SpHLA.H"
#include "SpDefineSimObj.H"
#include "SpDefineEvent.H"

class S_Car : public S_SpHLA {
  public:
    S_Car(char* objClassName = "Car") : S_SpHLA(objClassName) {}

    void   Init() {
        InitDynamicBaseTypes();
        InitPolynomial();
        InitPosition();
    }
    void   InitDynamicBaseTypes();
    void   InitPolynomial();
    void   InitPosition();
    void   ChangeDynamicBaseTypes();

  private:
    STRING_ATTRIBUTE           ModelName;
    DYNAMIC_INT_ATTRIBUTE      Speed;
    DYNAMIC_DOUBLE_ATTRIBUTE   RadioFrequency;
    DYNAMIC_LOGICAL_ATTRIBUTE  IgnitionState;
    DYNAMIC_DOUBLE_ATTRIBUTE   Temperature;
    DYNAMIC_POSITION_ATTRIBUTE Position;
};
DEFINE_SIMOBJ(S_Car, 2, SCATTER);
DEFINE_SIMOBJ_EVENT_0_ARG(Car_ChangeDynamicBaseTypes,
                          S_Car, ChangeDynamicBaseTypes);
#endif
```

Example 10.25: Car HLA Simulation Object Definition

```
// S_Car_InitPolynomial.C
#include "SpFreeDynAttributes.H"
#include "RB_SpFrameworkFuncs.H"
#include "SpGreatCircle.H"
#include "S_Car.H"
#include "CarProxy.H"

int CarProxy::PositionRef;

void S_Car::InitPosition() {

  DEFINE_ATTRIBUTE(Position,  "Position");
  /*
   * The DYNAMIC_POSITION_ATTRIBUTE will be defined to contain two
   * GREAT_CIRCLE item intervals.  The first GREAT_CIRCLE item will be
   * defined using the constant known velocity method.
   */
  GREAT_CIRCLE* greatCircleItem;
  double        latStart  = 10.0;
  double        lonStart  = 50.0;
  double        altStart  = 5.0;
  double        latEnd    = 5.0;
  double        lonEnd    = 25.0;
  double        altEnd    = 2.0;
  double        startTime = 0.0;
  double        endTime;
  double        kmPerSec  = 5.0;

  greatCircleItem = (GREAT_CIRCLE *)
    RB_FREE_NEW(FreeDynamicAttributes, GREAT_CIRCLE_ID);
  endTime = greatCircleItem->init_Vconstant(
    latStart, lonStart, altStart, startTime,
    latEnd, lonEnd, altEnd, kmPerSec);
  Position += greatCircleItem;
  /*
   * The next GREAT_CIRCLE item will be defined using the constant fixed
   * start and end point method.  First, set the start points of the new
   * position item to the end points of the last item.
   */
  latStart  = latEnd;
  lonStart  = lonEnd;
  altStart  = altEnd;
  latEnd    = 89.0;
  lonEnd    = 45.0;
  altEnd    = 0.0;

  greatCircleItem = (GREAT_CIRCLE *)
    RB_FREE_NEW(FreeDynamicAttributes, GREAT_CIRCLE_ID);
  kmPerSec = greatCircleItem->init_Tconstant(
    latStart, lonStart, altStart, endTime,
    latEnd, lonEnd, altEnd, 3600.0);
  Position += greatCircleItem;
}
```

Example 10.26: Car HLA Simulation Object InitPosition Method

The code in Example 10.26 shows an example of how to initialize a dynamic position attribute. Initialization of a dynamic position attribute is very similar to the dynamic attribute initialization shown in Example 10.24. This example is initializing a GREAT_CIRCLE item. Two items are retrieved from the dynamic attribute free list. The two items are initialized by using the constant velocity method and by using the start and end point method. They are initialized such that there is no discontinuity at the point where the intervals cross over from the first item to the second item.

Example 10.27 shows the new proxy for this example.

```
// CarProxy.H
#ifndef CarProxy_H
#define CarProxy_H

#include "SpObjProxy.H"

class CarProxy : public SpObjProxy {
  public:
    CarProxy() {
      ModelNameRef      = GetReference("ModelName",      "Car");
      SpeedRef          = GetReference("Speed",          "Car");
      RadioFrequencyRef = GetReference("RadioFrequency", "Car");
      IgnitionStateRef  = GetReference("IgnitionState",  "Car");
      TemperatureRef    = GetReference("Temperature",    "Car");
      PositionRef       = GetReference("Position",       "Car");
    }
    const char*  GetModelName() {return GetString(ModelNameRef);}
    int          GetSpeed(double time) {
      return GetDynamicInt(SpeedRef, time);
    }
    double       GetRadioFrequency(double time) {
      return GetDynamicFloat(RadioFrequencyRef, time);
    }
    int          GetIgnitionState(double time) {
      return GetDynamicLogical(IgnitionStateRef, time);
    }
    double       GetTemperature(double time) {
      return GetDynamicFloat(TemperatureRef, time);
    }

    void         GetEARTHPosition(double time,
                                  double pos[3],
                                  double vel[3],
                                  double acc[3]) {
      GetDynamicPosition(PositionRef, pos, vel, acc, EARTH, time);
    }


  private:
    static int ModelNameRef;
    static int SpeedRef;
    static int RadioFrequencyRef;
    static int IgnitionStateRef;
    static int TemperatureRef;
    static int PositionRef;
};
```

```
#endif
```

Example 10.27: Car HLA Simulation Object InitPosition Method

The same technique used to retrieve the data stored on a dynamic attribute is used to extract the data dynamic position attribute. The integer reference for the attribute is first calculated. Next, method `GetEARTHPosition` has been defined on the Car proxy. This calls method `GetDynamicPosition`. Separate methods for `ECI` and `ECR` could have been defined or the `Get` method could have been designed which takes `EARTH`, `ECI`, and `ECR` as input parameters.

The code shown in Example 10.28 shows the new Garage implementation.

```
// S_Garage.C
#include "F_SpProxyItem.H"
#include "S_Garage.H"
#include "CarProxy.H"

void S_Garage::Init() {
  SCHEDULE_Garage_Display(0.0, SpGetObjHandle());
}

void S_Garage::Display() {
  int       i;
  double    pos[3];
  double    vel[3];
  double    acc[3];
  RB_queue* remoteProxyList;  // Car Proxies in this case.

  RB_cout << SpGetTime() << endl;
  remoteProxyList = GetRemoteObjectProxies();
  F_SpProxyItem* proxyItem = (F_SpProxyItem *) remoteProxyList->get_top();
  for (i = 0; i < remoteProxyList->get_length(); ++i){
    CarProxy*    carProxy = (CarProxy *) proxyItem->GetObjProxy();
    carProxy->GetEARTHPosition(SpGetTime(), pos, vel, acc);
    RB_cout << carProxy->GetModelName() << endl
      << "Car:Speed=        " << carProxy->GetSpeed(SpGetTime()) << endl
      << "Car:Radio Freq=   " << carProxy->GetRadioFrequency(SpGetTime())
      << endl
      << "Car:Ignition=     " << carProxy->GetIgnitionState(SpGetTime())
      << endl
      << "Car:Temperature=  " << carProxy->GetTemperature(SpGetTime())
      << endl
      << "Car:Pos=          " << pos[0] << " " << pos[1] << " " << pos[2]
      << endl
      << endl;
    proxyItem = (F_SpProxyItem *) proxyItem->get_link();
  }
  RB_cout << endl;
  SCHEDULE_Garage_Display(SpGetTime() + 300.0, SpGetObjHandle());
}
```

Example 10.28: Garage HLA Simulation Object Implementation

The Garage used the same techniques previously discussed to get the proxies on its remote proxy list and examining the contents of the proxy.

As with the `DYNAMIC_DOUBLE_ATTRIBUTE`, different dynamic item types can be used within the same `DYNAMIC_POSITION_ATTRIBUTE`. See the SPEEDES API Reference Manual for the details of how to use these and other dynamic motion types.

## 10.3 Tips, Tricks, and Potholes

- All of the examples in this Chapter used integer ids to look up each attribute on the proxy. Integer ids provide the fastest mechanism when doing these look ups. However for each proxy `Get` method the string name for the attribute could have been used as well. See the SPEEDES API Reference Manual for additional details.

Chapter 11

# Data Distribution Management (DDM)

The object proxy code provides a powerful method for distributing portions of simulation objects to other nodes in a time-ordered fashion. The previous two sections discussed object proxies and their delivery, which only uses the DM portion of the SPEEDES framework. The drawback to DM is that any required filtering must be done by the receiving object (e.g. a sensor sees all objects, not just objects within its sensor range). All publications of an object are received by the subscribing simulation object when only DM is used. In order to improve performance, it would be preferable if only the proxies of simulation objects within a valid or correct range would be delivered to the subscribing simulation objects. The SPEEDES framework provides a built-in capability that allows users to specify the valid ranges for different attributes of publishing objects and, if the attributes are within this range, the subscribing object receives the publishing object's proxy. This level of filtering is called DDM.

This chapter will introduce the various types of DDM provided in the SPEEDES framework. First, we will present a simple example of not using DDM and then, we will take this example and enhance it with differing levels of DDM. We will first examine dynamic class type filter specification rather than static class type filters that are offered by DM. Next, we will present methods for filtering on enumerated types, ranges of doubles, and range-based filtering.

## 11.1 Declaration Management Simulation Example

Before diving into the DDM specifics of the SPEEDES framework, let us examine a simulation that uses only DM for filtering. When using DM, a simulation will receive all proxies for any simulation object for which it has subscribed to. Each receiving simulation object will then have to examine each received proxy to determine if the data on that proxy is within its desired range (e.g. distance, frequency, velocity, etc.).

The code shown in Examples 11.1 through 11.5 contains four simulation objects consisting of three ships and one submarine. The simulation is set up on a rectangular 50 km by 50 km grid. Ship 0 will be moving from right to left starting at $(40, 0)$ at $t = 0.0$ and ending at the location $(-60, 0)$ at $t = 3600.0$ seconds. Ship 1 will be moving from top to bottom starting at $(0, 50)$ at $t = 0.0$ and ending at the location $(0, -50)$ at $t = 3600.0$ seconds. Ship 2 will be moving from upper left to bottom right starting at $(-40, 40)$ at $t = 0.0$ and ending at the location $(50, -50)$ at $t = 3600.0$ seconds. The submarine will be stationary at $(-10, -6)$. Ships 0, 1, and 2 will have an active radar with detection ranges of 10, 15, and 0 km, respectively. The submarine will have a passive detection device that can detect other ships at a range of 20 km.

Figure 11.1 shows the current positions of the simulation objects at $t = 720.0$. The circles shown in the figure are the detection ranges for each simulation object. Since all proxies are delivered to all simulation objects, each simulation object will have to examine each proxy to determine if the received proxy is within range of its sensor. If it is within range, then it can detect the object, otherwise it cannot. This is a polling technique, which in our case, starts at $t = 0.0$ and goes to $t = 3600.0$. Each simulation object has an event that will execute every 10.0 seconds examining its received proxies. This event will print out the distance to the other ships (i.e range, country and radar frequency), if the other ships are within range of its sensors (range-based filtering). In addition, the submarine will print out the frequency of the other ship's radars and their country of origin. Each event must examine all three proxies on its list in order to determine if each object is within range. Many of these calculations are done for objects that are out of range. Hence, wasted effort was expended during the simulation. Use of DDM can reduce the number of proxies on the object's proxy list, thereby reducing the amount of unnecessary calculation, thus improving simulation run-time performance.

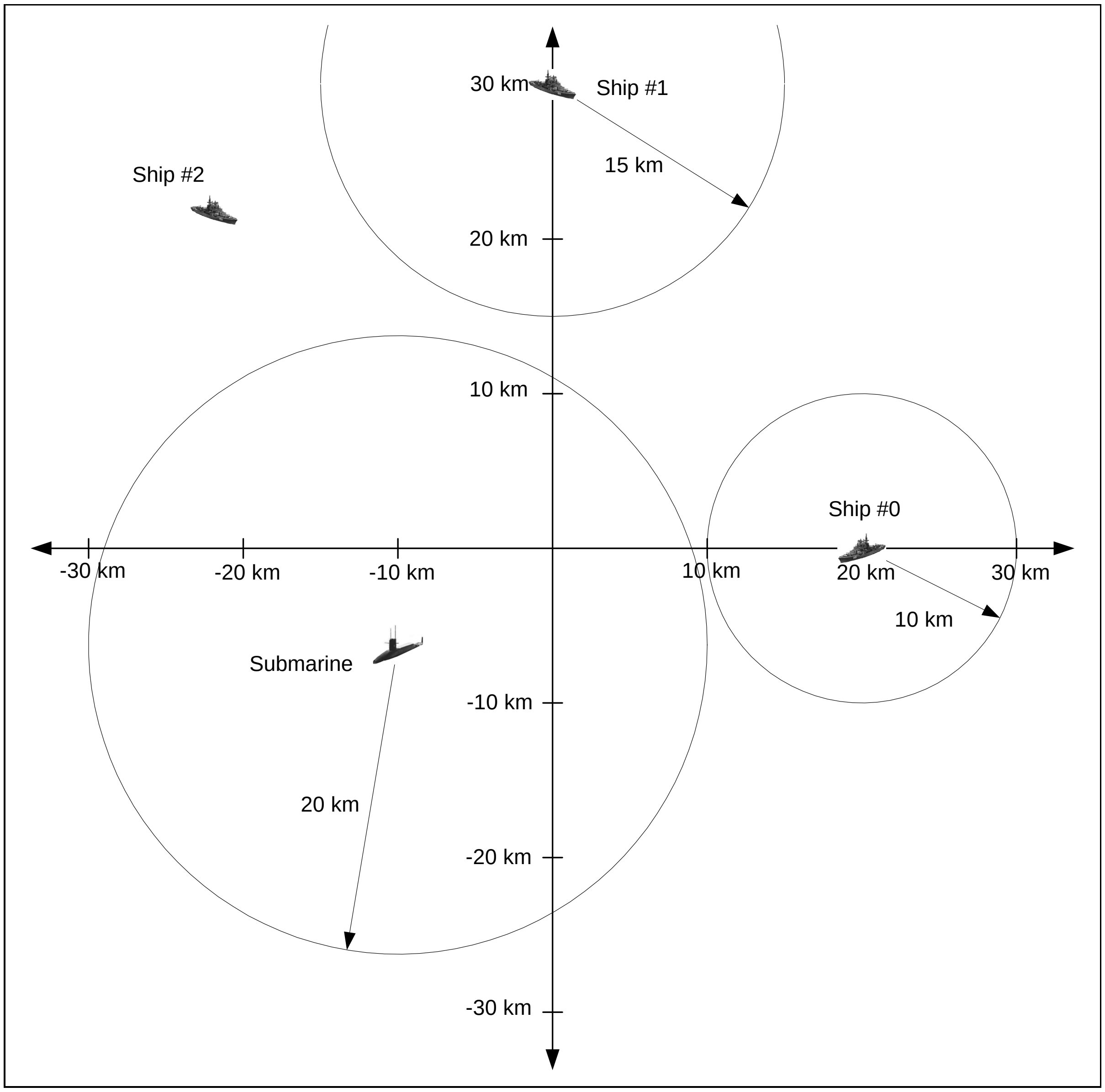


Figure 11.1: DM only Ship Simulation Layout

Examples 11.1 and 11.2 show the definition and implementation code for the Ship simulation object. The simulation object's radar frequency, country, position, and radar search event (i.e. `ShipSensorSearch`) are initialized in method `S_Ship::Init`. Event `ShipSensorSearch` executes every 10.0 seconds, examining every proxy which has been delivered to it. If it finds an object within range, then it outputs data to the terminal.

```
// S_Ship.H
#ifndef S_SHIP_H
#define S_SHIP_H

#include "S_SpHLA.H"
#include "SpDefineSimObj.H"
#include "SpDefineEvent.H"

class S_Ship : public S_SpHLA {
  public:
    S_Ship() : S_SpHLA("Ship") {}
    ~S_Ship() {}

    void   Init();
    void   SensorSearch();
    void   ChangeRadarFrequency();

  private:
    DOUBLE_ATTRIBUTE           RadarFrequency;  // Gigahertz
    INT_ATTRIBUTE              Country;         // Country
    DYNAMIC_POSITION_ATTRIBUTE Position;        // Ship Position
    double                     RadarRange;      // Kilometers

    void   ProcessShipProxy(SpObjProxy* proxy);
    void   ProcessSubmarineProxy(SpObjProxy* proxy);
    void   PrintDistance(SpObjProxy* proxy, double difference[3]);
};

DEFINE_SIMOBJ(S_Ship, 3, SCATTER);
DEFINE_SIMOBJ_EVENT_0_ARG(ShipSensorSearch, S_Ship, SensorSearch);
DEFINE_SIMOBJ_EVENT_0_ARG(ShipChangeRadarFrequency, S_Ship,
                          ChangeRadarFrequency);
#endif
```

Example 11.1: DM Ship Simulation Object Definition

```
// S_Ship.C
#include "SpGlobalFunctions.H"
#include "SpSchedule.H"
#include "RB_ostream.H"
#include "SpObjProxy.H"
#include "SpPolyMotion.H"
#include "SpConstantPosition.H"
#include "SpFreeDynAttributes.H"
#include "F_SpProxyItem.H"
#include "S_Ship.H"

void S_Ship::Init() {
  DEFINE_ATTRIBUTE(RadarFrequency,  "RadarFrequency");
  DEFINE_ATTRIBUTE(Country,         "Country");
  DEFINE_ATTRIBUTE(Position,        "Position");

  POLY_1_MOTION* line = (POLY_1_MOTION *)
     FreeDynamicAttributes->new_object(POLY_1_MOTION_ID);

  line->SetECR();
  if (SpGetSimObjKindId() == 0) {
    line->AddPoints(0.0,    3600.0,
                    0.0,    6400.0,  40.0, 0.0, 1.0,
                    3600.0, 6400.0, -60.0, 0.0, 1.0,
                    END_POLY_MOTION);
    Country        = 0;
    RadarFrequency = 5;
    RadarRange     = 10;
  }
  else {
    if (SpGetSimObjKindId() == 1) {
      line->AddPoints(0.0,    3600.0,
                      0.0,    6400.0, 0.0,  50.0, 1.0,
                      3600.0, 6400.0, 0.0, -50.0, 1.0,
                      END_POLY_MOTION);
      Country        = 1;
      RadarFrequency = 8;
      RadarRange     = 15;
    }
    else {
      line->AddPoints(0.0,    3600.0,
                      0.0,    6400.0, -40.0,  40.0, 1.0,
                      3600.0, 6400.0,  50.0, -50.0, 1.0,
                      END_POLY_MOTION);
      Country        = 2;
      RadarFrequency = 11;
      RadarRange     = 0;
    }
  }
  line->MakePoly();
  Position += line;

  SCHEDULE_ShipSensorSearch(5.0, SpGetObjHandle());
  SCHEDULE_ShipChangeRadarFrequency(10.0, SpGetObjHandle());
}

void S_Ship::SensorSearch() {
  int i;
  static int shipMgrId         = SpGetSimObjMgrId("S_Ship_MGR");
  static int submarineMgrId    = SpGetSimObjMgrId("S_Submarine_MGR");
  F_SpProxyItem* testProxyItem = (F_SpProxyItem *)
    GetRemoteObjectProxies()->get_top();

  for (i = 0; i < GetRemoteObjectProxies()->get_length(); ++i) {
    SpObjProxy* proxy = testProxyItem->GetObjProxy();

    if (proxy->GetProxySimObjMgrId() == shipMgrId) {
      ProcessShipProxy(proxy);
    }
    if (proxy->GetProxySimObjMgrId() == submarineMgrId) {
      ProcessSubmarineProxy(proxy);
    }
    testProxyItem = (F_SpProxyItem *) testProxyItem->get_link();
  }
  SCHEDULE_ShipSensorSearch(SpGetTime() + 10.0, SpGetObjHandle());
}

void S_Ship::ChangeRadarFrequency() {
  RadarFrequency = RadarFrequency + 0.5;
  if (RadarFrequency > 5.0 + 3 * SpGetSimObjKindId() + 1.0) {
    RadarFrequency = 5.0 + 3 * SpGetSimObjKindId() - 1.0;
  }
  SCHEDULE_ShipChangeRadarFrequency(SpGetTime() + 10.0, SpGetObjHandle());
}

void S_Ship::ProcessShipProxy(SpObjProxy* proxy) {
  double     myECR[3];
  double     remoteECR[3];
  double     difference[3];
  static int refPosition =
    proxy->GetReference("Position", "Ship");

  Position(SpGetTime(), ECR, &myECR[0]);
  proxy->GetDynamicPosition(refPosition, remoteECR,
                            ECR,         SpGetTime());
  Difference(myECR, remoteECR, difference);
  PrintDistance(proxy, difference);
}

void S_Ship::ProcessSubmarineProxy(SpObjProxy* proxy){
  double     MyECR[3];
  double     RemoteECR[3];
  double     difference[3];
  static int refPosition =
    proxy->GetReference("Position", "Submarine");

  Position(SpGetTime(), ECR, &MyECR[0]);
  proxy->GetPosition(refPosition, RemoteECR,
                     ECR,         SpGetTime());
  Difference(MyECR, RemoteECR, difference);
  PrintDistance(proxy, difference);
}

void S_Ship::PrintDistance(SpObjProxy* proxy, double difference[3]) {
  double euclidDistance = Magnitude(difference);
  if (euclidDistance < RadarRange) {
     RB_cout << "Ship "     << SpGetSimObjKindId()
             << " is "      << euclidDistance
             << " km from " << proxy->GetProxyName()
             << " at time " << (double) SpGetTime()
             << " seconds." << endl;
  }
}
```

Example 11.2: DM Ship Simulation Object Implementation

Examples 11.3 and 11.4 show the definition and implementation code for the Submarine simulation object. The submarine has a receiver that can detect radar signals. The ship receives all ship proxies and examines each proxy to determine if it is within its radar detector's range, emitting radar frequency, and country of origin.

```
// S_Submarine.H
#ifndef S_Submarine_H
#define S_Submarine_H

#include "S_SpHLA.H"
#include "SpDefineSimObj.H"
#include "SpDefineEvent.H"

class S_Submarine : public S_SpHLA {
  public:
    S_Submarine() : S_SpHLA("Submarine") {}
    ~S_Submarine() {}

    void   Init();
    void   SensorSearch();

  private:
    INT_ATTRIBUTE              Country;         // Country
    POSITION_ATTRIBUTE         Position;        // Submarine position
};

DEFINE_SIMOBJ(S_Submarine, 1, SCATTER);
DEFINE_SIMOBJ_EVENT_0_ARG(SubSensorSearch, S_Submarine, SensorSearch);
#endif
```

Example 11.3: DM Submarine Simulation Object Definition

```
// S_Submarine.C
#include "SpGlobalFunctions.H"
#include "SpSchedule.H"
#include "RB_ostream.H"
#include "SpObjProxy.H"
#include "SpExportAttribute.H"
#include "F_SpProxyItem.H"
#include "S_Submarine.H"

void S_Submarine::Init() {
  DEFINE_ATTRIBUTE(Country,   "Country");
  DEFINE_ATTRIBUTE(Position,  "Position");

  Country = 2;
  double pos[3] = {6400.0, -10.0, -6.0};
  Position.SetECR(pos);
  SCHEDULE_SubSensorSearch(0.0, SpGetObjHandle());
}

void S_Submarine::SensorSearch() {
  int            i;
  double         MyECR[3];
  double         RemoteECR[3];
  double         difference[3];

  static int     shipMgrId     = SpGetSimObjMgrId("S_Ship_MGR");
  F_SpProxyItem* testProxyItem = (F_SpProxyItem *)
    GetRemoteObjectProxies()->get_top();

  for (i = 0; i < GetRemoteObjectProxies()->get_length(); ++i) {
    SpObjProxy* proxy             = testProxyItem->GetObjProxy();
    static int  refCountry        =
      proxy->GetReference("Country",        "Ship");
    static int  refRadarFrequency =
      proxy->GetReference("RadarFrequency", "Ship");
    static int  refPosition       =
      proxy->GetReference("Position",       "Ship");

    int    targetCountry          = proxy->GetInt(refCountry);
    double targetFrequency        = proxy->GetFloat(refRadarFrequency);
    /*
     * Print Ship’s Country and Radar Frequency
     */
    RB_cout << "Submarine "                     << SpGetSimObjKindId()
            << " detects "                      << proxy->GetProxyName()
            << " from country "                 << targetCountry
            << " emitting radar frequency at " << targetFrequency
            << " GHz at time "                  << (double) SpGetTime()
            << endl;
    /*
     * Range base filtering
     */
    Position.GetECR(&MyECR[0], SpGetTime());
    proxy->GetDynamicPosition(refPosition, RemoteECR,
                              ECR,         SpGetTime());
    Difference(MyECR, RemoteECR, difference);
    double euclidDistance = Magnitude(difference);
    if (20.0 > euclidDistance) {
      RB_cout << "Submarine " << SpGetSimObjKindId()
              << " is "       << euclidDistance
              << " km from "  << proxy->GetProxyName()
              << " at time "  << (double) SpGetTime()
              << " seconds."  << endl;
    }
    testProxyItem = (F_SpProxyItem *) testProxyItem->get_link();
  }
  SCHEDULE_SubSensorSearch(SpGetTime() + 10.0, SpGetObjHandle());
}
```

Example 11.4: DM Submarine Simulation Object Implementation

The `main` for this object is shown in Example 11.5. As usual, `main` is used to plug in all simulation objects and events, and is followed by a call to `ExecuteSpeedes`.

```
// Main.C
#include "SpMainPlugIn.H"
#include "SpFreeObjProxy.H"
#include "S_Ship.H"
#include "S_Submarine.H"
```

```

void PlugInHLA();

int main(int argc, char **argv) {
  PLUG_IN_SIMOBJ(S_Ship);
  PLUG_IN_SIMOBJ(S_Submarine);

  PLUG_IN_EVENT(ShipSensorSearch);
  PLUG_IN_EVENT(ShipChangeRadarFrequency);
  PLUG_IN_EVENT(SubSensorSearch);

  PlugInHLA();
  ExecuteSpeedes(argc, argv);
}

SpFreeObjProxy::SpFreeObjProxy(int n) { set_ntypes(n); }
```

Example 11.5: DM main

Finally, since the Ship and Submarine objects inherit from class `S_SpHLA`, there must be an `Objects.par` file. Figure 11.2 shows the `Objects.par` for the Ship simulation example.

```
Ship {
  reference SUBSCRIBE        Ship
  reference SUBSCRIBE        Submarine
  define    float            RadarFrequency     // Gigahertz
  define    int              Country            // 0 - Blue
                                                // 1 - Green
                                                // 2 - White
  define    dynamic_position Position           // Ships can move
}

Submarine {
  reference SUBSCRIBE        Ship
  define    int              Country            // 0 - Blue
                                                // 1 - Green
                                                // 2 - White
  define    position         Position           // Static Position
}
```

Figure 11.2: DM Example Objects.par

This example schedules events that examine all proxies that have been received to determine if any data is ready to be processed (i.e. polling design). Obviously, in the case of our example, there is time wasted checking to see if the other ship is within range of our ship's radar. It would be preferable if only the proxies for the ships that are within range of our sensor are on our proxy list. If this was true, then this would eliminate the time spent on the verification of data for objects which are out of range. This is the purpose of DDM, the elimination of data which is outside of the desired subscription range(s).

The goal of DDM is to reduce the number of proxies that an object receives, so that the amount of effort that is expended is greatly reduced and polling is eliminated whenever possible. Additionally, if an object has subscribed to fewer objects, there are fewer internal SPEEDES framework `TOUCH_PROXY` events that are executed whenever a proxy is changed. This results in a greater potential for parallelism, as well as fewer rollbacks and a greater potential for more performance gains. The following sections will enhance this small model, making incremental improvements which allow for much improved performance.

## 11.2 Spaces, Regions and Dimensions

In order to understand DDM and how it is used in SPEEDES, you must first understand three SPEEDES DDM terms: Region, Dimension, and Space. Regions are used to specify areas which define a simulation object's publishing or subscription data areas (i.e. area in which a simulation object is producing data or has interest). Within each Region, simulation objects define the exact range of data for which they plan to produce or consume data in. These are called Dimensions. Figure 11.3 shows two objects in one Region with their respective Dimensions. The overlap represents the common area for the two simulation objects in this Region.

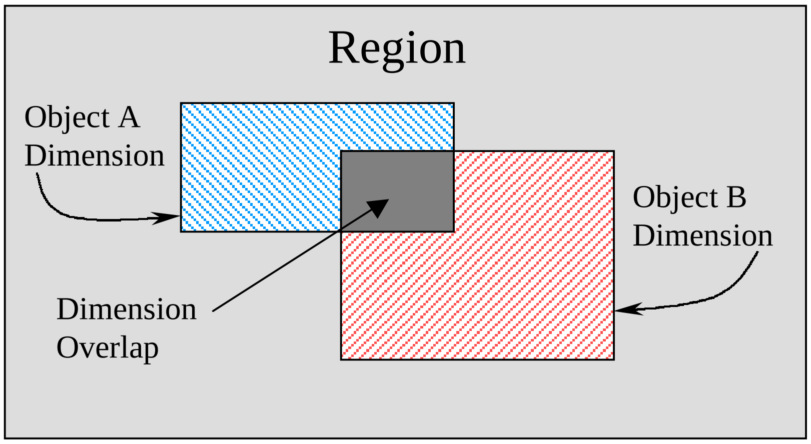


Figure 11.3: DDM Regions and Dimensions Definition

For example, the Region represented here could be frequency, with Object A producing frequencies in the range of 2 to 10 gigahertz (GHz), while Object B is looking for frequencies in the range of 8 to 15 GHz. We can set up our simulation stating that Object A is publishing frequency data in the 2 to 10 GHz range and Object B is subscribing to data in the 8 to 15 GHz range. By doing this, Object A's proxy will only be on Objects B's proxy list if the frequency is in the correct range. Hence, polling is unnecessary.

Spaces are made up of one or more Regions. For example, two different Regions could be defined with Objects A and B having areas of interest in both Regions. A Space could then be defined as containing these two Regions. This type of configuration would form an "OR" type of Region. An object subscribing to this Space would receive the proxy if the data in either Region was valid. Figure 11.4 shows two Regions defined to make up a Space. Of course, a Space can be defined with any number of Regions, and any object that publishes data in the subscriber's area of interest, will have its proxy delivered to the subscriber.

For example, Region 1 could describe the same frequency region as previously discussed. Region 2 could describe an Identify Friend or Foe area. Once again, Object A would publish its affiliation and Object B would subscribe to its area of interest (Friend, Foe or both, depending on use). With this setup, Object B would receive Object A's proxy data if either the frequency or country were within the defined ranges.

Alternatively, a subscribing object could set up two Dimensions in one Region (e.g. frequency and country). With this configuration, the frequency and country subscription in one Region would form an "AND" condition. In this case, both frequency and country must be within the specified valid range in order for the proxy to be delivered to the subscribing object.

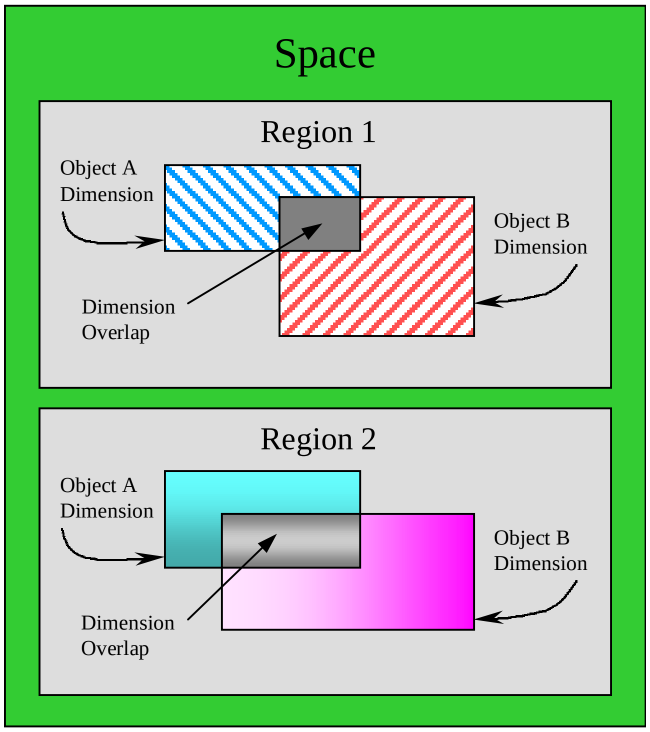


Figure 11.4: DDM Space, Regions and Dimensions Definitions

In summary, Spaces are made of Regions and Regions have well-defined Dimensions. The Space shown in Figure 11.4 contains two Regions that are used to form an "OR" subscription area. Therefore, subscription data that is in the correct data range, in either Region, will be delivered to the subscribing simulation object. Alternatively, the Space could contain one Region which contains both sets of Dimensions. This configuration would represent an "AND" subscription area, indicating that the data must be valid in both Dimensions in order for the data to be delivered to the subscriber.

## 11.3 Built-In DDM Classes and Methods

In order to use DDM, one or more Spaces must be defined. Spaces form the basis of the filtering rules used for DDM. Within each Space, one or more Regions can be defined. When more than one Region is defined, then any data which is true in any region will cause the subscriber to receive the proxy (i.e. "OR"). For each defined Region, zero or more Dimensions can be defined. Therefore, Dimension is an optional field for Regions. When more than one Dimension is defined in a Region, then all of the data in each Dimension must be true in order for the subscriber to receive the proxy (i.e. "AND").

The first required step for using DDM is to have simulation objects (i.e. children of `S_SpHLA`) define their publication and subscription Spaces. Class `S_SpHLA` provides two methods for Space publication and subscription called:

```
SpPublishSpace*  PublishSpace   (const SpSimTime& time,
                                 const char*      spaceName)
SpSubcribeSpace* SubscribeSpace (const SpSimTime& time,
                                 const char*      spaceName)
```

Additional methods on class `S_SpHLA` allow users to unpublish, unsubscribe, and search for spaces:

| Parameter | Description |
|---|---|
| time | Represents the time at which the object will start publishing or subscribing to a space. |
| spaceName | The string name for the space. |

Table 11.1: DDM Space Publication and Subscription API

```
int                SpUnpublishSpace   (char*       spaceName)
int                SpUnsubscribeSpace (char*       spaceName)
SpPublishSpace*    GetPublishSpace    (const char* spaceName)
SpSubcribeSpace*   GetSubscribeSpace  (const char* spaceName)
```

When using DDM, an additional parameter file called `InterestSpaces.par` is required (in addition to `Objects.par`). The format for `InterestSpaces.par` will be specified in each DDM filtering type in later sections. This file defines the basic structure for Spaces, Regions, and Dimensions. The reason for introducing this file now is to let the user know that the string names used for defining Spaces must have a corresponding entry in this file.

Once the Spaces have been created, Regions can be added, deleted, and searched on within each Space by using classes `SpPublishSpace` and `SpSubscribeSpace`. These classes have the following API:

```
class SpPublishSpace : public SpDDMSpace {
  public:
    SpPublishRegion*  CreateRegion (const char* regionName);
    SpPublishRegion*  FindRegion   (const char* regionName);
    void              DeleteRegion (const char* regionName);
    void              Update();
    ...
};

class SpSubscribeSpace : public SpDDMSpace {
  public:
    SpSubscribeRegion*  CreateRegion (const char* regionName);
    SpSubscribeRegion*  FindRegion   (const char* regionName);
    void                DeleteRegion (const char* regionName);
    void                Update();
    ...
};
```

Parameter `regionName` specifies the user-defined string name for the Region defined in the Space. This string can be any name and does not have a corresponding entry in `InterestSpaces.par`.

Finally, the Dimensions for the Region must be initialized. Classes `SpPublishRegion` and `SpSubscribeRegion` provide the following methods for manipulating Dimensions:

```
// Enumeration dimension modifier
void CreateDimension(const char* dimName, const char* enumName, ...)

// Double dimension modifier
void CreateDimension(const char* dimName, double lo, double hi, ...)

SpDDMDim* FindDimension   (const char *dimName)
void      DeleteDimension (const char *dimName)
```

```
// The following methods are available on SpSubscribeRegion only
void AddClass    (const char *className, ...)
void RemoveClass (const char *className, ...)
```

| Parameter | Description |
|---|---|
| dimName | This is the name of the Dimension which must have a corresponding entry in `InterestSpaces.par`. This name must be defined in the Space section in `InterestSpaces.par`. |
| enumName | A name within an enumeration Space. This name must exist within the enumeration Dimension, as specified in the `InterestSpaces.par`. |
| lo | The lower boundary value for a double Dimension. The value specified here must be between the `Lo` and `Hi` values specified for this Dimension in `InterestSpaces.par`. |
| hi | The upper boundary value for a double Dimension. The value specified here must be between the `Lo` and `Hi` values specified for this Dimension in `InterestSpaces.par`. |
| className | For the `AddClass` and `RemoveClass` methods, class names must be supplied that match the names of classes, as specified in `Objects.par`. |
| ... | The ellipse operator indicating that these methods accept variable length argument list. The list can be one or more of the previously defined argument(s). In the case of the double Dimension methods, the input argument must be specified in `Lo` and `Hi` input pairs. |

Table 11.2: DDM Dimension Publication and Subscription API

This completes the description of the classes and methods required to use DDM. In summary, the basic steps required to use DDM are:

1. Define the publication and subscription Spaces
2. Define the publication and subscription Regions.
3. Define the publication and subscription Dimension. This step is required for Class Type filtering.
4. Call method `Update` on the publication and subscription Spaces.

In order to better understand how DDM works within the constructs of SPEEDES, let us reexamine the ship and submarine example previously introduced and apply different filtering techniques to it. The following sections will show how to perform class type, enumeration, double, and range-based filtering.

## 11.4 Class Type Filtering

DM provides basic filtering for objects on a class-by-class basis. This type of filtering is set in either `Objects.par` or can be set by DM services found in `S_SpHLA` during run time. This is useful when the number of objects is quite small or when subscription of this type really is necessary. However, when simulations begin to grow or the subscriptions are not always necessary, a finer version of class based subscription can be very useful.

DDM allows for subscription and publication to be performed dynamically on a class-by-class basis. When using class `S_SpHLA` as the base class for simulation objects, users must supply an `Objects.par` as introduced previously. When using DDM, there is an additional required parameter file called `InterestSpaces.par`. For the Ship and Submarine example, the `InterestSpaces.par` file will contain one Space called “Ocean”, as shown in Figure 11.5.

```
InterestSpaces {
  reference Space Ocean                  // Space called "Ocean"
}

Ocean {
}
```

Figure 11.5: InterestSpaces.par for Class Type Filtering

Example 11.6 shows the new Ship implementation file. The changes made to this file are shown on lines 52 through 59. Specifically, lines 52 through 54 register this object's publication Space called "Ocean" with SPEEDES at $t = 0.0$ (i.e. dynamic subscription). Lines 55 through 59 create a subscription space at $t = 0.0$.

The strings used when defining the publication and subscription regions can be any user-defined string. The names defined in `InterestSpaces.par` and the Region names used in the source code have no relationship to each other. The Ship simulation objects use method `AddClass` on line 58 to add class types "Ship" and "Submarine" to its class type subscription list (i.e. all Ship and Submarine proxies will be delivered to the Ship simulation object). Notice that the names "Ship" and "Submarine" used in method `AddClass` are the same names defined in `Objects.par`.

```
 1  // S_Ship.C
 2  #include "SpGlobalFunctions.H"
 3  #include "SpSchedule.H"
 4  #include "RB_ostream.H"
 5  #include "SpObjProxy.H"
 6  #include "SpPolyMotion.H"
 7  #include "SpConstantPosition.H"
 8  #include "SpFreeDynAttributes.H"
 9  #include "F_SpProxyItem.H"
10  #include "S_Ship.H"

11  #include "SpPublishSpace.H"
12  #include "SpSubscribeSpace.H"
13  #include "SpDDMRegion.H"

14  void S_Ship::Init() {
15    DEFINE_ATTRIBUTE(RadarFrequency,  "RadarFrequency");
16    DEFINE_ATTRIBUTE(Country,         "Country");
17    DEFINE_ATTRIBUTE(Position,        "Position");

18    POLY_1_MOTION* line = (POLY_1_MOTION *)
19       FreeDynamicAttributes->new_object(POLY_1_MOTION_ID);

20    line->SetECR();
21    if (SpGetSimObjKindId() == 0) {
22      line->AddPoints(0.0,    3600.0,
23                      0.0,    6400.0,  40.0, 0.0, 1.0,
24                      3600.0, 6400.0, -60.0, 0.0, 1.0,
25                      END_POLY_MOTION);
26      Country        = 0;
27      RadarFrequency = 5;
28      RadarRange     = 10;
29    }
```

```
  else {
    if (SpGetSimObjKindId() == 1) {
      line->AddPoints(0.0,    3600.0,
                      0.0,    6400.0, 0.0,  50.0, 1.0,
                      3600.0, 6400.0, 0.0, -50.0, 1.0,
                      END_POLY_MOTION);
      Country        = 1;
      RadarFrequency = 8;
      RadarRange     = 15;
    }
    else {
      line->AddPoints(0.0,    3600.0,
                      0.0,    6400.0, -40.0,  40.0, 1.0,
                      3600.0, 6400.0,  50.0, -50.0, 1.0,
                      END_POLY_MOTION);
      Country        = 2;
      RadarFrequency = 11;
      RadarRange     = 0;
    }
  }
  line->MakePoly();
  Position += line;

  SpPublishSpace* pubSpace = PublishSpace(0.0, "Ocean");
  pubSpace->CreateRegion("Ship_Publication_Region");
  pubSpace->Update();

  SpSubscribeSpace*  subSpace  = SubscribeSpace(0.0, "Ocean");
  SpSubscribeRegion* subRegion =
    subSpace->CreateRegion("Ship_Subscription_Region");
  subRegion->AddClass("Ship", "Submarine", NULL);
  subSpace->Update();

  SCHEDULE_ShipSensorSearch(5.0, SpGetObjHandle());
  SCHEDULE_ShipChangeRadarFrequency(10.0, SpGetObjHandle());
}

void S_Ship::SensorSearch() {
  int i;
  static int shipMgrId         = SpGetSimObjMgrId("S_Ship_MGR");
  static int submarineMgrId    = SpGetSimObjMgrId("S_Submarine_MGR");
  F_SpProxyItem* testProxyItem = (F_SpProxyItem *)
    GetRemoteObjectProxies()->get_top();

  for (i = 0; i < GetRemoteObjectProxies()->get_length(); ++i) {
    SpObjProxy* proxy = testProxyItem->GetObjProxy();

    if (proxy->GetProxySimObjMgrId() == shipMgrId) {
      ProcessShipProxy(proxy);
    }

    if (proxy->GetProxySimObjMgrId() == submarineMgrId) {
      ProcessSubmarineProxy(proxy);
    }
    testProxyItem = (F_SpProxyItem *) testProxyItem->get_link();
  }
  SCHEDULE_ShipSensorSearch(SpGetTime() + 10.0, SpGetObjHandle());
}

void S_Ship::ChangeRadarFrequency() {
  RadarFrequency = RadarFrequency + 0.5;
  if (RadarFrequency > 5.0 + 3 * SpGetSimObjKindId() + 1.0) {
    RadarFrequency = 5.0 + 3 * SpGetSimObjKindId() - 1.0;
  }
  SCHEDULE_ShipChangeRadarFrequency(SpGetTime() + 10.0, SpGetObjHandle());
}

void S_Ship::ProcessShipProxy(SpObjProxy* proxy) {
  double     myECR[3];
  double     remoteECR[3];
  double     difference[3];
  static int refPosition =
    proxy->GetReference("Position", "Ship");

  Position(SpGetTime(), ECR, &myECR[0]);
  proxy->GetDynamicPosition(refPosition, remoteECR,
                            ECR,         SpGetTime());
  Difference(myECR, remoteECR, difference);
  PrintDistance(proxy, difference);
}

void S_Ship::ProcessSubmarineProxy(SpObjProxy* proxy){
  double     MyECR[3];
  double     RemoteECR[3];
  double     difference[3];
  static int refPosition =
    proxy->GetReference("Position", "Submarine");

  Position(SpGetTime(), ECR, &MyECR[0]);
  proxy->GetPosition(refPosition, RemoteECR,
                     ECR,         SpGetTime());
  Difference(MyECR, RemoteECR, difference);
  PrintDistance(proxy, difference);
}

void S_Ship::PrintDistance(SpObjProxy* proxy, double difference[3]) {
  double euclidDistance = Magnitude(difference);
  if (euclidDistance < RadarRange) {
     RB_cout << "Ship "     << SpGetSimObjKindId()
             << " is "      << euclidDistance
             << " km from " << proxy->GetProxyName()
             << " at time " << (double) SpGetTime()
             << " seconds." << endl;
  }
}
```

Example 11.6: Class Type Filter Modifications to S_Ship

Example 11.7 shows the new Submarine implementation file. The submarine publishes itself on lines 18 through 20 and subscribes to Ships on lines 21 through 25. Notice on line 21 that the Submarine is subscribing to the `Ocean` Space at $t = 500.0$. Now data will not be delivered to the Submarine until $t = 500.0$. Lines 80 through 84 show how to unsubscribe from a Space. After $t = 2400.0$, all proxy items on the Submarine proxy list will be removed. Hence, there will be no more data displayed on the screen.

```
// S_Submarine.C
#include "SpGlobalFunctions.H"
#include "SpSchedule.H"
#include "RB_ostream.H"
#include "SpObjProxy.H"
#include "SpExportAttribute.H"
#include "F_SpProxyItem.H"
#include "S_Submarine.H"

#include "SpPublishSpace.H"
#include "SpSubscribeSpace.H"
#include "SpDDMRegion.H"

void S_Submarine::Init() {
  DEFINE_ATTRIBUTE(Country,   "Country");
  DEFINE_ATTRIBUTE(Position,  "Position");

  Country = 2;
  double pos[3] = {6400.0, -10.0, -6.0};
  Position.SetECR(pos);

  SpPublishSpace* pubSpace = PublishSpace(0.0, "Ocean");
  pubSpace->CreateRegion("Submarine_Publication_Region");
  pubSpace->Update();

  SpSubscribeSpace*  subSpace  = SubscribeSpace(500.0, "Ocean");
  SpSubscribeRegion* subRegion =
    subSpace->CreateRegion("Submarine_Subscription_Region");
  subRegion->AddClass("Ship", NULL);
  subSpace->Update();

  SCHEDULE_SubSensorSearch(0.0, SpGetObjHandle());
  SCHEDULE_SubModifySubscriptions(0.0, SpGetObjHandle());
}

void S_Submarine::SensorSearch() {
  int            i;
  double         MyECR[3];
  double         RemoteECR[3];
  double         difference[3];

  static int     shipMgrId     = SpGetSimObjMgrId("S_Ship_MGR");
  F_SpProxyItem* testProxyItem = (F_SpProxyItem *)
    GetRemoteObjectProxies()->get_top();

  for (i = 0; i < GetRemoteObjectProxies()->get_length(); ++i) {
    SpObjProxy* proxy             = testProxyItem->GetObjProxy();
    static int  refCountry        =
      proxy->GetReference("Country",        "Ship");
    static int  refRadarFrequency =
      proxy->GetReference("RadarFrequency", "Ship");
    static int  refPosition       =
      proxy->GetReference("Position",       "Ship");

    int    targetCountry          = proxy->GetInt(refCountry);
    double targetFrequency        = proxy->GetFloat(refRadarFrequency);
    /*
     * Print Ship’s Country and Radar Frequency
     */
    RB_cout << "Submarine "                    << SpGetSimObjKindId()
            << " detects "                     << proxy->GetProxyName()
            << " from country "                << targetCountry
            << " emitting radar frequency at " << targetFrequency
            << " GHz at time "                 << (double) SpGetTime()
            << endl;
    /*
     * Range base filtering
     */
    Position.GetECR(&MyECR[0], SpGetTime());
    proxy->GetDynamicPosition(refPosition, RemoteECR,
                              ECR,         SpGetTime());
    Difference(MyECR, RemoteECR, difference);
    double euclidDistance = Magnitude(difference);
    if (20.0 > euclidDistance) {
      RB_cout << "Submarine " << SpGetSimObjKindId()
              << " is "       << euclidDistance
              << " km from "  << proxy->GetProxyName()
              << " at time "  << (double) SpGetTime()
              << " seconds."  << endl;
    }
    testProxyItem = (F_SpProxyItem *) testProxyItem->get_link();
  }
  SCHEDULE_SubSensorSearch(SpGetTime() + 10.0, SpGetObjHandle());
}

void S_Submarine::ModifySubscriptions() {
  /*
   * If simulation time has reached 2400.0 seconds then unsubscribe
   * (i.e. delete) Region "Submarine_Subscription_Region".
   */
  if (2400.0 == SpGetTime()) {
    SpSubscribeSpace* subSpace = FindSubscribeSpace("Ocean");
    subSpace->DeleteRegion("Submarine_Subscription_Region");
    subSpace->Update();
  }
  SCHEDULE_SubModifySubscriptions(SpGetTime() + 300.0, SpGetObjHandle());
}
```

Example 11.7: Class Type Filter Modifications to S_Submarine

Finally, Example 11.8 and Figure 11.6 show the new `main` and `Objects.par` files, respectively. The DDM functionality is added in `main` on line 7 with the call to `PlugInDDM`. `PlugInDDM` has to be called after `PlugInHLA`. Notice also that in `Objects.par` the `reference SUBSCRIBE` lines were removed, which turns DM off.

```
1  // Main.C
2  #include "SpMainPlugIn.H"
3  #include "SpFreeObjProxy.H"
```

```
#include "S_Ship.H"
#include "S_Submarine.H"

void PlugInHLA();
void PlugInDDM();

int main(int argc, char **argv) {
  PLUG_IN_SIMOBJ(S_Ship);
  PLUG_IN_SIMOBJ(S_Submarine);

  PLUG_IN_EVENT(ShipSensorSearch);
  PLUG_IN_EVENT(ShipChangeRadarFrequency);
  PLUG_IN_EVENT(SubSensorSearch);
  PLUG_IN_EVENT(SubModifySubscriptions);

  PlugInHLA();
  PlugInDDM();
  ExecuteSpeedes(argc, argv);
}

SpFreeObjProxy::SpFreeObjProxy(int n) { set_ntypes(n); }
```

Example 11.8: Class Type Filter Modifications to main

```
Ship {
  define    float            RadarFrequency     // Gigahertz
  define    int              Country            // 0 - Blue
                                                // 1 - Green
                                                // 2 - White
  define    dynamic_position Position           // Ships can move
}

Submarine {
  define    int              Country            // 0 - Blue
                                                // 1 - Green
                                                // 2 - White
  define    position         Position           // Static Position
}
```

Figure 11.6: Objects.par for Class Type Filtering Example

When this example is executed, notice that the Submarine does not output any data until $t = 500.0$, and it stops outputting data at $t = 3600.0$. Even though data is not being output, the event `SubSensorSearch` continues to execute every 10.0 seconds, which is, essentially, wasted processing time. Notice that the first output displayed by the Submarine is at $t = 510.0$ and not at $t = 500.0$. This is due to the fact that the priority fields of the proxy events have a lower priority (i.e. executes first) than the event for subscription. By using SPEEDES built-in event handlers, these problems can be significantly reduced. This is described in section 11.5.

### 11.4.1 Attribute Level Filtering

Although not technically part of the DDM system, attribute level subscription is another form of update filtering worth mentioning here. Above, we showed how DDM could be used to filter by object class. The next obvious step is to filter updates based on a subset of attributes in the class. That is, the subscriber specifies a subset of attributes within a class, and update events are sent to the subscriber only if one or more attributes in the subset has changed. This subset is called the attribute subscription for the class. Note that this filtering applies only to object updates and not to discover or undiscover events.

Attribute level subscription is an independent filtering system that can be used in conjunction with DDM (or DM). Refer to Section 9.4.5 for an explanation of how to define an attribute subscription for a given class.

## 11.5 DDM Event Handlers Optimization

SPEEDES contains four built-in event handlers that can be used in conjunction with DM and DDM. The names and descriptions of these event handlers are shown in Table 11.3.

| Handler Name | Description |
|---|---|
| Discover Object | Handlers added for this trigger respond when a new proxy is delivered. |
| Reflect Attributes | Handlers added for this trigger respond when a proxy is changed. |
| Update Attributes | Handlers added for this trigger respond when an object's own proxy is changed. |
| UnDiscover Object | Handlers added for this trigger respond when a proxy is removed. |

Table 11.3: DM and DDM Built-In Event Handler

For event handlers, the proxy being modified is available by calling function `SpGetMsgData` and type casting the returned value to a `F_SpProxyItem` pointer. These event handlers are used by registering the named event handler trigger string with SPEEDES (i.e. define the handler using macro `DEFINE_SIMOBJ_HANDLER` and register it with SPEEDES using method `AddHandler`).

Returning to the Ship and Submarine example, Examples 11.9 and 11.10 show the new code for the Submarine simulation object. Other than a new header file, lines 16, 17, 21, and 28 through 31 have been added to the `Submarine.H` file. The Submarine implementation file has been changed as follows:

1. Line 12 added include file `SpProc.H`.

2. Lines 29 through 31, initialize the counter semaphore to 0 and adds event handlers `Discover Object` and `UnDiscover Object`.

3. Turned the code in lines 33 through 85 from a point-to-point event into a process model event. The primary points of interest are on lines 45 and 82. The `WAIT_FOR` process model construct waits until the semaphore has been set. In this example, it will be set when a proxy has been delivered, to this simulation object (handler `Discover Object` sets the semaphore). The `WAIT` process model construct waits 10.0 seconds.

4. Lines 98 through 100 implement `Discover Object` event handlers.

5. Lines 101 implements `UnDiscover Object` event handlers.

```
1  // S_Submarine.H
2  #ifndef S_Submarine_H
3  #define S_Submarine_H

4  #include "S_SpHLA.H"
5  #include "SpDefineSimObj.H"
```

```
#include "SpDefineEvent.H"
#include "SpDefineHandler.H"
#include "SpProcSem.H"

class S_Submarine : public S_SpHLA {
  public:
    S_Submarine() : S_SpHLA("Submarine") {}
    ˜S_Submarine() {}
    void   Init();

    void   SensorSearch();
    void   ModifySubscriptions();
    void   DiscoverProxy();
    void   UnDiscoverProxy();

  private:
    INT_ATTRIBUTE              Country;         // Country
    POSITION_ATTRIBUTE         Position;        // Submarine position
    SpCounterSem               NumProxiesSem;   // Number of Active
                                                //  Proxies
};

DEFINE_SIMOBJ(S_Submarine, 1, SCATTER);
DEFINE_SIMOBJ_EVENT_0_ARG(SubSensorSearch, S_Submarine, SensorSearch);
DEFINE_SIMOBJ_EVENT_0_ARG(SubModifySubscriptions, S_Submarine,
                          ModifySubscriptions);
DEFINE_SIMOBJ_HANDLER(SubmarineDiscoverProxy, S_Submarine,
                      DiscoverProxy);
DEFINE_SIMOBJ_HANDLER(SubmarineUnDiscoverProxy, S_Submarine,
                      UnDiscoverProxy);
#endif
```

Example 11.9: Class Type Filter Modifications to S_Submarine Definition

```
// S_Submarine.C
#include "SpGlobalFunctions.H"
#include "SpSchedule.H"
#include "RB_ostream.H"
#include "SpObjProxy.H"
#include "SpExportAttribute.H"
#include "F_SpProxyItem.H"
#include "S_Submarine.H"

#include "SpPublishSpace.H"
#include "SpSubscribeSpace.H"
#include "SpDDMRegion.H"
#include "SpProc.H"

void S_Submarine::Init() {
  DEFINE_ATTRIBUTE(Country,   "Country");
  DEFINE_ATTRIBUTE(Position,  "Position");

  Country = 2;
  double pos[3] = {6400.0, -10.0, -6.0};
  Position.SetECR(pos);
  SpPublishSpace* pubSpace = PublishSpace(0.0, "Ocean");
  pubSpace->CreateRegion("Submarine_Publication_Region");
  pubSpace->Update();

  SpSubscribeSpace*  subSpace  = SubscribeSpace(500.0, "Ocean");
  SpSubscribeRegion* subRegion =
    subSpace->CreateRegion("Submarine_Subscription_Region");
  subRegion->AddClass("Ship", NULL);
  subSpace->Update();

  SCHEDULE_SubSensorSearch(0.0, SpGetObjHandle());
  SCHEDULE_SubModifySubscriptions(0.0, SpGetObjHandle());

  NumProxiesSem = 0;
  AddHandler(SubmarineDiscoverProxy_HDR_ID(),   "Discover Object");
  AddHandler(SubmarineUnDiscoverProxy_HDR_ID(), "UnDiscover Object");
}

void S_Submarine::SensorSearch() {
  P_VAR;
  int            i;
  double         MyECR[3];
  double         RemoteECR[3];
  double         difference[3];
  SpObjProxy*    proxy;
  static int     shipMgrId     = SpGetSimObjMgrId("S_Ship_MGR");
  F_SpProxyItem* testProxyItem = (F_SpProxyItem *)
    GetRemoteObjectProxies()->get_top();
  P_BEGIN(2);
  for (;;) {
    WAIT_FOR(1, NumProxiesSem, -1);
    for (i = 0; i < GetRemoteObjectProxies()->get_length(); ++i) {
      proxy             = testProxyItem->GetObjProxy();
      static int  refCountry        =
        proxy->GetReference("Country",        "Ship");
      static int  refRadarFrequency =
        proxy->GetReference("RadarFrequency", "Ship");
      static int  refPosition       =
        proxy->GetReference("Position",       "Ship");

      int    targetCountry          = proxy->GetInt(refCountry);
      double targetFrequency        = proxy->GetFloat(refRadarFrequency);
      /*
       * Print Ship’s Country and Radar Frequency
       */
      RB_cout << "Submarine "                     << SpGetSimObjKindId()
              << " detects "                      << proxy->GetProxyName()
              << " from country "                 << targetCountry
              << " emitting radar frequency at " << targetFrequency
              << " GHz at time "                  << (double) SpGetTime()
              << endl;
      /*
       * Range base filtering
       */
      Position.GetECR(&MyECR[0], SpGetTime());
      proxy->GetDynamicPosition(refPosition, RemoteECR,
                                ECR,         SpGetTime());
      Difference(MyECR, RemoteECR, difference);
      double euclidDistance = Magnitude(difference);
      if (20.0 > euclidDistance) {
        RB_cout << "Submarine " << SpGetSimObjKindId()
                << " is "       << euclidDistance
                << " km from "  << proxy->GetProxyName()
                << " at time "  << (double) SpGetTime()
                << " seconds."  << endl;
      }
      testProxyItem = (F_SpProxyItem *) testProxyItem->get_link();
    }
    WAIT(2, 10);      // Wait 10 Seconds
  }
  P_END;
}

void S_Submarine::ModifySubscriptions() {
  /*
   * If simulation time has reached 2400.0 seconds then unsubscribe
   * (i.e. delete) Region "Submarine_Subscription_Region".
   */
  if (2400.0 == SpGetTime()) {
    SpSubscribeSpace* subSpace = FindSubscribeSpace("Ocean");
    subSpace->DeleteRegion("Submarine_Subscription_Region");
    subSpace->Update();
  }
  SCHEDULE_SubModifySubscriptions(SpGetTime() + 300.0, SpGetObjHandle());
}

void S_Submarine::DiscoverProxy() {
  ++NumProxiesSem;
}

void S_Submarine::UnDiscoverProxy() {
  --NumProxiesSem;
}
```

Example 11.10: Class Type Filter Modifications to S_Submarine Implementation

Similar changes which are not shown here were made to the Ship simulation object. When the new simulation is executed, it contains $350$ fewer events, which in our case is a $14\%$ reduction. Also, at $t = 500$, two of the three Ship simulation objects appear. What happened to the third ship? Once again, there is a race condition when processing the proxies. When the semaphore is set and the process model is executed, only two of the proxies have been delivered to the Submarine simulation object. There are many solutions to this problem, which in reality, depend on the actual simulation implementation. In our case, we could have delayed the setting of the semaphore until after receipt of the all of the proxies, or perhaps created a separate process model event for each received Ship proxy. The actual implementation is left up to the reader.

## 11.6 DDM Built-In Filtering Types

We just completed discussing class type filtering, provided by DDM. This is essentially DM, except that the user can specify start and stop times for the filtering during simulation execution. While this may be all that is required, it offers little in the way of improved simulation run-time performance, since the filtering is very coarse. SPEEDES' built-in DDM contains three additional levels of filtering, which are the ability to filter on enumerated types (i.e. strings), doubles, and ranges (i.e. distance between objects). These different types of filtering techniques are discussed in the following sections.

### 11.6.1 Enumerated Value Filtering

Enumerated value filtering provides a mechanism for filtering on user-defined strings. This type of filter is achieved by adding the filter strings to file `InterestSpaces.par` and specifying the publication and subscription ranges for these strings in the simulation objects. Let us modify the Ship and Submarine example and use this filtering type for the `Country` attribute.

Figure 11.7 shows the new `InterestSpaces.par`. Notice that, in the Space `Ocean`, Dimension `Affiliation` has been declared as an `EnumType`. Lines 7 through 12 contain the definition for this enumeration type. It contains three enumerations called `Blue`, `Green`, and `White`. `Distribute` is a required field, which can have a value of "T" or "F". When this value is "T", SPEEDES creates one hierarchical grid per enumeration type and distributes these across available nodes. If it is "F", then only one hierarchical grid is created.

```
 1  InterestSpaces {
 2    reference Space       Ocean          // Space called "Ocean"
 3  }

 4  Ocean {
 5    reference EnumType    Affiliation    // Dimension for Enums
 6  }

 7  Affiliation {
 8    enum      Blue                       // Enumeration type "Blue"
 9    enum      Green                      // Enumeration type "Green"
10    enum      White                      // Enumeration type "White"
11    logical   Distribute  T
12  }
```

Figure 11.7: InterestSpaces.par for Enumeration Filtering

The code for creating and publishing the Ship simulation object in Example 11.10 should be replaced with the following:

```
  SpPublishSpace*  pubSpace  = PublishSpace(0.0, "Ocean");
  SpPublishRegion* pubRegion =
    pubSpace->CreateRegion("Ship_Publication_Region");
  pubRegion->CreateDimension("Affiliation",
                             GetAffiliation(SpGetSimObjKindId()),
                             NULL);
  pubSpace->Update();
```

The only change here is that we are now adding a Dimension to the Region. The name for the Dimension is the same as specified in `InterestSpaces.par`. The second parameter in `CreateDimension` is the string "Blue", "Green", or "White" (function `GetAffiliation` returns one of these values). The code for function `GetAffiliation` is shown in Example 11.11.

```
static const char *Countries[] = { "Blue", "Green", "White" };

const char* GetAffiliation(int index) {
  return (Countries[index % 3]);
}
```

Example 11.11: Function GetAffiliation

The code shown in Example 11.12 shows the new code for the Submarine simulation object. Specific changes include:

1. Lines 22 through 27 have been modified to support subscriptions to different countries. The Submarine simulation object creates one Region called `Submarine_Subscription_Region`. One enumeration Dimension is created that subscribes to countries Green and White within this region. In addition, the `AddClass` method is shown subscribing to class `Ship`. In our case, the results of this example are the same with or without this line, since there is only one object. However, if there were additional objects in this example that contained the same attributes and the Submarine could not detect these objects (e.g. satellite), then this method would remove those types. In other words, class type filtering can be used in conjunction with the other type of DDM filters.

2. Lines 88 through 98 were added to method `ModifySubscriptions`. This changes the current subscription in Dimension `Affiliation` from Green or White to Blue.

```
 1  // S_Submarine.C
 2  #include "SpGlobalFunctions.H"
 3  #include "SpSchedule.H"
 4  #include "RB_ostream.H"
 5  #include "SpObjProxy.H"
 6  #include "SpExportAttribute.H"
 7  #include "F_SpProxyItem.H"
 8  #include "S_Submarine.H"

 9  #include "SpPublishSpace.H"
10  #include "SpSubscribeSpace.H"
11  #include "SpDDMRegion.H"
12  #include "SpProc.H"

13  void S_Submarine::Init() {
14    DEFINE_ATTRIBUTE(Country,   "Country");
15    DEFINE_ATTRIBUTE(Position,  "Position");

16    Country = 2;
17    double pos[3] = {6400.0, -10.0, -6.0};
18    Position.SetECR(pos);

19    SpPublishSpace* pubSpace = PublishSpace(0.0, "Ocean");
20    pubSpace->CreateRegion("Submarine_Publication_Region");
21    pubSpace->Update();

22    SpSubscribeSpace*  subSpace  = SubscribeSpace(500.0, "Ocean");
23    SpSubscribeRegion* subRegion =
24      subSpace->CreateRegion("Submarine_Subscription_Region");
25    subRegion->AddClass("Ship", NULL);
```

```
  subRegion->CreateDimension("Affiliation", "Green", "White", NULL);
  subSpace->Update();

  SCHEDULE_SubSensorSearch(0.0, SpGetObjHandle());
  SCHEDULE_SubModifySubscriptions(0.0, SpGetObjHandle());

  NumProxiesSem = 0;
  AddHandler(SubmarineDiscoverProxy_HDR_ID(),   "Discover Object");
  AddHandler(SubmarineUnDiscoverProxy_HDR_ID(), "UnDiscover Object");
}

void S_Submarine::SensorSearch() {
  P_VAR;
  int            i;
  double         MyECR[3];
  double         RemoteECR[3];
  double         difference[3];
  SpObjProxy*    proxy;
  static int     shipMgrId     = SpGetSimObjMgrId("S_Ship_MGR");
  F_SpProxyItem* testProxyItem = (F_SpProxyItem *)
    GetRemoteObjectProxies()->get_top();
  P_BEGIN(2);
  for (;;) {
    WAIT_FOR(1, NumProxiesSem, -1);
    for (i = 0; i < GetRemoteObjectProxies()->get_length(); ++i) {
      proxy             = testProxyItem->GetObjProxy();
      static int  refCountry        =
        proxy->GetReference("Country",        "Ship");
      static int  refRadarFrequency =
        proxy->GetReference("RadarFrequency", "Ship");
      static int  refPosition       =
        proxy->GetReference("Position",       "Ship");

      int    targetCountry          = proxy->GetInt(refCountry);
      double targetFrequency        = proxy->GetFloat(refRadarFrequency);
      /*
       * Print Ship’s Country and Radar Frequency
       */
      RB_cout << "Submarine "                     << SpGetSimObjKindId()
              << " detects "                      << proxy->GetProxyName()
              << " from country "                 << targetCountry
              << " emitting radar frequency at " << targetFrequency
              << " GHz at time "                  << (double) SpGetTime()
              << endl;
      /*
       * Range base filtering
       */
      Position.GetECR(&MyECR[0], SpGetTime());
      proxy->GetDynamicPosition(refPosition, RemoteECR,
                                ECR,         SpGetTime());
      Difference(MyECR, RemoteECR, difference);
      double euclidDistance = Magnitude(difference);
      if (20.0 > euclidDistance) {
        RB_cout << "Submarine " << SpGetSimObjKindId()
                << " is "       << euclidDistance
                << " km from "  << proxy->GetProxyName()
                << " at time "  << (double) SpGetTime()
                << " seconds."  << endl;
      }
      testProxyItem = (F_SpProxyItem *) testProxyItem->get_link();
    }
    WAIT(2, 10);      // Wait 10 Seconds
  }
  P_END;
}

void S_Submarine::ModifySubscriptions() {
  /*
   * If simulation time has reached 1500.0 seconds then change
   * subscription Region to "Blue".
   */
  if (1500.0 == (double) SpGetTime())  {
    SpSubscribeSpace*  subSpace  = FindSubscribeSpace("Ocean");
    SpSubscribeRegion* subRegion =
      subSpace->FindRegion("Submarine_Subscription_Region");
    subRegion->ModifyDimension("Affiliation", "Blue", NULL);
    subSpace->Update();
  }
  /*
   * If simulation time has reached 2400.0 seconds then unsubscribe
   * (i.e. delete) Region "Submarine_Subscription_Region".
   */
  if (2400.0 == SpGetTime()) {
    SpSubscribeSpace* subSpace = FindSubscribeSpace("Ocean");
    subSpace->DeleteRegion("Submarine_Subscription_Region");
    subSpace->Update();
  }
  SCHEDULE_SubModifySubscriptions(SpGetTime() + 300.0, SpGetObjHandle());
}

void S_Submarine::DiscoverProxy() {
  ++NumProxiesSem;
}

void S_Submarine::UnDiscoverProxy() {
  --NumProxiesSem;
}
```

Example 11.12: Enumeration Filter Modifications to S_Submarine

When the Ship and Submarine example executes, Ship 0 is filtered out by DDM, as shown by this object not appearing in the output. What happened here is that DDM filtered out all objects that did not have the correct enumeration value. The result of this filtering is that the Submarine simulation object only had two proxies on its proxy list. At $t = 1500.0$, the subscription is changed. Now, the data for Ship 0 is output and the data for Ship 1 and Ship 2 are removed by filtering.

### 11.6.2 Double Range Filtering

Double range filtering allows users to specify filtering based on floating point numbers being within a given range. Publications and Subscriptions are handled by suppling a lower and upper boundary for the floating point number. If a range is not desired, then a single point can be used by using the same value for the upper and lower boundary.

Figure 11.8 shows the new `InterestSpaces.par` that contains an example for double range filtering. Notice that in the `Ocean` Space, Dimension `RadarFrequency` has been declared as a type `Dimension`. Lines 14 through 21 contain the definition for this double range filter type. The required lines are `Lo`, `Hi`, `Resolution[0]`, `Resolution[1]`, and `Distribute`. `Lo` and `Hi` specify the upper and lower boundaries for this Dimension, specifically 2.0 and 20.0 GHz, respectively. The data value specified in `Resolution[0]` is used in conjunction with the upper and lower boundaries to calculate how many hierarchical grids are to be created for this dimension. In this example, $(20.0-2.0)/2.0$ rounded up creates nine grids. The next two resolutions are used to further subdivide each grid, which can help optimize simulation performance.

```
 1  InterestSpaces {
 2    reference Space       Ocean           // Space called "Ocean"
 3  }

 4  Ocean {
 5    reference EnumType    Affiliation     // Dimension for Enums
 6    reference Dimension   RadarFrequency  // Dimension for Doubles
 7  }

 8  Affiliation {
 9    enum      Blue                        // Enumeration type "Blue"
10    enum      Green                       // Enumeration type "Green"
11    enum      White                       // Enumeration type "White"
12    logical   Distribute  T
13  }

14  RadarFrequency {
15    float     Lo          2.0             // GHz
16    float     Hi          20.0            // GHz

17    float     Resolution[0] 2.0
18    float     Resolution[1] 0.5
19    float     Resolution[1] 0.1
20    logical   Distribute    T
21  }
```

Figure 11.8: InterestSpaces.par for Double Range Filtering

Returning to the Ship and Submarine example, replace the Ship simulation objects publication code with the following:

```
SpPublishSpace*  pubSpace  = PublishSpace(0.0, "Ocean");
SpPublishRegion* pubRegion =
  pubSpace->CreateRegion("Ship_Publication_Region");
pubRegion->CreateDimension("Affiliation",
                           GetAffiliation(SpGetSimObjKindId()),
                           NULL);
pubRegion->CreateDimension("RadarFrequency",
                           RadarFrequency, RadarFrequency,
                           NULL);
pubSpace->Update();
```

The only change here is that Dimension `RadarFrequency` has been added to Space `Ocean`. This Dimension was added to the same Region as Dimension `Affiliation`. Both `Lo` and `Hi` arguments to method `CreateDimension` are the same, indicating that this object is producing data at one value. In this case, the frequencies are 5.0, 8.0, and 11.0 GHz for Ships 0, 1, and 2, respectively.

The code shown in Example 11.13 shows the new implementation for the Submarine simulation object. Specific changes include:

1. Lines 22 through 28 are used to to subscribe to our different Dimensions. The Submarine simulation object creates one Region called “Submarine_Subscription_Region”. Within this region, one enumeration Dimension is created that subscribes to countries “Green” and “White”. Using the same Region, another Dimension is created called `RadarFrequency`. In this dimension, the Submarine simulation object subscribes to all objects whose frequencies are between 10.0 and 14.0 GHz. This current implementation shows an “AND” condition (i.e. both must be true in order for proxies to be delivered to the Submarine). If a new Region had been created, then this would have implemented an “OR” condition.

2. Lines 100 through 122 were added to method `ModifySubscriptions`. The changes here will change the subscription regions for the Submarine. At $t = 1800.0$, the subscription Region `Submarine_Subscription_Region`, Dimension `RadarFrequency` is changed from 10.0 to 14.0 GHz to 3.5 to 6.5 GHz. At $t = 2100.0$, a new Region called `New_Submarine_Subscription_Region` is created. This Region’s `RadarFrequency` Dimensions are initialized with a frequency range of 6.5 to 9.5 GHz.

```
1   // S_Submarine.C
2   #include "SpGlobalFunctions.H"
3   #include "SpSchedule.H"
4   #include "RB_ostream.H"
5   #include "SpObjProxy.H"
6   #include "SpExportAttribute.H"
7   #include "F_SpProxyItem.H"
8   #include "S_Submarine.H"

9   #include "SpPublishSpace.H"
10  #include "SpSubscribeSpace.H"
11  #include "SpDDMRegion.H"
12  #include "SpProc.H"

13  void S_Submarine::Init() {
14    DEFINE_ATTRIBUTE(Country,   "Country");
15    DEFINE_ATTRIBUTE(Position,  "Position");

16    Country = 2;
17    double pos[3] = {6400.0, -10.0, -6.0};
18    Position.SetECR(pos);

19    SpPublishSpace* pubSpace    = PublishSpace(0.0, "Ocean");
20    pubSpace->CreateRegion("Submarine_Publication_Region");
21    pubSpace->Update();

22    SpSubscribeSpace*  subSpace = SubscribeSpace(500.0, "Ocean");
23    SpSubscribeRegion* subRegion =
24      subSpace->CreateRegion("Submarine_Subscription_Region");
25    subRegion->AddClass("Ship", NULL);
26    subRegion->CreateDimension("Affiliation", "Green", "White", NULL);
27    subRegion->CreateDimension("RadarFrequency", 10.0, 14.0, NULL);
```

```
  subSpace->Update();

  SCHEDULE_SubSensorSearch(0.0, SpGetObjHandle());
  SCHEDULE_SubModifySubscriptions(0.0, SpGetObjHandle());

  NumProxiesSem = 0;
  AddHandler(SubmarineDiscoverProxy_HDR_ID(),   "Discover Object");
  AddHandler(SubmarineUnDiscoverProxy_HDR_ID(), "UnDiscover Object");
}

void S_Submarine::SensorSearch() {
  P_VAR;
  int             i;
  double          MyECR[3];
  double          RemoteECR[3];
  double          difference[3];
  SpObjProxy*     proxy;
  static int      shipMgrId     = SpGetSimObjMgrId("S_Ship_MGR");
  F_SpProxyItem* testProxyItem = (F_SpProxyItem *)
    GetRemoteObjectProxies()->get_top();
  P_BEGIN(2);
  for (;;) {
    WAIT_FOR(1, NumProxiesSem, -1);
    for (i = 0; i < GetRemoteObjectProxies()->get_length(); ++i) {
      proxy             = testProxyItem->GetObjProxy();
      static int  refCountry        =
        proxy->GetReference("Country",        "Ship");
      static int  refRadarFrequency =
        proxy->GetReference("RadarFrequency", "Ship");
      static int  refPosition       =
        proxy->GetReference("Position",       "Ship");

      int    targetCountry          = proxy->GetInt(refCountry);
      double targetFrequency        = proxy->GetFloat(refRadarFrequency);
      /*
       * Print Ship's Country and Radar Frequency
       */
      RB_cout << "Submarine "                     << SpGetSimObjKindId()
              << " detects "                      << proxy->GetProxyName()
              << " from country "                 << targetCountry
              << " emitting radar frequency at " << targetFrequency
              << " GHz at time "                  << (double) SpGetTime()
              << endl;
      /*
       * Range base filtering
       */
      Position.GetECR(&MyECR[0], SpGetTime());
      proxy->GetDynamicPosition(refPosition, RemoteECR,
                                ECR,         SpGetTime());
      Difference(MyECR, RemoteECR, difference);
      double euclidDistance = Magnitude(difference);
      if (20.0 > euclidDistance) {
        RB_cout << "Submarine " << SpGetSimObjKindId()
                << " is "       << euclidDistance
                << " km from "  << proxy->GetProxyName()
                << " at time "  << (double) SpGetTime()
                << " seconds."  << endl;
      }
      testProxyItem = (F_SpProxyItem *) testProxyItem->get_link();
    }
    WAIT(2, 10);      // Wait 10 Seconds
  }
  P_END;
}

void S_Submarine::ModifySubscriptions() {
  /*
   * If simulation time has reached 1500.0 seconds then change
   * subscription Region Affiliation to "Blue".
   */
  if (1500.0 == (double) SpGetTime()) {
    SpSubscribeSpace*  subSpace  = FindSubscribeSpace("Ocean");
    SpSubscribeRegion* subRegion =
      subSpace->FindRegion("Submarine_Subscription_Region");
    subRegion->ModifyDimension("Affiliation", "Blue", NULL);
    subSpace->Update();
  }
  /*
   * If simulation time has reached 1800.0 seconds then change
   * subscription Region RadarFrequency range to 3.5 to 6.5 GHz".
   */
  if (1800.0 == (double) SpGetTime()) {
    SpSubscribeSpace*  subSpace  = FindSubscribeSpace("Ocean");
    SpSubscribeRegion* subRegion =
      subSpace->FindRegion("Submarine_Subscription_Region");
    subRegion->ModifyDimension("RadarFrequency", 3.5, 6.5, NULL);
    subSpace->Update();
  }
  /*
   * If simulation time has reached 2100.0 seconds then add
   * a new subscription Region and create a new Dimension
   * RadarFrequency with a range of 6.5 to 9.5 GHz".
   */
  if (2100.0 == (double) SpGetTime()) {
    SpSubscribeSpace*  subSpace  = FindSubscribeSpace("Ocean");
    SpSubscribeRegion* subRegion =
      subSpace->CreateRegion("New_Submarine_Subscription_Region");
    subRegion->CreateDimension("RadarFrequency", 6.5, 9.5, NULL);
    subSpace->Update();
  }
  /*
   * If simulation time has reached 2400.0 seconds then unsubscribe
   * (i.e. delete) Region "Submarine_Subscription_Region".
   */
  if (2400.0 == SpGetTime()) {
    SpSubscribeSpace* subSpace = FindSubscribeSpace("Ocean");
    subSpace->DeleteRegion("Submarine_Subscription_Region");
    subSpace->Update();
  }
  SCHEDULE_SubModifySubscriptions(SpGetTime() + 300.0, SpGetObjHandle());
}
```

```
void S_Submarine::DiscoverProxy() {
  ++NumProxiesSem;
}

void S_Submarine::UnDiscoverProxy() {
  --NumProxiesSem;
}
```

Example 11.13: Double Range Filter Modifications to S_Submarine

When this example is executed, the following occurs:

1. At $t = 500.0$, the Submarine starts to print out Ship 2's data. DDM is filtering out Ship 0 and Ship 1 due to these objects not meeting the country and frequency DDM requirements, respectively.

2. At $t = 1500.0$, the Submarine changes its `Affiliation` Dimension to subscribe to "Blue". This results in countries "Green" and "White" being removed from this Dimension and country "Blue" being added. This causes Region `Submarine_Subscription_Region` to have Subscriptions for any Ship that resides from country "Blue" whose radar is transmitting in the range of 10.0 to 14.0 GHz. In this example, no Ships meet this criteria. Therefore, the Submarine stops outputting Ship data.

3. At $t = 1800.0$, the Submarine changes its `RadarFrequency` Dimension to a frequency range of 3.5 to 6.5 GHz, which causes the previous definition of 10.0 to 14.0 GHz to be removed. Ship 0's radar frequency is within this range. Therefore, the Submarine starts to output Ship 0's data.

4. At $t = 2100.0$, the Submarine creates a new Region called `New_Submarine_Subscription_Region` and creates a `RadarFrequency` Dimension whose frequency range is 6.5 to 9.5 GHz. This has created two Regions within the `Ocean` Space. The result is an "OR" condition. This means that any Ship whose publication criteria meets either Submarine subscription specification, will have its data output. Ship 0 has already met the subscription criteria for Region `Submarine_Subscription_Region`. Therefore, its data continues to be output. Ship 1's radar frequency meets the new Region `New_Submarine_Subscription_Region` subscription criteria. Therefore, Ship 1's data is also displayed.

5. At $t = 2400.0$, the Submarine deletes Region `Submarine_Subscription_Region`. The result of this is Ship 0 is now filtered out by DDM. Therefore, Ship 0's data is not output. Ship 1's output is still displayed.

### 11.6.3 Range-Based Filtering

Range-based filtering is the final built-in DDM filter type. It is perhaps the most powerful and most complex of the filter types. When range-based filtering is enabled, SPEEDES automatically determines when objects are within a specified range. If we reexamine Figure 11.1, we notice that each Ship has a circular radar detection range. However, implementation of a circular detection range is extremely complex and CPU expensive. It is much easier to have a detection algorithm that is based on a rectangular grid. Figure 11.9 shows this. Notice that the simulation object's sensor detection ranges are now square instead of the original circular detection range pattern. Actually, not only are they square, but due to the DDM algorithm, the square is actually larger than shown. Therefore, while range-based filters will reduce the amount of proxies delivered to the subscribing object, proxies will be delivered to simulation objects that may be out of range (sensor range in this case). Therefore, if this is critical to the performance of the simulation, users should perform additional checks on ranges to verify that the detected objects are within range.

To use range-based filtering, a section has to be added to `InterestSpaces.par`, as shown in Figure 11.10. For range-based filtering, a `Theater` Dimension must be added to the Space. For this example, `Earth` has been added on line 7. The definition of Dimension `Earth` is shown on lines 24 through 45. The example `InterestSpaces.par` shows the required fields necessary to create a `Theater` Dimension. For range-based filtering, the `Theater` Dimension must specify the ranges for latitude, longitude, and altitude. These elements are specified in `InterestSpaces.par` by the latitude and longitude pair `LatLng`, which is broken down into `Latitude` and `Longitude`. Altitude is specified in the `Altitude` section. Common to each of those sections are the attributes `Lo`, `Hi`, `Resolution[0]`, `Resolution[1]`, and `Distribute`. The lower and upper ranges for latitude and longitude are $\pm 90°$ and $\pm 180°$, respectively. Altitude is specified in km, with 0 indicating the surface of the earth.

```
 1  InterestSpaces {
 2    reference Space       Ocean           // Space called "Ocean"
 3  }

 4  Ocean {
 5    reference EnumType    Affiliation     // Dimension for Enums
 6    reference Dimension   RadarFrequency  // Dimension for Doubles
 7    reference Theater     Earth           // Dimension for Range
 8                                          //  Based Filtering
 9  }

10  Affiliation {
11    enum      Blue                        // Enumeration type "Blue"
12    enum      Green                       // Enumeration type "Green"
13    enum      White                       // Enumeration type "White"
14    logical   Distribute  T
15  }

16  RadarFrequency {
17    float     Lo          2.0             // GHz
18    float     Hi          20.0            // GHz

19    float     Resolution[0] 2.0
20    float     Resolution[1] 0.5
21    float     Resolution[1] 0.1
22    logical   Distribute    T
23  }

24  Earth {
25    LatLng {
26      Latitude {
27        float Lo -5.0
28        float Hi  5.0
29      }

30      Longitude {
31        float Lo -10.0
32        float Hi  10.0
33      }

34      float Resolution[0] 1000
35      float Resolution[1] 500
```

```
    logical Distribute  T
  }

  Altitude {
    float Lo 0.0
    float Hi 4.0
    float Resolution[0] 4
    float Resolution[1] 1
    logical Distribute  T
  }
}
```

When a theater section is added to the `InterestSpaces.par` file, the range-based filtering is automatically enabled for all objects that publish or subscribe to this Space. No additional methods for specifying a theater's Dimensions are required. However, range-based filtering depends on the users supplying the code for seven methods that are used by the range-based filter algorithm. The required virtual methods located on `S_SpHLA` are:

- `double GetLookAheadSec()`
  All objects participating in range-based filtering must use the same value of `GetLookAheadSec`. The value returned by `GetLookAheadSec` should be smaller than the value returned by `GetMinRescheduleTimeSec`.

- `double GetMinRescheduleTimeSec()`
  Smaller values result in better filtering, but may result in too large an amount of CPU being expended for the DDM alone.

- `double GetMinExpansionKm()`
  This gives the minimum expansion of the sensor range. Smaller values result in better filtering at the expense of more frequent events to determine the proxy distribution.

- `double GetMaxSpeedKmPerSec()`
  Returns the maximum speed in km/second that this object can achieve.

- `double GetMaxSensorRangeKm()`
  Gives the maximum range of the "sensor." That is, gives the minimum range at which this object should give proxies of other objects doing range-based filtering. For example, for a radar, it would be its detection range, but for a weapon, it would be its fire range.

-
  ```
  int GetPositionTimes(double  time,
                       double& startTime,
                       double& endTime)
  ```
  Arguments `startTime` and `endTime` are filled in with the bounds of the equation of motion bounds for the passed in time. The return code is ignored for non-zero values and aborts for return of 0. For fixed position objects, fill in -$\infty$ and +$\infty$ for `startTime` and `endTime` and return 1.

-
  ```
  void GetPosition(double time,
                   double position[3],
                   double velocity[3],
                   double acceleration[3])
  ```
  Returns the `position`, `velocity` and `acceleration` for this object in Earth coordinate system at `time`.

The first three functions are tuning parameters for DDM. Lower values result in more accurate DDM (proxies are delivered closer to the correct distance) but also result in more CPU overhead for the DDM filtering.

The last four function are used for the actual computation of the delivery time for the proxies. The frequency of these computations occurs based on the sizes of the tuning parameters, and all the computations are recalculated whenever either proxy changes.

Let us modify the Ship and Submarine example one last time and incorporate the changes necessary for range-based filtering. The changes made are shown in Examples 11.14 and 11.15 for the Ship and Submarine simulation objects, respectively. The only changes made here are the addition of the seven methods required for range-based filtering. Also, in the example provided, the conditional checks for the target range were deleted. Therefore, if you run the example, you will see output from the example with a distance to the target outside of the specified sensor range, as expected.

```
// S_Ship.H
#ifndef S_SHIP_H
#define S_SHIP_H

#include "S_SpHLA.H"
#include "SpDefineSimObj.H"
#include "SpDefineEvent.H"
#include "SpDefineHandler.H"
#include "SpProcSem.H"

class S_Ship : public S_SpHLA {
  public:
    S_Ship() : S_SpHLA("Ship") {}
    ~S_Ship() {}

    void   Init();
    void   SensorSearch();
    void   ChangeRadarFrequency();
    void   DiscoverProxy();
    void   UnDiscoverProxy();
    /*
     * Range Based Filtering required methods
     */
    double GetLookAheadSec()         {return 5.0;}
    double GetMinRescheduleTimeSec() {return 10.0;}
    double GetMinExpansionKm()       {return 0.1;}
    double GetMaxSpeedKmPerSec()     {return (100.0 / 60 / 60);}
    double GetMaxSensorRangeKm()     {return RadarRange;}
    int GetPositionTimes(double  time,
                         double& startTime,
                         double& endTime) {
      return (Position.GetTimeInterval(time, startTime, endTime));
    }
    void GetPosition(double t,
                     double position[3],
                     double velocity[3],
                     double acceleration[3]) {
      Position(t, EARTH, &position[0], &velocity[0], &acceleration[0]);
    }
  private:
    DOUBLE_ATTRIBUTE           RadarFrequency;  // Gigahertz
    INT_ATTRIBUTE              Country;         // Country
    DYNAMIC_POSITION_ATTRIBUTE Position;        // Ship Position
    double                     RadarRange;      // Kilometers
    SpCounterSem               NumProxiesSem;   // Number of Active
                                                //  Proxies
    void   ProcessShipProxy(SpObjProxy* proxy);
    void   ProcessSubmarineProxy(SpObjProxy* proxy);
    void   PrintDistance(SpObjProxy* proxy, double difference[3]);
};

DEFINE_SIMOBJ(S_Ship, 3, SCATTER);
DEFINE_SIMOBJ_EVENT_0_ARG(ShipSensorSearch, S_Ship, SensorSearch);
DEFINE_SIMOBJ_EVENT_0_ARG(ShipChangeRadarFrequency, S_Ship,
                          ChangeRadarFrequency);
DEFINE_SIMOBJ_HANDLER(ShipDiscoverProxy, S_Ship, DiscoverProxy);
DEFINE_SIMOBJ_HANDLER(ShipUnDiscoverProxy, S_Ship, UnDiscoverProxy);
#endif
```

Example 11.14: Range-Based Filtering Modications to S_Ship

```
// S_Submarine.H
#ifndef S_Submarine_H
#define S_Submarine_H

#include "S_SpHLA.H"
#include "SpDefineSimObj.H"
#include "SpDefineEvent.H"
#include "SpDefineHandler.H"
#include "SpProcSem.H"

class S_Submarine : public S_SpHLA {
  public:
    S_Submarine() : S_SpHLA("Submarine") {}
    ~S_Submarine() {}
    void   Init();

    void   SensorSearch();
    void   ModifySubscriptions();
    void   DiscoverProxy();
    void   UnDiscoverProxy();
    /*
     * Range Based Filtering required methods
     */
    double GetLookAheadSec()         {return 1.0;}
    double GetMinRescheduleTimeSec() {return 2.0;}
    double GetMinExpansionKm()       {return 0.1;}
    double GetMaxSpeedKmPerSec()     {return 0.0001;}
    double GetMaxSensorRangeKm()     {return 20.0;}
    int GetPositionTimes(double  time,
                         double& startTime,
                         double& endTime) {
      startTime = -INFINITY;
      endTime   = INFINITY;
      return 1;
    }
    void GetPosition(double t,
                     double position[3],
                     double velocity[3],
                     double acceleration[3]) {
      int i;
      Position.GetEARTH(position[0], position[1], position[2]);
      for (i = 0; i < 3; ++i) {
        velocity[i]     = 0.0;
        acceleration[i] = 0.0;
      }
    }

  private:
    INT_ATTRIBUTE              Country;         // Country
    POSITION_ATTRIBUTE         Position;        // Submarine position
    SpCounterSem               NumProxiesSem;   // Number of Active
                                                //  Proxies
};

DEFINE_SIMOBJ(S_Submarine, 1, SCATTER);
DEFINE_SIMOBJ_EVENT_0_ARG(SubSensorSearch, S_Submarine, SensorSearch);
DEFINE_SIMOBJ_EVENT_0_ARG(SubModifySubscriptions, S_Submarine,
                          ModifySubscriptions);
DEFINE_SIMOBJ_HANDLER(SubmarineDiscoverProxy, S_Submarine,
                      DiscoverProxy);
DEFINE_SIMOBJ_HANDLER(SubmarineUnDiscoverProxy, S_Submarine,
                      UnDiscoverProxy);
#endif
```

Example 11.15: Range-Based Filtering Modications to S_Submarine

The output from this example is very similar to the previous results. The biggest difference is that, where in the previous example Ship 1's country and radar frequency was output starting at $t = 2400$ to the end of the simulation, this example does not. While it does start to output this information at $t = 2400$, Ship 0 moves out of range at approximately $t = 2800$ seconds, at which point range-based filtering filters out this object.

## 11.7 DDM Performance

Table 11.4 shows the event statistics from each example described in this section. As each level of filtering is increased, the number of zero lookahead events decreases (i.e. `SpTouchProxy` and `SpReflectAttribute`). The reduction of these events add additional parallelism to a simulation and reduce the number of rollbacks of user events.

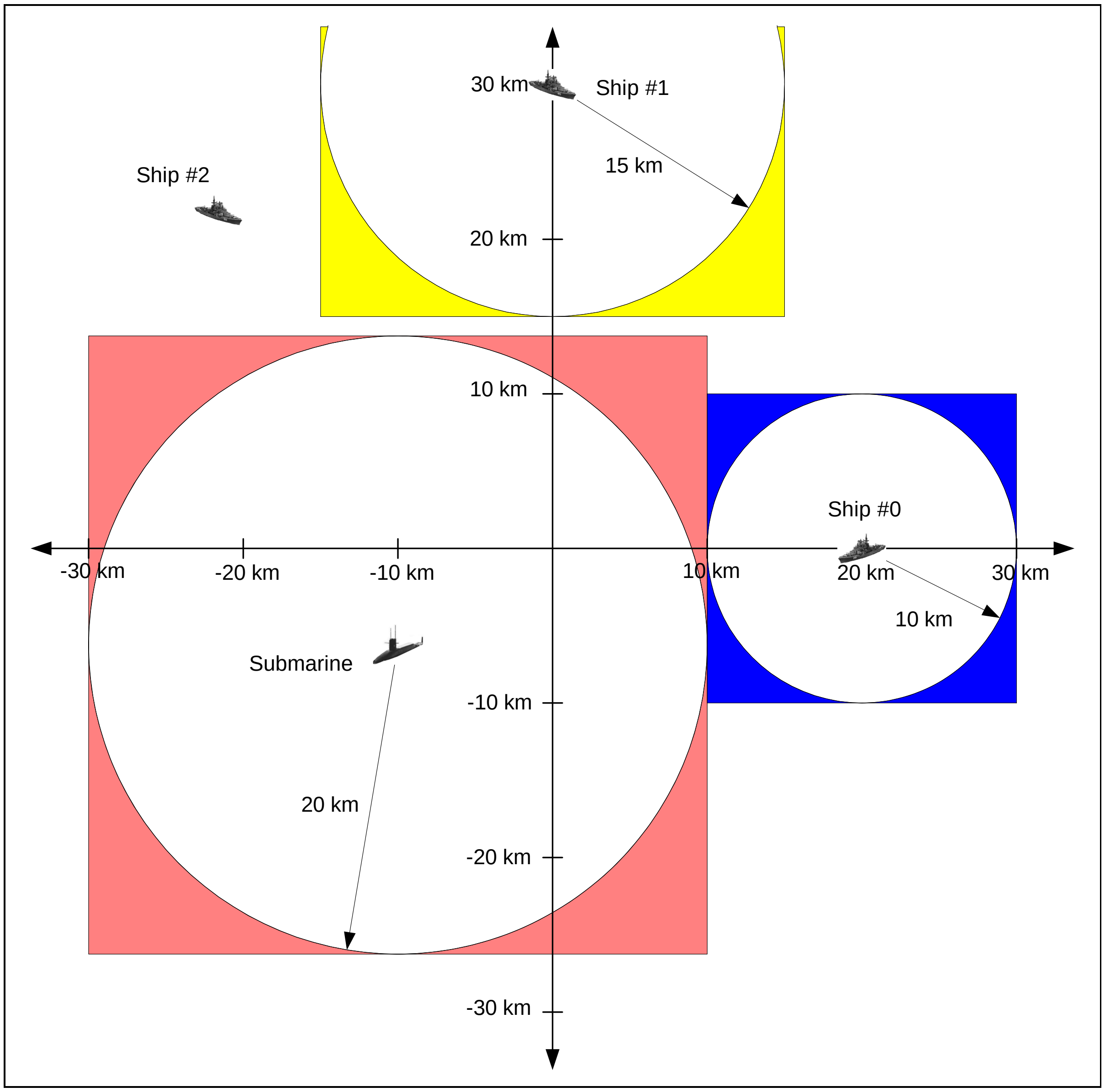


Figure 11.9: DDM Ship Simulation Layout

Figure 11.10: InterestSpaces.par for Range-Based Filtering

| Event Name | Filter Type | | | | | |
|---|---|---|---|---|---|---|
| | DM | Class Type | Handlers | Enumeration | Double | Range-Based |
| ShipSensorSearch | 1080 | 1080 | 903 | 903 | 903 | 719 |
| ShipChangeRadarFrequency | 1080 | 1080 | 1080 | 1080 | 1080 | 1080 |
| SubSensorSearch | 361 | 361 | 193 | 193 | 284 | 264 |
| SubModifySubscriptions | N/A | 13 | 13 | 13 | 13 | 13 |
| SubscribeAllClasses | N/A | 4 | 4 | 4 | 4 | 4 |
| SpTouchProxy | 3240 | 2730 | 2090 | 2090 | 2110 | 526 |
| SpReflectAttributes | 1080 | 1080 | 920 | 920 | 910 | 384 |
| SpDeliverProxyPtr | N/A | 12 | 12 | 12 | 996 | 268 |
| SpUndeliverProxyPtr | N/A | 3 | 3 | 3 | 986 | 268 |
| SpNewProxy | N/A | 7 | 18 | 18 | 516 | 244 |
| SpRemoveProxy | N/A | 3 | 3 | 3 | 428 | 63 |
| SpPublishGrid | N/A | 4 | 6 | 6 | 1110 | 1498 |
| SpUnPublishGrid | N/A | 0 | 0 | 0 | 504 | 657 |
| SpSubscribeGrid | N/A | 4 | 552 | 552 | 4967 | 34990 |
| SpUnSubscribeGrid | N/A | 4 | 3 | 3 | 11 | 76 |
| SpUnSubscribeSpace | N/A | 3 | 3 | 3 | 2 | 59 |
| SpAddSubscriberSpace | N/A | 7 | 24 | 24 | 1200 | 1557 |
| SpDeliverProxyPtrSpace | N/A | 0 | 6 | 6 | 78 | 54 |
| SpUndeliverProxyPtrSpace | N/A | 0 | 0 | 0 | 0 | 55 |
| SpPublishSpaceProcess | N/A | 12 | 12 | 12 | 3252 | 3250 |
| SpSubscribeSpaceProcess | N/A | 10 | 372 | 372 | 376 | 1810 |
| Totals | 6841 | 6414 | 6113 | 6217 | 19730 | 43572 |

Table 11.4: DDM Summary Results

The number of user events also decreased as DDM filter is increased. The examples presented in this Chapter were created for demonstrative purposes and, hence, are simple. Specifically, the user events are performing very little work. However, if these user events were very time consuming, then reducing the number of these event can increase the simulation performance.

### 11.7.1 Hierarchical Grids

A number of references have been made to hierarchical grids. These are simulation objects used to optimize the DDM within SPEEDES so that it scales in both memory and numbers of objects. These objects determine the overlaps between simulation objects that are publishing and subscribing so that the proper proxies can be delivered.

In general, creating more hierarchical grid objects results in fewer rollbacks, but this is done at the expense of a larger memory footprint. On the other hand, creating fewer hierarchical grid objects results in a smaller memory impact but will result in more rollbacks and, potentially, worse run-time performance.

## 11.8 Tips, Tricks, and Potholes

- When setting the `Distribute` parameter in `InterestSpaces.par` use the following guideline. If the dimension changes frequently, then set the parameter to true in order to minimize the number of rollbacks. If the dimension changes infrequently, then set the parameter to false.
- Specifying large values for methods `GetLookAheadSec`, `GetMinRescheduleTimeSec`, and `GetMinExpansionKm` decreases the number of DDM events at the expense of less accurate filtering.

Part V

# External Interfaces

# Chapter 12

# External Modules

The SPEEDES framework is a powerful simulation tool that contains many capabilities, which simplify the creation of parallel discrete-event simulations. But how does the user get valuable information out of the simulation for data analysis, real-time displays, debugging, etc.? SPEEDES provides a built-in class `SpStateMgr` to receive "committed" or "released" events from the simulation for use. Use of the state manager avoids having to interact with non-optimistic interfaces. The state manager offers users the following capabilities:

- Time Management
- Object Proxy Data
- User-Defined events

Note that the features described in this chapter do not work in sequential mode. When running on one node, be sure that `optimize_sequential` is set to false in `speedes.par`. See Appendix C.1 for more information.

## 12.1 Simple External Module

Use of the built-in SPEEDES external module class provides users with an easy way to to interface with a SPEEDES application.

The basic steps are:

1. Create a communication path to the SPEEDES application.
2. Inform SPEEDES of the objects and data about which the external module will receive information.
3. Advance time.
4. Process received simulation data.

This chapter will explore an external module which communicates with the example in Section 11.6.3. This external module will act like a commander, keeping an eye on what the Submarine sees and issuing commands to attack targets. We will need to extend both the submarine and ship models in order to allow the submarine to attack the ship.

First, a method `Attack` will be added to the `S_Submarine` class, and a corresponding method `CheckForDamage` will be added to the `S_Ship` class. These methods need to be added to their respective definition file, have the `DEFINE_SIMOBJ_EVENT` macro applied to the methods, and then have the new events plugged into `main`. The implementation of these methods appears in Examples 12.1 and 12.2.

```
#include "SpGlobalFunctions.H"
#include "F_SpProxyItem.H"
#include "S_Submarine.H"
#include "S_Ship.H"

void S_Submarine::Attack() {
  int            i;
  int            numObjProxies;
  int            found = 0;
  char*          objName;
  RB_queue*      remoteProxies;
  F_SpProxyItem* pItem;
  SpObjProxy*    remoteObject;

  objName       = SpGetMsgData(); // Get extra data passed to
                                  //  Attack method
  remoteProxies = GetRemoteObjectProxies();
  pItem         = (F_SpProxyItem *) remoteProxies->get_top();
  numObjProxies = remoteProxies->get_length();

  for (i = 0; i < numObjProxies; ++i) {
    remoteObject = (SpObjProxy *) pItem->GetObjProxy();
    if (!strcmp(remoteObject->GetProxyName(), objName)) {
      found = 1;
      break;
    }
    pItem = (F_SpProxyItem *) pItem->get_link();
  }
  if (found == 1) {
    static int  refCountry        =
      remoteObject->GetReference("Country", "Ship");
    int targetCountry = remoteObject->GetInt(refCountry);
    if (targetCountry != Country){
      SpObjHandle
        remoteObjectHandle(remoteObject->GetProxyNode(),
                           remoteObject->GetProxySimObjMgrId(),
                           remoteObject->GetProxySimObjLocalId());
      SCHEDULE_ShipCheckForDamage(SpGetTime() + 2.0, remoteObjectHandle);
    }
    else{
      RB_cout << "Will not attack ship of same country" << endl;
    }
  }
  else {
    RB_cout << "Could not find object " << objName
            << " to attack " << endl;
  }
}
```

Example 12.1: Attack method on S_Submarine

The Submarine receives a command to attack and obtains the name of the object to attack from the data. It then looks up the object on its list of proxies (these are the only items in range) and schedules an event for the target to check itself for damage in two seconds. Upon receiving this event, the target ship throws a random number and, 90% of the time, survives. When it does not survive, it removes all the dynamic items from its list and stops publishing or subscribing.

```
#include "SpGlobalFunctions.H"
#include "SpBaseDynItem.H"
#include "SpFreeDynAttributes.H"  // Needed for FreeDynamicAttributes
#include "RB_SpFrameworkFuncs.H"  // Needed for RB_FREE_DELETE
#include "S_Ship.H"

void S_Ship::CheckForDamage() {
  RB_SpRandom* myRandom = SpGetRandom();
  if (myRandom->GenerateUniform() > 0.9) { // 10% chance of hit
    /*
     * Return all position items to dynanmic attribute free list.
     * Remove this object from DDM participation.
     */
    SpDynItem* currItem = (SpDynItem *) Position.GetFirstElement();
    while (currItem != NULL) {
      Position -= currItem;
      RB_FREE_DELETE(FreeDynamicAttributes, currItem);
      currItem = (SpDynItem *) Position.GetFirstElement();
    }
    SpUnpublishSpace("Ocean");   // Stop publishing
    SpUnsubscribeSpace("Ocean"); // Stop subscribing
  }
}
```

Example 12.2: CheckForDamage method on S_Ship

Other changes will need to be made to the rest of the Ship model in order to manage the rest of the Ship's operations but are more extensive than can be presented here. These changes include:

- Stop the `SensorSearch` process because the Ship is no longer functioning.
- Store cancel handles for many events that may be changing the subscription or publication space, or make these events check for a `NULL` space prior to operating on it.
- Change any other related code that changes the spaces in order to check if it exists first.

Now that a simulation is available, the actual external module can be written for extracting and inserting data from and into the simulation. There are three classes most often used for building external modules. These are the `SpStateMgr`, the `SpStateMgrEvent`, and the `SpEmHostUser`.

Communication with the external modules is handled by the `HostRouter` interface of the `Speedes-Server` application. The location (i.e. machine name) of the `SpeedesServer` is specified in file `speedes.par`. Section C.3 describes the necessary configurations for the `SpeedesServer`. For this example, use a `speedes.par` with the `SpeedesServer` section missing. This will default the `SpeedesServer` to be be located on local host. Users must start the `SpeedesServer` application manually on the command line.

The most simple possible external module is then shown below in Example 12.3.

```
#include "SpDataParser.H"
#include "SpStateMgr.H"

int main(int argc, char** argv) {
  double       timeLag = 10.0;
  SpDataParser speedesDotPar("speedes.par");
  SpStateMgr   stateMgr(&speedesDotPar, timeLag);
}
```

Example 12.3: Simple External Module

The first argument to the `SpStateMgr` constructor is a pointer to a `SpDataParser`, which is opened to `speedes.par`. The second argument is `timeLag`. This argument specified the maximum amount of time the external module is allowed to fall behind, or lag, the simulation. This value is required to be positive and, in general, large values for `timeLag` are best if you do not want the external module to slow down the advancment of GVT in the simulation.

Now, all this external module does is connect to the simulation and then immediately disconnect, which is exceptionally uninteresting. In order to do something interesting, the external module needs to receive data from the simulation and advance its own time. There are four state manager methods used for subscribing to data within the simulation. These are shown below:

```
class SpStateMgr {
  public:
    SubscribeAll();
    SubscribeType(char* type, ...);
    SubscribeObject(char* objectName, ...);
    SubscribeData(char* name, char* type);
};
```

- `SubscribeAll`:
  all objects in the simulation. This should be used with caution as it can result in excessive amounts of data being sent to the external module.

- `SubscribeType`:
  Subscribes the external module to the object proxies of all objects passed into this method. The list should be `NULL` terminated. The names that are passed into this method should be the name of the object plugged in appended with the string "`_MGR`". For example, if `S_Ship` is plugged in, passing in `S_Ship_MGR` will subscribe the external module to the proxies of all ships.

- `SubscribeObject`:
  Subscribes the external module to the object proxies of the objects passed in on the command line. This is the most specific form of object proxy subscription and results in the least amount of data being transmitted. The list of objects should be `NULL` terminated or undefined behavior will occur.

- `SubscribeData`:
  Subscribes to the "named data" with the string name for all objects with the given type. Objects within the simulation should determine if any external modules are subscribed to a given string before attempting to send this data out, in order to reduce overhead.

The external module is a discrete-event simulator, just like the main SPEEDES framework, and work is performed by writing code to respond to messages. The first code we will add to our external module is a response to changes in object proxies.

```
class MyReflect : public SpStateMgrEvent {
  public:
    MyReflect() {}
    void Process() {
      cout << "MyReflect::Process" << endl;
    }
}

void *NewMyReflect(int n) {
  return new MyReflect[n];
}
```

Notice that this code declares a function that allocates new `MyReflect` classes. This is used in the state manager event registration process.

Once we have designed our processing classes, we need to register this class with the state manager. When we register our class, we must register it with a string handle. There are two predefined SPEEDES handles (i.e. `DISCOVER_OBJECT` and `REFLECT_ATTRIBUTES`) and the user can define their own handles for names of events they may receive from the SPEEDES application or for locally scheduled events. Definitions for these handles are:

- `DISCOVER_OBJECT` - When an object in a SPEEDES simulation is subscribed to, the first event the external module will receive will be a `DISCOVER_OBJECT` event. However, since most simulation objects are constructed at simulation start-up and not when the object is used the first time, the user will receive these events when they first subscribe. If the user subscribes during simulation initialization, then all of their discovered objects will have a time stamp of $-\infty$ on them. If the user subscribes after simulation startup, the time stamp will be at connection time.
- `REFLECT_ATTRIBUTES` - For any SPEEDES object that the user has subscribed to whose proxy was changed, object proxy data will be sent to the external module for processing at the appropriate logical time.
- User-Defined Events - These events are local state manager events that are executed at the appropriate logical time.

The following code shows how we can register our new class `MyReflect` with the state manager.

```
/*
 * Number of local events defined in the state manager
 */
stateManager.DefineNumberOfEvents(1);
```

```
  stateManager.DefineEvent(
                 "REFLECT_ATTRIBUTES",    // Event name
                 0,                       // Unique integer
                 NewMyReflect,            // Function to "new" this class
                 sizeof(MyReflect),       // Size of event class
                 100);                    // Initial number of items on
                                          //  free list
```

Now, any message named REFLECT_ATTRIBUTES received from the SPEEDES application will execute the process method.

The previous steps show how to make a connection with a SPEEDES application, subscribe to SPEEDES objects, and define a local event class to process incoming messages. These are the basic initialization steps. All that is left to do now is to start advancing time and let the local events process the data we receive for which we have registered. The basic format for doing this is to enter a loop in which local time is advanced by a fixed step size. The following code example illustrates the essential procedure:

```
  stateManager.GoToTime(0.0);
  while (stateManager.SpeedesExecuting() == 1) {
    stateManager.GoToTime(stateManager.GetCurrentTime() + 100.0);
  }
```

The method GoToTime is a blocking call. For example a call to GoToTime instructs SPEEDES to advance GVT to the requested time plus time lag. For instance, if the input time were 300.0, then SPEEDES will set up a barrier at 310.0 (assuming timeLag = 10.0, as in main above). Once GVT advances to a value greater than or equal to the requested time, all of the received messages are processed by the appropriate local event that was registered with the state manager in logical time order. Any user-defined events are also processed. After all events are processed, the processing leaves this method (only to be reentered in our example). Examples 12.4 and 12.5 show the above code for an external module.

```
// MyReflect.H
#ifndef MyReflect_H
#define MyReflect_H

#include "SpStateMgrEvent.H"

class MyReflect: public SpStateMgrEvent {
  public:
    MyReflect() {}
    virtual ~MyReflect() {}
    virtual void Process() {
      cout << "MyReflect Process at " << GetTimeTag()
           << ", Global Id: " << GetSimObjGlobalId()
           << ", Event Name: " << GetEventName() << endl;
    }
};

static void* NewMyReflect(int n) {
  return new MyReflect[n];
}
#endif
```

Example 12.4: External Module Proxy Processing Class

```
// Main.C (Example #1)
#include "SpIostream.H"
#include "SpStateMgr.H"
#include "SpDataParser.H"
#include "SpFreeObjProxy.H"
#include "MyReflect.H"

SpFreeObjProxy::SpFreeObjProxy(int n) {set_ntypes(n);}

enum {
  REFLECT_ATTRIBUTES_ID,
  NUMBER_OF_EVENTS
};

int main(int argc, char** argv) {
  double        timeLag      = 10.0;
  SpDataParser  speedesDotPar("speedes.par");
  SpStateMgr    stateManager(&speedesDotPar, timeLag);

  stateManager.SubscribeType("S_Submarine_MGR", NULL);
  /*
   * Number of local events defined in the state manager
   */
  stateManager.DefineNumberOfEvents(NUMBER_OF_EVENTS);

  stateManager.DefineEvent(
                 "REFLECT_ATTRIBUTES",  // Event name
                 REFLECT_ATTRIBUTES_ID, // Unique integer
                 NewMyReflect,          // Function to new event class
                 sizeof(MyReflect),     // Size of event class
                 100);                  // Initial number of items on
                                        //  free list
  stateManager.GoToTime(0.0);

  while (stateManager.SpeedesExecuting() == 1) {
    cout << "externalHostUser->SpeedesExecuting"
         << ", CurrentTime= " << stateManager.GetCurrentTime()
         << ", GrantedTime= " << stateManager.GetGrantedTime()
         << endl;
    stateManager.GoToTime(stateManager.GetCurrentTime() + 100.0);
  }
}
```

Example 12.5: External Module main (Example #1)

In addition to this code, you must also provide the constructor for class `SpFreeObjProxy` used in the original SPEEDES application. In this case, the constructor was the minimal constructor written in `main` in Example 11.8. For this reason, we added the same constructor to our `main`.

## 12.2 Sending and Receiving Messages (Non-Proxy)

In the previous section, we discussed connecting with the SPEEDES framework, retrieving proxies, and advancing time. There can be instances where data is needed by the external module which is not in proxy form, or perhaps the external module has some data that needs to be relayed into the simulation.

Additional state manager functionality provides exactly these types of features. Similarly, sometimes the external module needs to affect the simulation. We will look at this situation first. There currently exist two methods for sending data into SPEEDES. These are methods `SpStateMgr::SendCommand` and `SpEmHostUser::ScheduleEvent`. Each of these methods contains two versions which allow users to send messages into SPEEDES simulation objects via global id or object instance name. Of course, with either method, there must exist an event inside the simulation which can process the message. When `SendCommand` is used, the processing simulation event has access to the external id, which allows the simulation to respond directly to the external module, if necessary. The disadvantage of `SendCommand` is that the receiving application event is executed at the current GVT, which can cause rollbacks. Method `ScheduleEvent` allows you to schedule an event in the SPEEDES application at a specific time, which allows you to minimize rollbacks by scheduling events in the future. The API for the `SpStateMgr` method `SendCommand` and the `SpEmhostUser` method `ScheduleEvent` is shown below:

```
int SendCommand(char* commandName,
                int   simObjGlobalId,
                char* message,
                int   bytes)
```

| Parameter | Description |
|---|---|
| commandName | Event name to be scheduled. |
| simObjGlobalId | Global id of object for this schedule. |
| message | Any data for the event. |
| bytes | Length of the message. |

Table 12.1: External Module Sendcommand API

```
SCHEDULE_EVENT_REPLY_MESSAGE*
ScheduleEvent(SpSimTime time,
              int       globalId,
              char*     eventName,
              char*     msg,
              int       msgBytes,
              char*     data,
              int       dataBytes,
              TimeMode  tmMode = IF_IN_PAST_IGNORE,
              int       cancelHandleReq = 1)
```

| Parameter | Description |
|---|---|
| time | Time at which event will be scheduled. |
| globalId | Global id for which this event is scheduled. |
| eventName | Name of event to be scheduled. |
| msg | The message for the event |
| msgBytes | The size of the message class. |
| data | Any additional data for this event. |
| dataBytes | The number of bytes of data. |
| tmMode | If it is equal to `IF_IN_PAST_IGNORE`, then the event will be ignored if it arrives in the past. If it is equal to the `IF_IN_PAST_SCHEDULE_AT_GVT`, then the event will be scheduled at GVT if the time it is scheduled for is before GVT. |
| cancelHandleReq | If equal to `1`, a cancel handle is returned. Otherwise, `NULL` is returned. |

Table 12.2: External Module ScheduleEvent API

The format and content of each are self-explanatory. Notice that parameter `msg` in method `ScheduleEvent` is actually a `SpMsg` (i.e. a member of the message or "`M_`" class). Prior to the implementation of the unified API for events and methods, users of SPEEDES were required to provide this class.

However, now this class is auto-generated by the macro `DEFINE_SIMOBJ_EVENT` (see Chapter 6). Therefore, for the external module, users must instantiate an empty `SpMsg` class for this parameter which is currently ignored. Due to the use of hidden `M_` classes, only zero argument event methods can currently be scheduled using this method.

We will now extend our previous external module example to use these two methods. The following two code examples show how to use methods `SendCommand` and `ScheduleEvent` for sending messages into our ship and submarine simulation:

```
stateManager.SendCommand("SubAttack", "S_Submarine_MGR 0",
                         "S_Ship_MGR 1", strlen("S_Ship_MGR 1") + 1);

SpMsg theM_Underscore;
int   globalId =
  stateManager.GetEmHostUser()->NameLookup("S_Submarine_MGR 1");
stateManager.GetEmHostUser()->
  ScheduleEvent(stateManager->GetCurrentTime() + 17.0, globalId,
                "SubAttack", (char *) &theM_Underscore, sizeof(SpMsg),
                "S_Ship_MGR 1", strlen("S_Ship_MGR 1") + 1);
                IF_IN_PAST_SCHEDULE_AT_GVT);
```

Each of these methods cause the event `SubAttack` to be executed in the simulation and instructs the submarine to attack the object with the name "`S_Ship_MGR 1`". The call to `SendCommand` schedules event `SubAttack` on the simulation object whose global id is 0. The event is executed as soon as it is received, thus, rolling back all events on that object. `ScheduleEvent` schedules event `SubAttack` on the simulation object whose global id is 1 at the current external module time, plus 17.0 seconds.

We have just described how to send data into the simulation, but suppose a simulation object had simulation data that is needed by the external module. Can we send the data to the external module? Yes, by using either method `SpHostUser::RB_SendSubscribedData` or method `SpHost-User::RB_SendNamedData`. The method called determines how the external module receives the associated data. The method `RB_SendSubscribedData` requires you to register an event with the state manager similar to the way `REFLECT_ATTRIBUTES` was done previously for proxy events. In order to receive SPEEDES messages, the first thing to be done is to create an object which inherits from class `SpStateMgrEvent`. The class you make must contain the virtual method `Process`, which is where the code for processing the received data resides.

We will modify the submarine model once again so that it will notify the external modules of the proxies that it has received. We will modify the methods `S_Submarine::DiscoverProxy` and `S_Submarine::UnDiscoverProxy`, as shown in Example 12.6.

```
#include "S_Submarine.H"
#include "F_SpProxyItem.H"

void S_Submarine::DiscoverProxy() {
  ++NumProxiesSem;
  if (CheckSubscription("SubDiscoverProxy")) {
    F_SpProxyItem* proxyItem = (F_SpProxyItem*)SpGetMsgData();
    SpObjProxy* discoveredObject = proxyItem->GetObjProxy();
    char* proxyName = discoveredObject->GetProxyName();
    SpGetHostUser()->RB_SendSubscribedData(
                       SpGetTime(), "SubDiscoverProxy",
                       proxyName, strlen(proxyName) + 1,
                        SpGetSimObjGlobalId());
  }
}

void S_Submarine::UnDiscoverProxy() {
  --NumProxiesSem;
  if (CheckSubscription("SubUnDiscoverProxy")) {
    F_SpProxyItem* proxyItem = (F_SpProxyItem*)SpGetMsgData();
    SpObjProxy* discoveredObject = proxyItem->GetObjProxy();
    char* proxyName = discoveredObject->GetProxyName();
    SpGetHostUser()->RB_SendSubscribedData(
                        SpGetTime(), "SubUnDiscoverProxy",
                        proxyName, strlen(proxyName) + 1,
                        SpGetSimObjGlobalId());
  }
}
```

Example 12.6: New Discover and UnDiscover Proxy Methods for the Submarine

We now need to create two state manager events that respond to this named data. These are similar to the code used to respond to the reflects we used earlier and are shown in Example 12.7.

```
// DiscoverUndiscover.H
#ifndef DiscoverUndiscover_H
#define DiscoverUndiscover_H
#include "SpStateMgrEvent.H"

class DiscoverShip : public SpStateMgrEvent {
  public:
    DiscoverShip() {}
    virtual ~DiscoverShip() {}

    virtual void Process() {
      cout << "Submarine is discovering a ship named " << GetData()
           << " at time " << GetTimeTag() << endl;
    }
};

class UndiscoverShip : public SpStateMgrEvent {
  public:
    UndiscoverShip() {}
    virtual ~UndiscoverShip() {}

    virtual void Process() {
      cout << "Submarine is losing track of a ship named " << GetData()
           << " at time " << GetTimeTag() << endl;
    }
};
static void* newDiscoverShip(int n){return new DiscoverShip[n];}
static void* newUndiscoverShip(int n){return new UndiscoverShip[n];}
#endif
```

Example 12.7: External Module User-Defined Events

After the class has been created, the event name which the simulation is using to send information out to the external module must be registered with the state manager. For example:

```
  stateManager.DefineNumberOfEvents(3);
  stateManager.DefineEvent(
               "SubDiscoverProxy",     // Event name
               1,                      // Unique integer
               NewDiscoverShip,        // Function to new event class
               sizeof(DiscoverShip),   // Size of event class
               100);                   // Initial number of items on
                                       //  free list
  stateManager.DefineEvent(
               "SubUnDiscoverProxy",   // Event name
               2,                      // Unique integer
               NewUndiscoverShip,      // Function to new event class
               sizeof(UndiscoverShip), // Size of event class
               100);                   // Initial number of items on
                                       //  free list
```

Notice that input argument in method `DefineNumberOfEvents` is `3` due to the external module application now having three events to process (i.e. event `REFLECT_ATTRIBUTES` for proxy updates was registered in the previous example). This is also why the second argument in the `DefineEvent` methods are set to `1` and `2` respectively since `0` has been taken by event `REFLECT_ATTRIBUTES` above.

Example 12.8 shows the modifications made to our original external module with the additional features added in. This includes the response to subscribed data “SubDiscoverProxy” and “SubUndiscoverProxy”, with the added feature that every discovered ship will be fired upon using the `ScheduleEvent` method call.

```
// Main.C (Example #2)
#include "SpIostream.H"
#include "SpStateMgr.H"
#include "SpDataParser.H"
#include "SpStateMgrEvent.H"
#include "SpMsg.H"
#include "SpFreeObjProxy.H"

SpFreeObjProxy::SpFreeObjProxy(int n) {set_ntypes(n);}

class MyReflect: public SpStateMgrEvent {
  public:
    MyReflect() {}
    virtual ~MyReflect() {}
    virtual void Process() {
      cout << "MyReflect Process at " << GetTimeTag()
           << ", Global Id: " << GetSimObjGlobalId()
           << ", Event Name: " << GetEventName() << endl;
    }
};

class DiscoverShip : public SpStateMgrEvent {
  public:
    DiscoverShip() {}
    virtual ~DiscoverShip() {}
    virtual void Process() {
      cout << "Submarine is discovering a ship named " << GetData()
           << " at time " << GetTimeTag() << endl;
      SpMsg theM_Underscore;
      int   globalId = GetSimObjGlobalId();
      StateMgr->GetEmHostUser()->
        ScheduleEvent(StateMgr->GetCurrentTime() + 17.0, globalId,
                      "SubAttack", (char *) &theM_Underscore,
                      sizeof(SpMsg), GetData(),
                      strlen(GetData()) + 1,
                      IF_IN_PAST_SCHEDULE_AT_GVT);
    }
};

class UndiscoverShip : public SpStateMgrEvent {
  public:
    UndiscoverShip() {}
    virtual ~UndiscoverShip() {}
    virtual void Process() {
      cout << "Submarine is losing track of a ship named " << GetData()
           << " at time " << GetTimeTag() << endl;
    }
};

static void *NewMyReflect(int n){return new MyReflect[n];}
static void* NewDiscoverShip(int n){return new DiscoverShip[n];}
static void* NewUndiscoverShip(int n){return new UndiscoverShip[n];}

enum {
  REFLECT_ATTRIBUTES_ID,
  SUB_DISCOVER_PROXY_ID,
  SUB_UNDISCOVER_PROXY_ID,
  NUMBER_OF_EVENTS
};

int main(int argc, char** argv) {
  double        timeLag      = 10.0;
  SpDataParser  speedeDotPar("speedes.par");
  SpStateMgr    stateManager(&speedeDotPar, timeLag);

  stateManager.SubscribeType("S_Submarine_MGR", NULL);
  /*
   * Number of local events defined in the state manager
   */
  stateManager.DefineNumberOfEvents(NUMBER_OF_EVENTS);

  stateManager.DefineEvent(
                 "REFLECT_ATTRIBUTES",  // Event name
                 REFLECT_ATTRIBUTES_ID, // Unique integer
                 NewMyReflect,          // Function to new event class
                 sizeof(MyReflect),     // Size of event class
                 100);                  // Initial number of items on
                                        //  free list
  stateManager.DefineEvent(
                 "SubDiscoverProxy",     // Event name
                 SUB_DISCOVER_PROXY_ID,  // Unique integer
                 NewDiscoverShip,        // Function to new event class
                 sizeof(DiscoverShip),   // Size of event class
                 100);                   // Initial number of items on
```

```
                                         //  free list
  stateManager.DefineEvent(
                 "SubUnDiscoverProxy",   // Event name
                 SUB_UNDISCOVER_PROXY_ID,// Unique integer
                 NewUndiscoverShip,      // Function to new event class
                 sizeof(UndiscoverShip), // Size of event class
                 100);                   // Initial number of items on
                                         //  free list
  stateManager.GoToTime(0.0);

  stateManager.SubscribeData("SubDiscoverProxy",   "S_Submarine_MGR");
  stateManager.SubscribeData("SubUnDiscoverProxy", "S_Submarine_MGR");

  while (stateManager.SpeedesExecuting() == 1) {
    cout << "externalHostUser->SpeedesExecuting"
         << ", CurrentTime= " << stateManager.GetCurrentTime()
         << ", GrantedTime= " << stateManager.GetGrantedTime()
         << endl;
    stateManager.GoToTime(stateManager.GetCurrentTime() + 100.0);
  }
}
```

Example 12.8: External Module main (Example #2)

## 12.3 Local Events

In addition to responding to external messages from the simulation, an external module can also schedule events of its own. To illustrate this, we will create an event called `AttackShip` and have this event scheduled from the `DiscoverShip` event. Just like all other events, this event needs to be integrated with the free lists in order to optimize the use of available resources. In Example 12.8, the `class DiscoverShip` and the enum of event ids will change, as shown in Example 12.9.

```
#include "SpStateMgr.H"
#include "SpStateMgrEvent.H"

enum {
  REFLECT_ATTRIBUTES_ID,
  SUB_DISCOVER_PROXY_ID,
  SUB_UNDISCOVER_PROXY_ID,
  ATTACK_SHIP_ID,
  NUMBER_OF_EVENTS
};

class DiscoverShip : public SpStateMgrEvent{
  public:
    DiscoverShip() {}
    virtual ~DiscoverShip() {}
    virtual void Process() {
      cout << "Submarine is discovering a ship named " << GetData()
           << " at time " <<GetTimeTag() << endl;
      AttackShip* attackShipPtr = (AttackShip *)
        StateMgr->GetFreeUserEvents().new_event(ATTACK_SHIP_ID);
      attackShipPtr->Init(GetData());
      // Attack in 2 seconds.  Local Event
      ScheduleEvent(attackShipPtr, GetTimeTag() + 2.0);
    }
};

class AttackShip : public SpStateMgrEvent {
  public:
    AttackShip() : ShipToAttack(NULL) {}
    ˜AttackShip() {}
    void Init(char* shipToAttack) {
      free(ShipToAttack);
      ShipToAttack = strdup(shipToAttack);
    }

    virtual void Process() {
      SpMsg theM_Underscore;
      int   globalId =
        StateMgr->GetEmHostUser()->NameLookup(GetData());
      StateMgr->GetEmHostUser()->ScheduleEvent(
                      StateMgr->GetCurrentTime() + 17.0, globalId,
                      "SubAttack", (char *) &theM_Underscore,
                      sizeof(SpMsg),"S_Ship_MGR 1",
                      strlen("S_Ship_MGR 1") + 1,
                      IF_IN_PAST_SCHEDULE_AT_GVT);}
  private:
    char* ShipToAttack;
};

void* NewAttackShip(int n){return new AttackShip[n];}
```

Example 12.9: External Module Local Events

Of course, as with other events, the local event needs to be plugged into the state manager as given in the following example:

```
  stateManager.DefineEvent(
               "AttackShip",            // Event name
               ATTACK_SHIP_ID,          // Unique integer
               NewAttackShip,           // Function to new event class
               sizeof(AttackShip),      // Size of event class
               100);                    // Initial number of items on
                                        //  free list
```

## 12.4 Record and Playback

The state manager has one additional useful feature, which is the capability to record all messages received from the SPEEDES simulation and record or save these messages for later playback. To enable this capability, users must use state manager method `RecordInputMessages`. Once data has been saved to a file, the data can be played back by constructing a state manager using the data file's file name and writing the rest of the external module. For example, in the previous examples you could replace the state manager instantiation shown in the examples with a state manager instantiation with the data file's file name. The procedure for modifying one of the previous examples is:

1. Add a call to method `RecordInputMessages` right after you create the state manager.

2. Execute the simulation and external module.

3. Replace the original state manager instantiation with the new one and delete method `Record_Input_Messages`.

4. Replace the `SpStateMgr` constructor with the one that takes a file name as an argument and pass the name of the saved file to the constructor.

5. Run the external module

## 12.5 Optimizing Memory Use

The `SpStateMgr` has the ability to go both backwards and forwards in time through the `GoToTime`. Many situations do not require these capabilities such as gateways or other external modules that only move forward in time. In these sorts of situations, the memory usage of the external module can be improved by calling the method `DisableRollbackSupport`. By calling this method before any `GoToTime` method calls are made, the memory footprint of the `SpStateMgr` will be reduced.

## 12.6 Tips Tricks and Potholes

- Chosing a small value for time lag will guarantee that the simulation will not get far ahead of the external module but may result in causing the simulation to lag. In general, choose a large value for time lag. If the external module can run faster than the simulation, it will never end up lagging the simulation significantly.

- As stated in the introduction, many features do not work when running in sequential mode. The following features do not work in sequential mode:

  - Time lag for the `SpStateMgr` is ignored.
  - Pauses and resumes do not work.
  - Creating or removing barriers does not work.
  - Method `GoToTime` does not work.

# Chapter 13

# Command-Line Utilities

SPEEDES contains many command-line utilities/tools. These tools allow the user to perform such actions as pausing or resuming a simulation, locking the simulation to real-time, examining object names, etc. A prerequisite for the usage of any of these tools is that `SpeedesServer` must be running. The following sections describe the usage for each command-line utility.

Note that most of the features described in this chapter do not work in sequential mode. When running on one node, be sure that `optimize_sequential` is set to false in `speedes.par`. See Appendix C.1 for more information. The one exception is that utility `SpFilterTrace` does not depend on the `SpeedesServer`.

## 13.1 Querying Object By Names and Types

Name:

```
SpObjNames
SpObjType
```

Synopsis:

```
SpObjNames [-g id] [Object_Type_Name]
SpObjType [-g id] [Object_Type_Name]
```

Description:
`SpObjNames` queries SPEEDES for a list of object names and their respective global ids. `Object_Type_Name` is an optional parameter that, if given, will output all objects of that type and their respective global ids. If `Object_Type_Name` is not specified, then all object types with their respective global id are output to the terminal. Option `-g id` can be used to specify the group so that one `SpeedesServer` can be used with multiple simulations.

`SpObjType` queries SPEEDES for a list of object types and their respective type ids. `Object_Type_Name` is an optional parameter that, if given, will query SPEEDES for the specified type name. If `Object_Type_Name` is not specified, then all object types and their respective type id's are output to the terminal. Option `-g id` can be used to specify the group so that one `SpeedesServer` can be used with multiple simulations.

## 13.2 Query

Name:

```
SpQuery
```

Synopsis:

```
SpQuery Object_Name [Query_Name]
```

Description:
`SpQuery` queries SPEEDES for information about the simulation object. The information returned is dependent on whether or not the virtual methods `Query` and `NamedQuery` in class `SpSimObj` are defined by the user's simulation object or the default `SpSimObj` implementation. When `Query` is used without `Query_Name`, then the virtual method `Query` is executed. If optional parameter `Query_Name` is specified, then the virtual method `NamedQuery` is executed. If the virtual methods are not defined, then SPEEDES returns the default data of Global Id, Query Time and the Number of Events for the queried object. If the virtual methods are defined by the user, then `SpQuery` outputs the user-defined data.

## 13.3 Time

Name:

```
SpTime
```

Synopsis:

```
SpTime [lag]
```

Description:
`SpTime` queries SPEEDES for the current simulation time (i.e. GVT). If lag is specified, the `SpTime` queries SPEEDES for the current time using lag as an input value for the call to SPEEDES. This has the effect of setting a barrier up in SPEEDES at the current GVT plus lag. Therefore, for large input lags, this utility does not inhibit GVT advancement.

## 13.4 Changing Lock To Wall Clock Scaler

Name:

```
SpChangeScaler
```

Synopsis:

```
SpChangeScaler scaler
```

Description:
`SpChangeScaler` changes the rate of time advancement for SPEEDES when the parameter `scaled-_time` in `speedes.par` is set to true. For example, `SpChangeScaler 50` changes the SPEEDES time advancement rate to 50 times wall clock, provided that the simulation can run that fast.

## 13.5 Scheduling and Canceling Events

Name:

```
SpScheduleEvent
SpCommand
SpCancelEvent
```

Synopsis:

```
SpScheduleEvent Time Object_Name Event_Name [User_Data]
SpCommand Command_Name Object_Name Data_String
SpCancelEvent CancelHandle
```

Description:
`SpScheduleEvent` is used to schedule pre-defined events on a SPEEDES simulation object. The user supplies the time that the event should be executed at, the name of the object that the event will run on, and the name of the event to be executed. The user can supply user-defined data (`char *`) as an optional fourth input parameter. The insertion of an event by this utility can fail in several ways, including:

- Specified time is in the simulation past (earlier than current GVT).
- Object specified is not a valid object.
- Event name is incorrect (silent error).

Upon successful event insertion, `SpScheduleEvent` will return a cancel handle that can be used by the `SpScheduleEvent` counterpart `SpCancelEvent` to cancel the event.

`SpCommand` also can be used to schedule events or event handlers, but this utility does not return cancel handles. The input parameter `Data_String` must be a plain text string. Actual SPEEDES events data can be composed of complex data types. For these types of events this utility cannot be used.

## 13.6 Pause and Resume

Name:

```
SpPause
SpResume
```

Synopsis:

```
SpPause [Named_Of_Pause [Time_Of_Pause]]
SpResume [Named_Pause]
```

Description:
Use `SpPause` to “pause” the simulation by setting up a barrier inside the simulation, thus preventing GVT from advancing. If the utility is used without an input parameter, then the simulation will pause immediately. Using the `SpResume` command with no parameters causes the pause set with `SpPause` to be removed. `SpPause` can be supplied with optional parameters. The first optional parameter is the “name” of a pause (i.e. `Name_Of_Pause`). If a name is given to a pause, it is known as a “Named

Pause". A Named Pause behaves exactly like a normal pause in that the simulation pauses once a Named Pause is received. However, the only way a Named Pause can be removed is with utility `SpResume` using the same Named Pause. If a Named Pause is used, then an optional time parameter can be supplied, which causes the simulation to pause at the specified time. If the time given is before GVT, then the simulation will pause immediately.

`SpResume` is the counterpart to `SpPause`. `SpResume` removes a Named Pause or unnamed pause created by `SpPause`. `SpResume` can also be used to remove pauses that were created in section `NamedPauses` in file `speedes.par`.

## 13.7 Simulation Time Controller

Name:

```
SpForceToWallClock
```

Synopsis:

```
SpForceToWallClock -m num -r num [-from_start]
```

Description:
`SpForceToWallClock` utility allows the user to lock time in their simulation to some multiple of wall clock. The mandatory option is `-m`, which specifies what rate GVT will run at relative to wall clock time. For example, if a user chooses 4, then `SpForceToWallClock` will attempt to keep GVT at 4 times wall clock time (i.e. for every 1 second of elapsed wall clock time GVT will advance by 4 seconds). Option `-r` specifies the frequency at which `SpForceToWallClock` will verify GVT time. For example, a `-r` setting of 0.5 seconds specifies that `SpForceToWallClock` will check simulation time every 0.5 seconds and calculate GVT time errors relative to wall clock time. If an error exists, then the appropriate time correction is applied.

Option `-from_start` specifies that simulation GVT advancement will be passed until elapsed wall clock time is equal to GVT. For example, if the simulation GVT is at 20.0 when `SpForceToWall-Clock` starts with option `-from_start`, then GVT pauses for 20.0 seconds (assuming that the `-m` option is 1.0 seconds).

## 13.8 Killing Simulations

Name:

```
SpKillSim
```

Synopsis:

```
SpKillSim
```

Description:
`SpKillSim` kills all of the simulations currently running.

## 13.9 Sorted Output

Name:

`SpSortedOutput`

Synopsis:

```
SpSortedOutput [-q] [-f filename] [-l lag] stream_name
```

Description:
Simulations that use `RB_exostream` can display data generated on multiple nodes in a sorted fashion using utility `SpSortedOutput`. `RB_exostream` constructors are given plain text names inside of the simulation and by specifying this name (`stream_name`), then this utility will display all data written to this stream sorted by time. Option `-q` prevents output from being displayed on the terminal. Output can be saved to a file using the `-f` option. Option `-q` and `-f` can be used in conjunction to save binary data. Option `-l` is used to specify the amount of time, in seconds, that this utility can lag behind the SPEEDES simulation.

## 13.10 Trace File Filtering

Name:

`SpFilterTrace`

Synopsis:

```
SpFilterTrace Trace_File_Name String...
```

Description:
SPEEDES has an event trace capability which records event processing statistics to a file for later use (e.g. simulation debug, post simulation event analysis, etc.). The amount of data in the trace file can become quite large, which can make the trace data difficult to analyze. Utility `SpFilterTrace` can filter the section blocks in the trace file based on user input strings. `SpFilterTrace` input parameters are the `Trace_File_Name` and one or more input strings for `FilterTrace` to search the trace file for. Any data found in the trace file that matches the input search strings are output to the terminal. To create a trace file during a simulation run, set the parameter `trace` to `T` in section `trace` in file `speedes.par`.

Section 17.2 shows some example `SpFilterTrace` usage commands.

# Part VI

# Advanced Topics

# Chapter 14

# Simulation Objects

## 14.1 Dynamic Objects

SPEEDES allows simulation objects to be created dynamically during simulation execution. The API provides an interface for defining events that will create and initialize a dynamically created simulation object. A simulation object may define an unlimited number of dynamic initialization methods with the following API macro call found in `SpDefineEvent.H`:

```
DEFINE_CREATE_EVENT_<numParam>_ARG(eventName,
                                   className,
                                   methodName,
                                   [paramList])
```

| Parameter | Description |
|---|---|
| numParam | The number of parameters used in the method being converted into an event (valid range is 0 to 8). |
| eventName | Any user-defined string representing the name of the creation event (legal characters for string names include alphanumeric and underscore characters). |
| className | The name of the simulation object class that will be dynamically created. |
| methodName | The name of the method that will be called upon the construction of this object. |
| paramList | Comma delimited list of the parameter types found in the method. |

Table 14.1: Macro DEFINE_CREATE_EVENT API

As with other SPEEDES events, any event created by macro `DEFINE_CREATE_EVENT` must be plugged into `main`.

A byproduct of the above macro is a schedule function that is nearly identical to the schedule function for a standard event. The one difference is that the `SpObjHandle` parameter is not constant. This is because the schedule function will determine what the object handle of the newly created object has to be. Once it is determined, this object will be created and the handle of the newly created object is returned in the `objHandle` parameter.

```
SpCancelHandle
SCHEDULE_<eventName>(const SpSimTime simTime,
                     SpObjHandle&    objHandle,
                                     [paramList],
                     const char*     data = NULL,
                     int             dataBytes = 0)
```

| Parameter | Description |
|---|---|
| eventName | This is the same name used when the event was defined. |
| simTime | This parameter specifies the time at which the event will be executed. The time scheduled can be the present time or a future time, but not a time in the past. |
| objHandle | This parameter is filled out by the schedule function with the object handle of the newly created object. See Section 3.3 for additional information on object handles. |
| paramList | This is a comma delimited list of the types of the parameters that are to be passed to the simulation object method. |
| data | This optional parameter allows users to send data to the receiving event for further processing. The data can be binary or character stream data. If the data contains pointers, make sure to write "wrap" and "unwrap" functions to enable packing and unpacking a buffer that represents the data. |
| dataBytes | This parameter represents the size, in bytes, of the buffer sent as the "data" parameter. If you do not use the "data" parameter, then there is no need to use this parameter either. |

Table 14.2: Function SCHEDULE API for Dynamically Created Objects

The following example illustrates how to use the dynamic object creation functionality. The example consists of two simulation object types. The code shown in Example 14.1 and 14.2 shows an object that will be dynamically created. The code shown in Example 14.3 shows an object that is created using the traditional approach (static definition).

```
// S_DynamicObject.H
#ifndef S_DynamicObject_H
#define S_DynamicObject_H

#include "SpSimObj.H"
#include "SpDefineSimObj.H"
#include "SpDefineEvent.H"

class S_DynamicObject: public SpSimObj {
  public:
    S_DynamicObject() {}
    void Init() {}
    void DynamicObjectInit(int initVar);
    void PrintName();
};

DEFINE_SIMOBJ(S_DynamicObject, 0, SCATTER);
DEFINE_CREATE_EVENT_1_ARG(DynamicObject_DynamicObjectInit,
                          S_DynamicObject, DynamicObjectInit, int);
DEFINE_SIMOBJ_EVENT_0_ARG(DynamicObject_PrintName,
                          S_DynamicObject, PrintName);
#endif
```

Example 14.1: Dynamic Simulation Object Creation Definition

```
// S_DynamicObject.C
#include "RB_ostream.H"
#include "SpMainPlugIn.H"

#include "S_DynamicObject.H"

void S_DynamicObject::DynamicObjectInit(int initVar) {
  char name[80];

  sprintf(name, "Dyn Obj Number %d", initVar);
  SetName(name);
  RB_cout << "Dynamic Object whose name is \""
          << name << "\" created at " << SpGetTime()
          << endl;
}

void S_DynamicObject::PrintName() {
  RB_cout << "Dynamic Object is: \"" << GetName() << "\"" << endl;
}

void PlugInDynamicObject() {
  PLUG_IN_SIMOBJ(S_DynamicObject);
  PLUG_IN_EVENT(DynamicObject_DynamicObjectInit);
  PLUG_IN_EVENT(DynamicObject_PrintName);
}
```

Example 14.2: Dynamic Simulation Object Creation Implementation

The definition file uses the new macro DEFINE_CREATE_EVENT. This will now allow for simulation object S_DynamicObject to be dynamically created by using function SCHEDULE_DynamicObject_DynamicObjectInit. Notice that, even though these objects are dynamically created, the macro DEFINE_SIMOBJ must be used with its quantity specifier as zero. A non-zero value could have been used here as well. The result of this would have been the creation of the specified amount of simulation objects (i.e. S_DynamicObject) during simulation initialization. Dynamically created objects could still be created, thus simulation objects can be designed such that they can be created at initialization and during run time.

Notice that the implementation file specifies a unique name for each dynamically created object. This is necessary if the object handle for the dynamically created object will be needed during the simulation execution (and it will be if events are to be scheduled on the dynamically created object as it will be for this example).

Example 14.3 shows the simulation object that is created at simulation initialization.

```
// S_StaticObject.C
#include "SpSimObj.H"
#include "SpDefineSimObj.H"
#include "SpDefineEvent.H"
#include "SpMainPlugIn.H"
#include "SpGlobalFunctions.H"
#include "SpProc.H"
#include "RB_ostream.H"
#include "I_SpLookupDynSimObjByName.H" // GET_DYN_OBJ_HANDLE

#include "S_DynamicObject.H"

class S_StaticObject: public SpSimObj {
  public:
    void Init();
    void DynamicObjectName(char *name);
    char DynObjName[80];
};
```

```
DEFINE_SIMOBJ(S_StaticObject, 2, SCATTER);
DEFINE_LOCAL_EVENT_1_ARG(StaticObject_DynamicObjectName,
                         S_StaticObject, DynamicObjectName, char*);

void S_StaticObject::Init() {
  SpObjHandle dynamicObjHandle;

  cout << "Static Object Initialization for Object "
       << SpGetSimObjKindId() << endl;
  SpEnableDynSimObjLookup();

  sprintf(DynObjName, "Dyn Obj Number %d", SpGetSimObjKindId());
  SCHEDULE_StaticObject_DynamicObjectName(0.0 + SpGetSimObjKindId(),
                                          *this,
                                          DynObjName);
  SCHEDULE_DynamicObject_DynamicObjectInit(5.0,
                                           dynamicObjHandle,
                                           SpGetSimObjKindId());
  cout << "Static Object # " << SpGetSimObjKindId()
       << " Scheduled creation of object " << dynamicObjHandle << endl;
  SCHEDULE_StaticObject_DynamicObjectName(10.0 + SpGetSimObjKindId(),
                                          *this,
                                          DynObjName);
}

void S_StaticObject::DynamicObjectName(char* name) {
  P_VAR;
  SpObjHandle  lookupObjHandle;
  int          simObjMgrId;
  P_BEGIN(1);

  simObjMgrId = SpGetSimObjMgrId("S_DynamicObject_MGR");

  GET_DYN_OBJ_HANDLE(1, SpGetTime(), simObjMgrId, name,
                     lookupObjHandle);
  RB_cout << "Handle returned by GET_DYN_OBJ_HANDLE = "
          << lookupObjHandle  << " at " << SpGetTime() << endl;

  SpObjHandle nullObjHandle (-1, -1, -1);
  if (nullObjHandle != lookupObjHandle) {
    SCHEDULE_DynamicObject_PrintName(SpGetTime() + 100.0,
                                     lookupObjHandle);
  }

  P_END;
}

void PlugInStaticObject() {
  PLUG_IN_SIMOBJ(S_StaticObject);
  PLUG_IN_EVENT(StaticObject_DynamicObjectName);
}
```

Example 14.3: Static Simulation Object

Each static simulation object will create one `S_DynamicObject` object and schedule event `Dynamic-Object_PrintName` on that object. The `Init` method schedules three events as follows:

1. Event `StaticObject_DynamicObjectName` is scheduled on simulation object `S_StaticObject` $t = 0.0$. This event attempts to schedule event `DynamicObject_PrintName` on the specified dynamically created object. However at $t = 0.0$, no objects have been created yet, therefore the object handle looked up for the dynamic object is $(-1, -1, -1)$.

2. Event `DynamicObject_DynamicObjectInit` is scheduled at $t = 5.0$, which causes the `S_DynamicObject` simulation object to be created.

3. Event `StaticObject_DynamicObjectName` is scheduled again. However, this time the dynamic objects exist. Therefore, event `DynamicObject_PrintName` is scheduled on each dynamic object.

Method `DynamicObjectName` (i.e. event `StaticObject_DynamicObjectName`) is used to schedule an event on the dynamically created object. The first step is to look up the object handle for the dynamically created object. Currently, this requires that the process model be used with the process model construct `GET_DYN_OBJ_HANDLE`. After the object handle is retreived, an event on the dynamic object is scheduled. Recall that the `Init` method scheduled this event before and after the creation of the dynamic object, hence the `if` statement verifying that the object exists prior to scheduling the dynamic object event `DynamicObject_PrintName`.

The API for `GET_DYN_OBJ_HANDLE` is shown below:

```
GET_DYN_OBJ_HANDLE(reentryLabel,
                   simTime,
                   simObjMgrId,
                   dynSimObjName,
                   objHandle)
```

| Parameter | Description |
|---|---|
| reentryLabel | Integer id for the appropriate process model reentry label. For example, if `P_BEGIN` defines the number of reentry labels to be 3, then there should be 3 process model reentry constructs whose labels are 1, 2, and 3. |
| simTime | Specifies the simulation time when processing will continue. |
| simObjMgrId | Id of the simulation object manager that is responsible for the dynamic object being searched for. |
| dynSimObjName | Name of the dynamic simulation object for which the object handle is desired. |
| objHandle | The object handle for the requested dynamic simulation object. If an object is not found, then the handle is set to $(-1, -1, -1)$. |

Table 14.3: Macro GET_DYN_OBJ_HANDLE API

Prior to using `GET_DYN_OBJ_HANDLE`, the function `SpEnableDynSimObjLookup` must have been called.

Finally, the code shown in Example 14.4 completes this example.

```
// Main.C
#include "SpMainPlugIn.H"

void PlugInDynamicObject();
void PlugInStaticObject();

int main (int argc, char **argv) {
  PlugInDynamicObject();
  PlugInStaticObject();

  ExecuteSpeedes(argc, argv);
}
```

Example 14.4: main for Dynamic Object Creation Example

## 14.2 Components

In SPEEDES, a component is a class inheriting from `SpComponent` that can be plugged into and unplugged from a simulation object with simple method calls. This strategy allows the component to automate various tasks that are always associated with plugging in and unplugging the component. Other simulation objects can only schedule events for components via handler events, since they have no way of knowing whether any particular component is currently plugged in.

Components automate one key feature: adding a component's handlers to the simulation object when it is plugged in and removing its handlers when it is unplugged from the simulation object. This coordinates component handler addition and removal with whether their associated components are plugged in, thereby supporting an abstraction barrier between components and other simulation objects that schedule events related to them. That is, since other simulation objects can only schedule events for components via handler events, components can respond to events when plugged in, yet ignore events when unplugged.

### 14.2.1 The Component APIs

To create a component, inherit from `SpComponent`. This class then provides the methods for adding and removing handlers:

```
  void AddHandler      (const SpHandlerId& handlerId,
                              char*        trigger = NULL)
  void SubscribeHandler(const SpHandlerId& handlerId,
                              char*        trigger = NULL)
  void RemoveHandler   (const SpHandlerId& handlerId,
                              char*        trigger = NULL)
```

The parameter definitions for these methods are the same as for those described for the simulation objects (see Table 7.2). SPEEDES stores these handlers in the component until they are plugged into the simulation object, at which point SPEEDES adds the component handlers to the simulation object. When components are removed from the simulation object, SPEEDES automatically removes the component's handlers from the simulation object, but leaves them stored in the component so that the component may be plugged into the simulation object again later. The component handler's add and remove methods may be called at any simulation time, regardless of whether the component is currently plugged into the simulation object.

Handlers may be created as methods on the component, just as handlers are created as methods on any object (see Chapter 7). Handler methods on any object can be dynamically added to and removed from a component at any simulation time. For example:

```
#include "SpComponent.H"
#include "RB_ostream.H"
```

```
class MyComponent : public SpComponent {
  public:
    MyComponent()  {AddHandler(MyHandler_HDR_ID(this), "my trigger");}
    MyHandler()    {RB_cout << "MyHandler invoked" << endl;}
    ˜MyComponent() {RemoveHandler(MyHandler_HDR_ID(this),
                    "my trigger");}
};
DEFINE_HANDLER(MyHandler, MyComponent, MyHandler);
```

Example 14.5: Basic Component Example

Then, to plug components in and remove them from simulation objects, call these methods on the simulation object:

```
  void AddComponent   (SpComponent& component);
  void RemoveComponent(SpComponent& component);
```

For example:

```
#include "MyComponent.H"
#include "S_MySimObj.H"

class S_MySimObj : public SpSimObj {
  public:
    MyComponent Component;
    AddMyComponent()    {AddComponent(Component);}
    RemoveMyComponent() {RemoveComponent(Component);}
};
```

Example 14.6: Adding a Component to a Simulation Object

# Chapter 15

# Autonomous Events

Chapter 6 introduced simulation object events, local events, and autonomous events. Simulation object events and local events hide the internal event execution flow and complexity from the user. This makes these types of events the easiest type to design, use, and maintain. However, there may be instances where users need more control over the event execution, need additional event optimizations, or need to boost simulation run-time performance. In these cases, users should consider using autonomous events, since this type of event give users full access to the internal phases of an event through the event class virtual methods.

Autonomous events differ from other point-to-point and local events in that they are completely decoupled from the simulation object on which they act. This is different from point-to-point or local events, since these events are defined and reside directly on the simulation object on which they act. Autonomous events also differ from point-to-point and local events in that they provide direct access to virtual methods in the event class `SpEvent`.

The most important feature available for use when using autonomous events, which is not available when using point-to-point and local events is the use of the SPEEDES lazy functionality (i.e. virtual method `lazy`). By using lazy, events will not be reexecuted if the reexecution of the event does not change the outcome (i.e. reexecution of event produces same result). For time consuming events that get rolled back, lazy can prevent the reexecution of these events, thus leading to better simulation performance. Also, if lazy passes, then anti-messages are not sent, preventing yet more simulation objects from rolling back.

The following sections explain each virtual method available for use by the user when when designing autonomous events.

## 15.1 Implementing Autonomous Events

All autonomous events are a child class of `SpEvent`. An example of such an event is shown in Example 15.1

```
#ifndef E_MyEvent_H
#define E_MyEvent_H

#include "SpGlobalFunctions.H"
#include "SpEvent.H"
#include "SpDefineEvent.H"
```

```
#include "SpMsg.H"

#include "MySimObj.H"

class E_MyEvent : public SpEvent {
  public:
    E_MyEvent()                {/* user code */}
    virtual ˜E_MyEvent()       {/* user code */}
    void MyMethod() {
      int temp = ((MySimObj *) SpGetSimObj())->GetValue() + 1;
      ((MySimObj *) SpGetSimObj())->SetValue(temp);
    }
    virtual void init(SpMsg *) {/* user code */}
    virtual int  lazy()        {/* user code */}
    virtual void exchange()    {/* user code */}
    virtual void commit()      {/* user code */}
    virtual void cleanup()     {/* user code */}
  private:
};
DEFINE_AUTONOMOUS_EVENT_0_ARG(MyEvent, E_MyEvent, MyMethod);
#endif
```

Example 15.1: Generic Autonomous Event

Events always act on a simulation object. For the above example, the simulation object that this event works on is `MySimObj`. In order to get this simulation object, global function `SpGetSimObj` is used with its return pointer casted to the `MySimObj`. When designing an autonomous event, any combination of the virtual methods shown in the above example may be implemented.

After implementing the needed virtual method in the event class, the user event method needs to be turned into an event. This is done by applying the `DEFINE_AUTONOMOUS_EVENT` to the class and method. The API for this macro is described in Section 6.5.

## 15.2 Event Processing Phases

Fundamental to understanding autonomous events is understanding the phases involved in the SPEEDES event processing algorithm. Table Table 15.1 shows the different phases of an event.

| Phase | Task | Event Method Executed |
|---|---|---|
| 1 | Event construction | Event constructor |
| 2 | Event initialization | virtual SpEvent::init(SpMsg *) |
| 3 | Normal or lazy processing | User event method or virtual SpEvent::lazy() |
| 4 | Rollback preparation | virtual SpEvent::exchange() |
| 5 | Rollback[1] | virtual SpEvent::exchange() |
| 6 | Permanent processing | virtual SpEvent::commit() |
| 7 | Event cleanup | virtual SpEvent::cleanup() |
| 8 | Event destruction | SpEvent::˜SpEvent() |
| [1]If rolled back, return to phase 3 | | |

Table 15.1: Event Processing Phases

Each method shown in Table 15.1 is a virtual method on `SpEvent`. Users can supply their own code for each of these methods in order to customize their event design. Each phase is discussed in more detail below:

Phase 1. During simulation initialization, SPEEDES constructs events as part of free lists. Thus, event constructors are called only once during an entire simulation, even though an event is normally reused many times.

Phase 2. When an event is scheduled, an event of the scheduled type is pulled off the free list and its `init(SpMsg *)` method is called. SPEEDES then places this event in the appropriate simulation object's event queue ordered by time stamp.

Phase 3. At the appropriate time, SPEEDES pulls the event off the queue and performs one of two actions:

(a) Normal event processing (i.e. user code is executed).

(b) Lazy processing. If lazy is enabled in `speedes.par` and lazy has been enabled for the current event (i.e. `SpEvent::set_lazy` has been called), then method `lazy` is called. If this method returns 0, then the event is reprocessed. Otherwise, the event is skipped (i.e rolled forward).

Phase 4. Immediately after undergoing normal or lazy processing, SPEEDES calls `exchange` to prepare for processing later potential rollbacks. The exchange technique is the fastest known method for processing rollbacks for individual state variables.

Phase 5. This phase represents what happens during a rollback, if it ever occurs. Otherwise, SPEEDES skips this phase. If the event is rolled back, SPEEDES calls `exchange` again to restore state data stored in the original `exchange` call. Of course, rollback variables have their states restored automatically using the incremental state saving technique (see Section 4.1 and Section 5.4). At this point, the event is reinserted into the event priority queue according to its time stamp ordering, and SPEEDES moves back to phase 3. Phases 3 through 5 may occur numerous times.

Phase 6. Once GVT is updated to be greater than or equal to this event's time stamp, rollbacks are impossible. This is because no events may be processed with a time stamp earlier than the current GVT (see Appendix A). Thus, at this point, phases 3 through 5 permanently stop cycling and SPEEDES calls `commit`, calling user code that cannot be rolled back.

Phase 7. Just before the event is inserted back into its free list, SPEEDES calls `cleanup`. This is where the event is restored to a clean state, such as deleting memory allocated during `init`.

Phase 8. Finally, at the end of the simulation, the event's destructor is called, cleaning up anything occurring during its constructor or any other final operations. Like the constructor, the destructor is only called once, even though the event may be reused (going through phases 2-7) multiple times throughout the simulation.

The following subsections discuss how each of these phases relate to the virtual method in `SpEvent`.

### 15.2.1 Lazy Re-evaluation

Lazy re-evaluation is a technique that allows SPEEDES simulations to process events out of order without having to reexecute them, according to criteria established by the user. By processing events in parallel, while sometimes allowing events to be processed out of order, SPEEDES has the potential to "beat the critical path", since this technique bypasses the general rule that a simulation's critical path must be processed sequentially.

Users specify the criteria for passing lazy re-evaluation using the virtual method `SpEvent::lazy`. Whenever an event is rolled back and the event is participating in lazy re-evaluation, then SPEEDES holds back anti-messages. When it is time to reprocess the event, SPEEDES first calls `lazy`. If the reexecution of the user event will not change the final outcome of the event (i.e. the same results are achieved after the event is reexecuted or perhaps just "close enough"), then method `lazy` should return a non-zero value. Since the user implements method lazy, the user can use whatever criteria they choose. If the reexecution of the event produces different results, then lazy should return zero. SPEEDES will then send out anti-messages associated with this event followed by the reprocessing of the event user code (i.e. normal event execution).

To implement this technique, users need to do the following:

1. Edit `speedes.par` and enable lazy in section `parameters`.
2. Each event type that is going to participate in lazy needs to enable lazy by calling method `set_lazy` (defined in `SpEvent` class). This can be done in the event's constructor.
3. Implement method `lazy`. This method should evaluate whether or not the event should be reexecuted. If it does, then it should return 0. Otherwise, it should return 1. A common method of doing this is to save the simulation object state variables when the event user code is executed. Method `lazy` can then compare the saved values against the simulation object state variables and, if there are no differences, then it should return 1.

Events that pass `lazy` will roll forward rather than being reprocessed. SPEEDES will not send any anti-messages, hence other events will not be rolled back. This can save considerable overhead, especially for events that schedule many other events or for events that depend on a low percentage of the simulation object's state, or both.

While the lazy re-evaluation technique can save considerable optimism overhead, it also requires extra memory and performance overhead by withholding anti-messages longer, saving lazy state variable values in the event object, and performing the `lazy` calls. Users should consider these tradeoffs when considering whether or not to implement `lazy`.

### 15.2.2 Fast Rollbacks

As an alternative to using rollbackable variables, users can use a slightly higher performing technique called "exchange" to make simulation object variables "rollback proof". However, this method is, in general, more tedious to implement. The exchange technique manually exchanges old with new, and new with old, simulation object state variables for rollback preparation, rollbacks, and rollforwards. SPEEDES calls the virtual method `SpEvent::exchange` immediately after calling the event method, and calls the same `exchange` method again before rolling back the event, thereby restoring the original value. This provides the same functionality as rollbackable variables, but spares the rollback variable overhead associated with creating, storing, and managing incremental state saving items.

To implement the `exchange` method, use normal variables (i.e. non-rollbackable) to represent each state variable in the simulation object. In the event method, use a separate version of that variable and store it in the `SpEvent` object. Then, `exchange`, simply swaps the event variable and simulation object variable. Users can swap multiple state values in `exchange`. An example of this is shown in Example 15.2.

The exchange technique is a good way to implement a rollbackable version of a variable if it does not exist and it is cost prohibitive to write a rollbackable version of the variable (see Section 5.4). In addition, using the exchange technique is slightly more efficient than using rollbackable variables.

However, using rollbackable variables eliminates the need for users to concern themselves with simulation object state variable restoration on event rollbacks, hence making rollbackable variables easier to use. Thus, for maintainability and ease of use, using rollbackable variables is probably the best and easiest method when designing and implementing simulation objects. To create custom rollbackable variables, see Section 5.4.

### 15.2.3 Committing Events

While `exchange` makes simulation object state data "rollback proof", `commit` makes any kind of operation "rollback proof." In fact, `commit` is so general that a user could eliminate using rollback variables and operations entirely by using `exchange` and `commit`, respectively. As the `exchange` technique replaces rollbackable variables, `commit` replaces rollbackable operations. `Commit` provides a slightly higher performing, but more tedious alternative to using rollbackable operations in the event method.

The idea behind `commit` is to perform operations only once an event can no longer be rolled back. SPEEDES accomplishes this by waiting until GVT passes an event's time stamp to call that event's `commit` virtual method. Since, by definition, no events may be scheduled with a time stamp earlier than the current GVT, any operations that occur during `commit` are final and are, therefore, guaranteed not to rollback.

Developers implement actions in `commit` that occur as a result of calling the event method. Because these actions cannot be rolled back, there is no rollback infrastructure needed to support these actions. Thus, using `commit` is slightly more efficient than using rollback operations, which do require the rollbackable infrastructure. Also, `commit` is useful for implementing rarely used, system specific external commands (such as writing to an external data base), since using `commit` relieves developers from having to develop custom rollbackable operations.

However, using `commit` has the same drawback as using `exchange`. That is, it exposes an additional coding layer to handle rollback issues that is transparent when using rollbackable operations. To create custom rollbackable operations, see Section 5.4.

## 15.3 Increasing Efficiency of Autonomous Events

The efficiency of autonomous events can be increased using `init`, `cleanup`, and the `SpEvent` constructor . There are only rare cases where these methods will be useful. Using the `SpEvent` constructor allows users to perform costly operations once per free list event. Since events are managed using free lists, each event in the free list is only constructed once during the entire simulation. Thus, data that rarely or never changes can be accessed from a database and stored in the event object (probably as a static variable) during construction. If exiting SPEEDES without memory leaks is important, any memory allocated should be deallocated in the `SpEvent` destructor.

The `init` virtual method is called exactly once per scheduled event. Operations that need to be performed each time the event is invoked, and that are not affected by rollbacks, can be placed in `init`. This generally applies to operations that do not access or change simulation object state. Moving event code into `init`, where possible, increases performance, since these operations will not be reprocessed each time the event is rolled back. Any memory allocated in `init` should be deallocated in `cleanup`. Also, `cleanup` is a good place to reinitialize event data, such as setting pointers to `NULL`.

## 15.4 Autonomous Event Example

Example 15.2 demonstrates each of the advanced techniques described in this chapter. It counts the number of rollbacks (i.e. when `lazy` does not pass) and the number of times `lazy` does pass for each event, as well as the total of each of these values for each free list event at the end of the simulation. For this example, assume that `S_MySimObj` has a non-rollbackable state integer, `val`.

```
#ifndef E_MyEvent_H
#define E_MyEvent_H

#include "SpGlobalFunctions.H"
#include "SpEvent.H"
#include "SpDefineEvent.H"
#include "SpMsg.H"

#include "MySimObj.H"

#define MY_SIMOBJ ((MySimObj *) SpGetSimObj())

class E_MyEvent : public SpEvent {
  public:
    E_MyEvent() {
      set_lazy();
      TotalRollbacks  = 0;
      TotalLazyPasses = 0;
    }
    virtual ~E_MyEvent() {
      cout << "Total rollbacks for this free list event = "
           << TotalRollbacks
           << ", total lazy passes for this free list event = "
           << CurrentLazyPasses << endl;
    }
    void MyMethod() {
      LazyVal     = 0;
      ExchangeVal = 0;
      Val         = 0;
      ExchangeVal = MY_SIMOBJ->GetValue() + 1;
    }
    virtual void init(SpMsg *) {
      CurrentRollbacks  = 0;
      CurrentLazyPasses = 0;
    }
    virtual int lazy() {
      if (MY_SIMOBJ->GetVal() == LazyVal) {
        CurrentLazyPasses++;
        return 1;
      }
      else {
        CurrentRollbacks++;
        return 0;
      }
    }
    virtual void exchange() {
      int temp;
      temp = MY_SIMOBJ->GetValue();
      MY_SIMOBJ->SetValue(ExchangeVal);
      ExchangeVal = temp;
    }
    virtual void commit() {
      cout << "Simulation object Value set to: " << ExchangeVal << endl;
    }
    virtual void cleanup() {
      TotalRollbacks  += CurrentRollbacks;
      TotalLazyPasses += CurrentLazyPasses;
      cout << "Rollbacks for this event = " << CurrentRollbacks
           << ", lazy passes for this event = " << CurrentLazyPasses;
    }

  private:
    int LazyVal;           // Original value of val at start of event
    int ExchangeVal;       // Value of val from SimObj at end of event
    int CurrentRollbacks;  // Number of rollbacks in this event
    int CurrentLazyPasses; // Number of times lazy passed in this event
    int TotalRollbacks;    // Total rollbacks for this event
    int TotalLazyPasses;   // Total times lazy passed for this event
};

DEFINE_AUTONOMOUS_EVENT_0_ARG(MyEvent, E_MyEvent, MyMethod);
#endif
```

Example 15.2: Autonomous Event Example

# Chapter 16

# Checkpoint/Restart: Using Persistence

Enabling a simulation to be checkpointed and restarted is an involved process which requires additional work from the user. Once enabled, a performance penalty ranging from 10% to 60% will be encountered, as well as short halts of the simulation while the checkpoint file is written to disk.

## 16.1 Persistence Memory Management Description

Consider a C++ class:

```
class foo {
  public:
    int       type;
    double    weight;
    double[3] position;
};
```

This class has a characteristic commonly called “flat.” This means that the class does not have any pointers or any other sort of dynamic memory contents. If this class were written out to disk in binary and then read back in again, it would be just as valid as it was before it was written out.

Now consider this C++ class:

```
class PointsToSelf{
  public:
    PointsToSelf :
      self(this) {};
  private:
    PointsToSelf* self;
};
```

In this class, the value `self` always points back to the class itself. This class is not flat and, if written to disk and then read back in, most likely would end up in a different memory location. This means that the value of `self` would most likely be incorrect and this pointer must be changed to the new address if the class is to be used correctly. This is the essence of persistence memory management.

## 16.2 Basic Changes to Enable Checkpoint/Restart

The SPEEDES framework provides functionality that allows users to save the simulation state to disk, as well as the ability to restore the simulation state for restart. There are a number of basic changes that must be made in order to support checkpoint and restart. The first change to be made is to modify `speedes.par` to support check points. Add the following section to `speedes.par`:

```
  Checkpoint {
    logical Enable           T             // Enable checkpoints
    float   WallTimeInterval -1.0          // Wall clock time between
                                           //  checkpoints
    float   SimTimeInterval  100.0         // Simulation time between
                                           //  checkpoints
    string  CheckpointPath   ./checkpoints // Path to checkpoint files
}
```

Checkpoints are enabled by setting the parameter `Enable` to `T` in section `Checkpoint`. Checkpoints occur at the rates specified by the parameters `WallTimeInterval` and `SimTimeInterval`. When a negative or zero time is specified, then this parameter will not be used when determining if it is time to perform a checkpoint. Check point files will then be written to files in the path specified by `CheckpointPath`.

The macro `PO_DEFINE_CLASS` is defined in `PO.H` and generates many short functions that are useful for persistence memory management. `PO_DEFINE_CLASS` should never be called by a user. Instead, users should use macro `RB_DEFINE_CLASS`, which internally calls `PO_DEFINE_CLASS`, to be used instead. Macro `RB_DEFINE_CLASS` is described in Section 5.1. Some of the more useful functions generated by `PO_DEFINE_CLASS(foo)` include:

```
  foo* PO_NEW_foo()              // Allocates a new foo and
                                 //  registers it with persistence
  foo* PO_NEW_ARRAY_foo(int n)   // Allocates an array of new foos and
                                 //  registers them with persistence
  foo* PO_DELETE_foo()           // Allocates a foo and unregisters
                                 //  it with persistence
  foo* PO_DELETE_ARRAY_foo(int n) // Deallocates an array of foos and
                                 //  unregisters them with persistence
  void PO_REGISTER_CLASS_foo()   // Registers foo with the database
                                 //  for later reconstruction.
```

These are similar to the `RB_DEFINE_CLASS` generated functions, except that these functions do not operate rollbackably. Therefore, it is recommended that these functions only be used when it is guaranteed that the code being processed will not be rolled back. Examples of this are the simulation object methods `Init`, `Terminate`, construction, destruction, or other times outside of normal event processing.

In order to checkpoint and restart a simulation, SPEEDES must be told which objects to store and restore, where the pointers are within those objects, and how to create those objects. These issues are addressed in the following sections.

Finally, there exists a global variable that contains the current on/off status of persistence:

```
  int PersistenceEnabled;
```

When `PersistenceEnabled` is set to `1`, then checkpoint/restart has been enabled in file `speedes.par` and checkpoints are being created.

### 16.2.1 Rollbackable Classes and Functions

All rollbackable classes and functions contained in the SPEEDES framework have been made checkpoint/restartable. This means no additional work must be performed to attach pointers for rollbackable container classes or other structures. One exception to this rule is the `RB_ostream` class. If a new `RB_ostream` is created from another stream, undetermined behavior will result with this `RB_ostream` when it is used or accessed after a restart.

For a user-defined class, `RB_NEW_class` and its variants from the `RB_DEFINE_CLASS` macro will automatically inform the database that class should be checkpointed. Similarly, `RB_DELETE_class` informs the database that piece of memory does not need to be stored.

### 16.2.2 Registering Classes to be Restored

Classes must be recreated upon a restart and, in order to know how to create them, they must be registered with the persistence database. SPEEDES automatically registers all the classes, before the call to `main` occurs, through a call to `*_REGISTER_CLASS_class` for each each `RB_DEFINE` class used.

Whenever `*_REGISTER_CLASS_class` is called, an instance of the class is created so that their persistence database can identify information about the class, such as the locations of virtual function table (vtable) (see Section 16.2.5). When this temporary version of the class is created, the following static data member is accessable:

```
SpPoDataBase::DoNotAllocateMemoryInConstructor
```

Its value is `1` during persistence initialization and `0` at other times.. The constructor of the class being registered should examine this value and, if it is set, avoid assuming anything about the state of the program upon its construction. This will help avoid unexpected behavior as well as memory leaks. In general, when `DoNotAllocateMemoryInConstructor` is set to `1`, users should initialize pointers to `NULL` and initialize primitive base types (integers, doubles, etc.) and nothing more.

### 16.2.3 Attaching Pointers

Identifying what pointers need to be stored and restored is done on a class instance-by-instance basis. `PO.H` defines a simple macro `PO_ATTACH_PTR(void*&)` that is called to indicate a specific pointer within an object instance is to be stored and restored.

Consider again this simple non-flat class:

```
class PointsToSelf {
  public:
    PointsToSelf :
      self(this) {}
  private:
    PointsToSelf* self;
};
```

This class can be modified to always attach the pointer to self by changing the constructor to read:

```
PointsToSelf::PointsToSelf :
  self(this) {
  PO_ATTACH_PTR(self);
}
```

This will cause the pointer self to be attached whenever this class is created (i.e. `RB_NEW` or `PO_NEW`). If it is desired that the pointer only be attached occasionally, then the call to `PO_ATTACH_PTR` can be made anywhere before a checkpoint is called. If the pointer is not attached before a checkpoint, the pointer will not be restored upon a restart. Any pointers that are attached in the constructor will only be automatically attached if the class is created using `RB_NEW` or `PO_NEW`.

### 16.2.4 Adding and Removing Memory From Persistence

The persistence database must be informed about which objects will be saved and restored. To reduce the amount of work a user must perform, all memory that is allocated using `RB_NEW` and deallocated with `RB_DELETE` is automatically added and removed from the persistence database. Other methods are available to add and remove memory, but these are the only recommended methods for adding or removing memory from persistence.

### 16.2.5 Classes with Virtual Functions

Modern compilers handle virtual methods by storing a pointer in the class to a table of virtual functions to be called (see Figure 16.1). This method is efficient but results in added complexity to persistence memory management.

Identification of the location of the vtable is done by inheriting from the class `SpPersistence-BaseClass` which can be found in the header file `SpPersistenceBaseClass.H`. All classes that contain a virtual function should directly inherit from `SpPersistenceBaseClass` (multiple inheritance is not sufficient).

```
class foo {
  public:
    virtual ~foo();
};


class bar :  public foo {
  public:
    ~bar();
    int x;
    foo containedFoo;
};
```

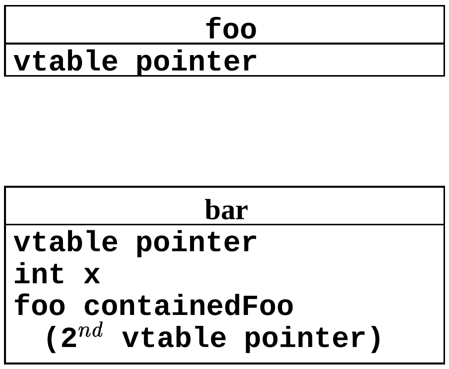


Figure 16.1: Placement of Virtual Function Table within Classes

### 16.2.6 Smart Pointers

The `RB_DEFINE_CLASS` macro also expands an additional class, which can be used to automatically attach pointers and also behave as a type-specific `RB_voidPtr`. This class looks like:

```
class RB_PTR_foo {
  public:
    RB_PTR_foo(){};
    RB_PTR_foo(foo* t);
    operator foo *();
    foo* operator =(foo* t);
    foo* operator -> ();
};
```

This class behaves in every way like a `foo*` and has the added advantage of being both a rollbackable and persistent pointer. It is automatically attached upon its first assignment and each assignment is undone upon a rollback.

## 16.3 Handling Events Which Pass Pointers

While defining events that take pointers is dangerous, there are still cases where a pointer needs to be passed in an event. Underneath the covers, SPEEDES uses messages to schedule events and the pointer in this message must be saved in order to be properly restored upon a restart.

Consider a case where an event had an interface defined as follows:

```
DEFINE_EVENT_INTERFACE_4_ARG(SpModifyFooAndBar,
                             int,
                             foo*,
                             double,
                             double,
                             bar*);
```

There would then be two pointers in this message that would need to be attached and subsequently restored upon a restart. The code necessary to schedule this event is shown in Example 16.1.

```
#include "PO.H"
RB_DEFINE_CLASS(PO_SPEEDES_MSG_TYPE);
...
  SpObjHandle objHandle;
  foo*        myFoo;
  bar*        myBar;
  int         count;
  double      velocity;
  double      altitude;
  ...
  M_ModifyFooAndBar_ARG* msg =
    (M_ModifyFooAndBar_ARG*)
    this->schedule(newTime,
                   ModifyFooAndBar_EVENT_ID,
                   objHandle->GetSimObjMgrId(),
                   objHandle->GetSimObjLocalId(),
                   objHandle->GetNodeId(),
                   "Additional data",
                   strlen("Additional data") + 1);
  msg->ExternalId = -1;
  msg->undirected = 0;
```

```
    msg->ProcReentryType = BEGIN_PROCESS;
    msg->EarliestStartTime = -1e20;
    msg->v1 = count;
    msg->v2 = myFoo;
    msg->v3 = velocity;
    msg->v4 = altitude;
    msg->v5 = myBar;
    if (PersistenceEnabled) {
      RB_PO_ADD((void*)msg,
                DefaultDataBase,
                PO_GET_PO_SPEEDES_MSG_TYPE_ID(),
                sizeof(M_ModifyFooAndBar_ARG) + msg->bytes);
      PO_ATTACH_PTR(msg->v2);
      PO_ATTACH_PTR(msg->v5);
      SpEvent::IgnoreDuplicatePO_Adds();
    }
```

Example 16.1: Persistence and Pointers as Arguments to Events

Here, the event is scheduled using the internal `SpEvent` method `schedule`. The message class `M_ModifyFooAndBar_ARG` is generated from the `DEFINE_EVENT_INTERFACE` macro. This message needs to be added to the database in a rollbackable fashion. The arguments to the `RB_PO_ADD` call are the data to be added, the database, the type of data, and the size of the data, respectively. The size of the message is the actual size of the class plus the size of any additional data added onto the end message.

The two pointers in the message `myFoo` and `myBar` are the second and fifth arguments respectively. The members of the message that correspond to these arguments are v2 and v5 respectively. Once the pointers to these arguments have been attached, the static method `SpEvent::IgnoreDuplicate-PO_Adds()` needs to be called to inform SPEEDES that the message has already been added to the database and to not try to add this message itself.

## 16.4 Printing the Database for Further Debugging

By executing SPEEDES with the argument “`-PrintDatabases type time`”, where `type` is either `SIM` or `WALL` and `time` is a time at which a checkpoint was created, a great deal of information is printed about the checkpoint file. Errors are printed out at the start and the first type of error encountered (if any) has the form:

```
  Tried to reconstruct class foo but this class
  has not yet been registered in this database.  This class
  will be restored as char buffers!!!!!!
```

The above message states that type `foo` was not registered before the call to `SpeedesExecute`. This should never happen and indicates either an internal error or a corrupted checkpoint file.

The next type of error appears as follows:

```
  !!!!!!!!!!!!!!!!!!!!!!!!Could not remap a pointer!!!!
  Pointer was inside an object of type bar
  with size 4400
  and had an offset of 3892
```

The exact piece of memory where the error occurred can be determined with the later output, but this indicates where within an object type the pointer error occurred. This generally means there was a `PO_ATTACH_PTR` called and the memory that was pointed to was not added to the database.

The actual database output then begins after the error messages. The format of this file is output and starts off with a brief description of the "shape" of a class.

```
Class name: foo   282
Class size: 380
Number virtual function tables: 3
     vtable at offset 0
     vtable at offset 284
     vtable at offset 336
```

This indicates the name and id of a class along with its size. Then follows information about the number of vtables within the class along with their offsets within the class. This information can be used to help identify classes that do not have all of their vtable pointers registered through inheritance, as described in Section 16.2.5.

Next comes output describing each block of memory that was added to persistence. A block of memory is allocated with `RB_NEW` or `PO_ADD`.

```
Class type:    foo  Dangling memory: Yes
Single item
Base address: 135789336
Memory block size: 380
Unnamed
Memory block has 18 pointers
     /---------------------------------------------------------\
    /------------------------Pointers------------------------\
   /-------------------------------------------------------------\
  |  Array    |  offset of pointer |  destination |  destination |
  | element   |   within element   |  of pointer  | in database? |
  |     0     |         32         |   135804816  |     YES      |
  |     0     |        152         |   135787200  |     YES      |
  |     0     |        164         |           0  |      -       |
  |     0     |        172         |           0  |      -       |
  |     0     |        184         |           0  |      -       |
  |     0     |        188         |           0  |      -       |
  |     0     |        192         |           0  |      -       |
  |     0     |        240         |   135803844  |     YES      |
  |     0     |        264         |           0  |      -       |
  |     0     |        304         |           0  |      -       |
  |     0     |        312         |           0  |      -       |
  |     0     |        324         |   135805056  |     YES      |
  |     0     |        328         |   135807728  |     YES      |
  |     0     |        332         |   143305872  |     YES      |
  |     0     |        348         |   141882400  |     YES      |
  |     0     |        352         |           0  |      -       |
  |     0     |        356         |   141986096  |     YES      |
  |     0     |        360         |   141986096  |     YES      |
  ---------------------------------------------------------------
```

Figure 16.2: Persistence Block Example

This output also starts with class type and an indication as to whether the memory is dangling (nothing else pointing to it) or not. The next line indicates if its a single item or an array of items and that is followed by the base address of the block when it was saved. Next follows the size, the name (if named) and the number and list of pointers. The pointers appear in a block format with one pointer per line. The first column is the array element within the block that the pointer appears. The second column is the offset of the pointer within that specific array element. The third column is the destination of the pointer and the fourth column states whether the destination is in the database or not. If the fourth column is `NO`, this indicates that persistence pointer (`PO_ATTACH`) pointer was not added (`PO_ADD`), resulting in one of the errors at the start of the output.

## 16.5 Tips, Tricks, and Potholes

1. Always use `RB_NEW` and `RB_DELETE` to allocate memory, even in a simulation object constructor. The overhead is minimal and this will ensure that memory is always added to the database.

2. Use of smart pointers can reduce the number of errors of not attaching pointers. When these cannot be used, attach the pointers in the constructors and always use `RB_NEW` and `PO_NEW` to ensure they are attached.

# Chapter 17

# Diagnostic Tools

Determining the correct operation of a discrete-event simulation can be a daunting task. In an optimistic processing simulation framework, such as SPEEDES, this task becomes nearly impossible without adequate tools to assist the user in diagnosing run-time errors and analyzing run-time performance.

## 17.1 Global Virtual Time (GVT) Statistics

GVT is calculated at fairly regular intervals and, upon every calculation, statistics can be printed updating the progress of the framework. This is enabled by setting parameter `statistics` in the `parameters` section of `speedes.par` to “`T`”. Summary lines are printed to the screen. The default GVT output line will be similar to that shown in Figure 17.1.

```
  3) GVT=415  cpu=1.1 wall=1.1 STAR=0     Tproc=0.15 Tcmt=0 Eff=0
     Nproc=1268 Ncmt=46  e=126 e/c=21 eg=1611 r=3382 m=223 a=188 c=4
  6) GVT=920  cpu=2.5 wall=2.5 STAR=360.2 Tproc=0.23 Tcmt=0 Eff=0
     Nproc=354  Ncmt=63  e=393 e/c=33 eg=2339 r=4792 m=268 a=192 c=12
  9) GVT=1615 cpu=3.8 wall=3.9 STAR=488.8 Tproc=0.32 Tcmt=0 Eff=0
     Nproc=867  Ncmt=132 e=714 e/c=40 eg=3436 r=7025 m=339 a=206 c=20
```

Figure 17.1: GVT Statistics Output Line

The content of the GVT output line is controlled through different parameters in the `statistics` section in `speedes.par` (see Section C.4 for additional information). A description of the `statistics` parameters, GVT output line symbol, and a parameter description is given in Table 17.1.

| Statistics Parameter | Output | Description |
|---|---|---|
| CYCLE | value) | Prints the GVT cycle number at the start of the statistics line. This is the number of times GVT has been calculated. |
| GVT | GVT= | The current GVT which is the latest simulation time that has been committed (cannot be rolled back). |
| CPU | cpu= | The total processing time spent thus far. |
| WALL | wall= | The total wall time spent since the start of the simulation. |
| STAR | STAR= | Simulation Time Advancement Rate for the previous cycle. This is the ratio of change in GVT to the change in wall clock. |

| **Statistics Parameter** | **Output** | **Description** |
|---|---|---|
| PROC | Tproc= | The total time spent processing events events since the start of the simulation. |
| COMMIT | Tcmt= | The total amount of committed (non-rolled back) event processing since the start of the simulation. |
| PROCEFF | Eff= | The ratio of committed processing time to total processing time since the start of the simulation. |
| PHASE1 | Nproc= | The number of events processed in the prior cycle. |
| PHASE2 | Ncmt= | The number of events committed in the prior phase. Note that the number committed can be greater than the number processed. |
| EVENTS | e= | Total number of events processed since the start of the simulation. |
| EVENTSCYCLE | e/c= | Total number of events processed during the last cycle. |
| EVTGVT | eg= | Total number of events committed since the start of the most recent GVT cycle |
| ROLLBACKS | r= | Number of rollbacks since the start of the simulation. |
| MESSAGES | m= | Total number of event messages sent since the start of the simulation. |
| ANTIMESSAGES | a= | Total number of anti-messages sent since the start of the simulation. An anti-message is a message canceling a scheduled event due to a rollback. |
| CANCELS | c= | Total number of events canceled since the start of the simulation. |

Table 17.1: GVT Output Line Description

The GVT statistic can be printed on a node-by-node basis by setting the additional parameters in the `statistics` section of the `speedes.par` file.

```
logical WriteGvtStatistics T
string  FileName           GvtStatistics
```

The first parameter enables and disables the printing of the statistics and the second sets the name of the output files. The file name specified is appended with ".#" where # is the node number for each output file. These output files will have the format as shown in Figure 17.2.

```
Cycle=396
GVT=95.0216
LVT=95.2964
Risk=0.0214499
EnterFB=0.00358595
ExitFB=0.0214499
UpdateGVT=0.0216699
EnterSpin=-1e+20
Commit=0.0236249
CPU=9.53
Wall=10.1205
STAR=10.3702
Tproc=2.83806
Tcmt=2.82824
Eff=0.996539
Nproc=116
Ncmt=89
Evts=41545
E/Cyc=105
E/gvt=1442
```

```
RBs=51
Msgs=20720
Anti=17
Cancel=0
Xmsgs=0
# events in queue=200
```

Figure 17.2: GVT Statistics File Output

Much of this information is very similar to that provided in the GVT output line but there are a few new items and much of the data is presented on a node-by-node basis. The particular lines and their descriptions are given in table 17.2. Those options that are only available in the output files are presented in bold face font.

| **Statistics Parameter** | **Output** | **Description** |
|---|---|---|
| CYCLE | Cycle= | Prints the GVT cycle number at the start of the statistics line. This is the number of times GVT has been calculated. |
| GVT | GVT= | The current GVT which is the latest simulation time that has been committed (cannot be rolled back). |
| **LVT** | LVT= | The Local Virtual Time (LVT). This is the time tag of the next event to be processed in the event queue. |
| **BTW** | Risk= | Number of seconds into the last cycle at which the simulation stopped processing events and sending messages. |
| | EnterFB= | Number of seconds into the last cycle at which this node of the framework requested an update of GVT. |
| | ExitFB= | Number of seconds into the last cycle at which all the nodes agreed to update GVT. |
| | UpdateGVT= | Number of seconds into the last cycle at which the framework finished updating GVT. |
| | EnterSpin= | Number of seconds into the last cycle at which the current node reached a point at which it would no longer process events. |
| | Commit= | Number of seconds into the last cycle at which the framework finished committing events. |
| CPU | CPU= | The total processing time spent thus far. |
| WALL | Wall= | The total wall time spent since the start of the simulation. |
| STAR | STAR= | Simulation Time Advancement Rate for the previous cycle. This is the ratio of change in GVT to the change in wall clock. |
| PROC | Tproc= | The total time spent processing events events since the start of the simulation. |
| COMMIT | Tcmt= | The total amount of committed (non-rolled back) event processing since the start of the simulation. |
| PROCEFF | Eff= | The ratio of committed processing time to total processing time since the start of the simulation. |
| PHASE1 | Nproc= | The number of events processed in the prior cycle. |
| PHASE2 | Ncmt= | The number of events committed in the prior phase. Note that the number committed can be greater than the number processed. |

| Statistics Parameter | Output | Description |
|---|---|---|
| EVENTS | Evts= | Total number of events processed since the start of the simulation. |
| EVENTSCYCLE | E/Cyc= | Total number of events processed during the last cycle. |
| EVTGVT | E/gvt= | Total number of events committed since the start of the simulation. |
| ROLLBACKS | RBs= | Number of rollbacks since the start of the simulation. |
| MESSAGES | Msgs= | Total number of event messages sent since the start of the simulation. |
| ANTIMESSAGES | Anti= | Total number of anti-messages sent since the start of the simulation. An anti-message is a message canceling a scheduled event due to a rollback. |
| CANCELS | Cancel= | Total number of events canceled since the start of the simulation. |
| **NUM_EVENTS_IN_QUEUE** | # events in queue= | The number of events in the event queue at the time of the GVT update. |

Table 17.2: GVT Output File Description

## 17.2 Trace Files

Trace files provide a mechanism for allowing event execution flow to be analyzed. For example, if event A schedules event B, which schedules event C, then this sequence of events is captured in a text file. Basically, a trace file contains the time and name for each committed event and the names of all events that it scheduled. By looking at these files, a user can trace the event sequence for a given simulation execution. To enable trace files, add the following sections to the `speedes.par` file:

```
trace {
  logical trace     T     // T = on, F = off
  string  tracefile Trace // File name appended by node number
}
```

Figure 17.3 shows the trace file from the car and stop light example described in Section 6.5.

```
Time = {30,0,0,5,0}
EvtName: StopLight_TurnsRed
TriggerName:
ObjName: S_StopLight_MGR 0
SimObjMgrId: S_StopLight_MGR
Handle:  (0,1,0)
GlobalId:  4
RBitems: 3
Cancel id: 8
EvtCPU:  0.000177992
TotCmtd: 0.000177992
TotTime: 0.000177992
CritPth: 0.000177992
ProcessMode: Timewarp
NumEventsBeyondGVT: 8
TimeBeyondGVT: 1e+20
NoTag:
  LocSched: Time = {30,0,0,6,4}
    EvtName:  Car_Stop
    SimObjMgrId:  0
    Handle:  (0,0,0)
    bytes:    101
  LocSched: Time = {30,0,0,8,4}
    EvtName:  Car_Stop
    SimObjMgrId:  0
    Handle:  (0,0,1)
    bytes:    101
  LocSched: Time = {60,0,0,10,4}
    EvtName:  StopLight_TurnsGreen
    SimObjMgrId:  1
    Handle:  (0,1,0)
    bytes:    80
  RskSched: Time = {30,0,0,7,4}
    EvtName:  Car_Stop
    Handle:  (1,0,0)
    bytes:    101
  RskSched: Time = {30,0,0,9,4}
    EvtName:  Car_Stop
    Handle:  (1,0,1)
    bytes:    101
```

Figure 17.3: Trace File Example Output

The trace file shows event `StopLight_TurnsRed` going off at time 30.0 seconds (stop light turned red). It scheduled event `Car_Stop` for each car object telling the car that the light was red. It then scheduled event `StopLight_TurnsGreen` at time 60.0 for itself so that the stop light can turn from red to green at the appropriate time. Additional information on each individual line contained in this file is shown in Table 17.3.

| **Line Number(s)** | **Description** |
|---|---|
| 1 - 7 | Basic information on current event being processed. This information is the time, event name, trigger name (for handlers), object name, type, object handle, and global id of the event being processed. |
| 8 | Specifies the number of rollback items created for this specific event. The number of rollback items is usually equal to the number of rollbackable operations performed in this event. |
| 9 | Specifies the cancel id for this event. |
| 10 | `EvtCPU` is the amount of time used by this specific event. Although the name includes the phrase “CPU”, the number will only sometimes represent CPU time. Other times, it may represent wall time or counts, depending on what `Timer`’s value is in the `statistics` section of `speedes.par`. |
| 11 | `TotCmtd` is the total committed amount of time used by this simulation object up to this point. |
| 12 | `TotTime` is the total amount of time used by the simulation object up to this point (this includes time that was “wasted” by a rollback). |
| 13 | `CritPth` is total length of critical path leading to this event. |
| 14 | `ProcessMode` will be one of the following:<br>• `Breathing Time Buckets`:<br>Indicates that this event was processed, but any events it scheduled are held back until the rest of the simulation catches up with this node. |

| Line Number(s) | Description |
|---|---|
| 14 (cont.) | • `Timewarp`: Indicates that this event was processed and the events it scheduled were released to other nodes.<br>• `Risk Free`: Indicates that this event was processed without risk (i.e. events were scheduled without fear of anti-messages calling them back).<br>• `GVT Update`: Indicates that this event was processed while updating GVT, and its events were held until the end of the GVT update.<br>• `Not running BTW`: Indicates that Breathing Time Warp (BTW) algorithm was not in operation when this event was processed. This case occurs when using other algorithms (such as sequential) or in unique situations, such as event `InitialEvent`. |
| 15 - 16 | `NumEventsBeyondGVT` and `TimeBeyondGVT` are the number of events beyond GVT that the current event has processed and the number of simulation seconds beyond GVT that the event has processed. Large values for these two parameters indicate a high degree of optimism in the processing of the current event. |
| 17 | Line 17 can have a value of `NoTag` or `BeginTag`. In the case of this entry being `BeginTag`, then additional user-definable data will be output followed by a marker called `EndTag`. Users can set this data in an event by filling out the data using class `SpTag`, which behaves similarly to `ostream`. For example, if the following code is added to event `StopLight_TurnsRed`:<br>`...`<br>`SpTag* tag = SpGetTag();`<br>`(*tag) << "Traffic light turned red at "`<br>`       << (double) SpGetTime() << " seconds." << endl`<br>`       << "Message sent to each car." << endl;`<br>`...`<br>then the trace file would contain the following:<br>`BeginTag:`<br>`Traffic light turned red at $30.0$ seconds.`<br>`Message sent to each car.`<br>`EndTag:` |
| 18 - 40 | These lines indicate all of the events that were scheduled by this event (i.e. `StopLight_TurnsRed`). Each event scheduled will be composed of three or four parts. The first part specifies one of the following:<br>• `LocSched`: Indicates that this event was locally scheduled (scheduled on the same node).<br>• `RskSched`: Indicates that the event was scheduled at risk of being rolled back due to an anti-message.<br>• `SafeSched`: Indicates that the event was scheduled on another processor without fear of anti-messages.<br>Each of these lines is followed by the time at which this event was scheduled. `EvtName` specifies the name of the event being scheduled. `SimObjMgrId` specifies the simulation object type on which this event is being scheduled. `Handle` specifies the object handle for the object for which this event was scheduled. `bytes` specifies the size of the scheduled event message in bytes. |

| Line Number(s) | Description |
|---|---|
| 18 - 40 (cont.) | Therefore, event StopLight_TurnsRed is scheduling five events. Event Car_Stop is scheduled on each car. The car simulation objects are distributed across two nodes. Therefore, two events are scheduled locally (i.e. LocSched) and two events are scheduled for the cars on the other node (i.e. RskSched). These later two events can cause the events on the car simulation objects to be rolled back, since these are events being scheduled for an object on a different node, especially since they are being scheduled for now. Event StopLight_TurnsGreen is scheduled on itself at 30.0 seconds in the future, so that it can turn the traffic light from red to green. |

Table 17.3: Trace File Definitions

SPEEDES provides a utility that will filter blocks of event data for post-processing analysis. The utility SpFilterTrace is detailed in Section 13.10 and is useful for following the behavior of a single object or a type of event. It will filter out events of a specific type on objects of specific types, names, or other searchable fields.

Here are some examples of commands that could be used to filter a trace file:

- Filter all events that were processed by stop lights:

  ```
  SpFilterTrace Trace_1 "S_StopLight_MGR"
  ```

- Filter all Car_Stop events on node 0:

  ```
  SpFilterTrace Trace_1 "EvtName: Car_Stop"
  ```

- Filter all Car_Stop events that have the data "Potential Bug" in the tag field.

  ```
  SpFilterTrace Trace_1 "EvtName: Car_Stop" "Potential Bug"
  ```

## 17.3 Flying Trace

The trace file provides an exact record of all the events that were processed and committed in a simulation. Given that measures were taken to ensure repeatability and that no external interactions were used, a trace file will be identical from run to run although different CPU times may occur due to computer clock differences.

Due to the optimistic nature of BTW, events may be processed, rolled back and reprocessed, or never processed again. The need to see what was really going on leads to the desire for a different type of trace known as "flying trace".

Once the application has been recompiled and linked, the flying trace capability needs to be enabled in speedes.par, as shown below:

```
FlyingTraceOutput {
  string TraceFileName flyingTraceFileName
}
```

This feature only works when the simulation runs on two or more nodes or when optimize_sequential is set to F.

A simulation run with the above modification will result in one file per node called `flyingTrace-FileName_#`, where # specifies the node number for which the data was collected. Figure 17.4 shows a portion of the flying trace file for the car and stop light simulation. This example shows the flying trace at $t = 0.0$. Recall that at $t = 30.0$ the traffic light turned red. The simulation was modified slightly in such a way as to cause a rollback on the stop light simulation object (event `StopLight_TurnsRed` gets rolled back), and therefore, the event executed at $t = 30.0$ is rolled back.

```
-------------------End non local messages------------
process: evtype:  1     objType:  0     localId:  0     time:0
process: evtype:  3     objType:  0     localId:  0     time:0
process: evtype:  1     objType:  0     localId:  1     time:0
process: evtype:  8     objType:  0     localId:  0     time:15
process: evtype: 10     objType:  0     localId:  0     time:15
process: evtype:  4     objType:  0     localId:  0     time:15
process: evtype:  3     objType:  0     localId:  1     time:20
process: evtype:  5     objType:  1     localId:  0     time:30
-------------------Begin non local messages------------
Schedule: evtype:  2 Node:  1 objType:  0 localId:  0 time:Time = {30}
Schedule: evtype:  2 Node:  1 objType:  0 localId:  1 time:Time = {30}
-------------------End non local messages------------
process: evtype:  2     objType:  0     localId:  0     time:30
process: evtype:  2     objType:  0     localId:  1     time:30
process: evtype:  8     objType:  0     localId:  1     time:35
process: evtype: 10     objType:  0     localId:  1     time:35

Got a rollback, evtype: 20     objType:  1     localId:  0     time:15
RollBack: evtype:  5     objType:  1     localId:  0     time:30
RollBack: evtype:  2     objType:  0     localId:  0     time:30
RollBack: evtype: 10     objType:  0     localId:  1     time:35
RollBack: evtype:  8     objType:  0     localId:  1     time:35
RollBack: evtype:  2     objType:  0     localId:  1     time:30
-----end of rollback------

process: evtype: 20     objType:  1     localId:  0     time:15
process: evtype:  5     objType:  1     localId:  0     time:30
-------------------Begin non local messages------------
Schedule: evtype:  2 Node:  1 objType:  0 localId:  0 time:Time = {30}
Schedule: evtype:  2 Node:  1 objType:  0 localId:  1 time:Time = {30}
-------------------End non local messages------------
process: evtype:  2     objType:  0     localId:  0     time:30
process: evtype:  2     objType:  0     localId:  1     time:30
process: evtype:  8     objType:  0     localId:  1     time:35
process: evtype: 10     objType:  0     localId:  1     time:35
```

Figure 17.4: Flying Trace File Example Output

Each event in a simulation is assigned a unique integer value during initialization. A table of these integers cross-referenced to event names is printed at the start of the simulation. By looking at this event name cross-referenced matrix, users can identify which event is processed in the above flying trace example. Table 17.4 explains the flying trace output file.

| Line Number(s) | Description |
|---|---|
| 1 - 9 | This shows what events were execuated in what order. At $t = 30.0$ the Stop Light object schedules an event on each car notifing the Car objects that the light is red. |
| 10 - 12 | This shows the events scheduled on objects not on this node. Since these are for cars, this shows that an event was scheduled for the cars on the other node (i.e. node 1). |
| 13 - 17 | Execution on this node continues. The first events executed are the `Car_Stop` events, which were scheduled by the event on line 9. The next events are the local Radio events. |
| 18 - 24 | The other node scheduled an event of type 20 on the Stop Light object on this node. This has caused the event executed on line 9 and all events it scheduled to be rolled back. Events of type 2, 8, and 10 executed on lines 14 - 17, are rolled back. Not shown in this print out are the cancel events for the two `Car_Stop` events scheduled on node 1. However, the other node's flying trace output file would show these events getting rolled back. |
| 25 - 34 | These lines once again show the normal timeline for the events. The traffic light once again turns red at $t = 30.0$, followed by the events that it scheduled. |

Table 17.4: Flying Trace File Definitions

## 17.4 Event Usage Statistics

Enabling trace and flying trace can result in significant performance degradation due to the increase in disk accesses (i.e. data writes to disk). A summation of some of these statistics can be obtained at the end of the simulation, as well as at fixed GVT time intervals. These statistics can be enabled for output by adding the correct parameter(s) to section `statistics` found in `speedes.par`. The following subsections describe each of the event statistics available to the user. The data shown in the tables below are based on the car stop light simulation example first described in Section 6.4. This example was sightly modified in that the traffic light now rolls back every 100 seconds.

### 17.4.1 Event Memory Usage

The amount of memory used by the event free list mechanism will be displayed by adding the following entry in the `speedes.par`:

```
statistics {
  logical MemoryUsage      T  // Memory statistics
}
```

The result of this will produce the data shown in Table 17.5.

| Event name | Id | Nevents | Nfree | Nmessages | Nfree | Bytes |
|---|---|---|---|---|---|---|
| Car_StopCar | 0 | 160 | 160 | 160 | 160 | 106240 |
| Car_Go | 1 | 540 | 540 | 30 | 30 | 317760 |
| Car_Stop | 2 | 530 | 530 | 0 | 0 | 309520 |
| Car_RadioController | 3 | 80 | 37 | 80 | 37 | 53120 |
| Car_RadioStation | 4 | 240 | 240 | 240 | 240 | 161280 |
| StopLight_TurnsRed | 5 | 60 | 47 | 60 | 47 | 39840 |
| StopLight_TurnsGreen | 6 | 60 | 60 | 60 | 60 | 39840 |
| Radio_Off | 7 | 30 | 30 | 0 | 37 | 17520 |
| Radio_On | 8 | 40 | 40 | 0 | 244 | 23360 |
| Radio_Scan | 9 | 240 | 175 | 0 | 2503 | 140160 |
| Radio_SendFrequency | 10 | 240 | 240 | 0 | 2633 | 140160 |
| RollBack | 11 | 30 | 29 | 30 | 30 | 19920 |
| SpInitial | 12 | 3 | 3 | 0 | 0 | 1788 |
| SpAddSimObjInteraction | 13 | 4 | 4 | 4 | 4 | 3040 |
| SpAddNodeInteraction | 14 | 8 | 8 | 4 | 4 | 5328 |
| SpHandler | 20 | 10 | 10 | 30 | 30 | 8600 |
| SpCancel | 21 | 40 | 40 | 40 | 40 | 26720 |
| Nholders = 8000, Nfree = 1901, Bytes = 480000 | | | | | | |
| Total bytes = 1894196 (1.8942 Meg) | | | | | | |

Table 17.5: Event Memory Usage Diagnostic Output Data

The data shown in this table is tabulated on an event type by event type basis. The columns shown are event name (`Event name`), event type id (`Id`), total number of events (`Nevents`) of that type in the free list, number of available events in the free list (`Nfree`), number of messages managed by the free list (`Nmessages`), number of available messages in the free list (`Nfree`), and the total bytes required to manage the events and messages (`Bytes`). Note that the number of events managed is not necessarily equal to the number of messages managed. Holders are used to contain the events and messages within the free list and their total number is presented here. Similarly, the total bytes for all messages, events and holders are presented.

### 17.4.2 Event Message Sending

Events on a simulation object can be scheduled on any node at anytime in the future. How do these events make their way to the appropriate simulation object on the correct node? The object handle correctly identifies the object instance and location. The act of "scheduling" an event causes SPEEDES to package up the event information and send it to the correct node so that this node can schedule the appropriate event on the specified object instance. This information is called a "message".

There are three types of messages in SPEEDES. The most common are asynchronous messages and anti-messages. Asynchronous messages are the messages that are normally sent in scheduling events. Most of the time, these messages are sent in an asynchronous fashion without waiting for the recipient of the message to respond. Anti-messages are messages sent to undo an event (i.e. rollback). Finally, coordinated messages are the standard event scheduling messages, but these are sent during a special phase of BTW (see Appendix A for more information on BTW) when the nodes are synchronized.

These are events that can be scheduled without fear of being rolled back (known as risk-free). The number of coordinated messages is, in general, small. If the number becomes large, this generally indicates that the value of `Nopt` in `speedes.par` is set too small.

To enable message sending statistics, add the parameter `MessageSending` to the `statistics` section in `speedes.par`:

```
statistics {
  logical MessageSending   T  // Message statistics.  Only valid
                              //  when simulation is run on
                              //  2 or more nodes.
}
```

The result of this will produce data shown in Table 17.6.

| **Event name** | **Nmess** | **Ncoord** | **Nanti** |
|---|---|---|---|
| Car_Go | 1380 | 15 | 1215 |
| Car_Stop | 1338 | 15 | 1173 |
| RollBack | 32 | 4 | 0 |
| SpAddNodeInteraction | 2 | 0 | 0 |
| SpHandler | 21 | 3 | 0 |

Number of asynchronous messages = 2773
Number of coordinated messages = 37
Number of antimessages = 2388
Total number of messages = 5198

Table 17.6: Event Message Sending Diagnostic Output Data

Columns 2, 3, and 4 show the number of asynchronous messages (`Nmess`), coordinated messages (`Ncoord`), and anti-messages (`Nanti`), respectively. This feature is unavailable when used only on one node.

### 17.4.3 Event Data Summary

Event processing statistics are available when parameter `EventProcessing` in section `statistics` is set in `speedes.par`. The format of this flag is shown below:

```
statistics {
  string  Timer            WallClock // Use CPU, WallClock, or Counter
                                     //  for event timings
  logical EventProcessing  T         // Event processing statistics
  logical CriticalPath     T         // Critical path statistics
}
```

The `Timer` setting in `speedes.par` affects how the data is measured for this feature. See The value of `Timer` can be one of `CPU`, `WallClock`, or `Counter`. When event summary data is being collected, the time sensitive data elements are collected using one of the following data collection methods.

- `CPU`:
  All time measurements are specified using CPU time.
- `WallClock`:
  All time measurements are specified using wall clock time (i.e. time passage on a watch or clock).
- `Counter`:
  Timer simply returns 1, regardless of the time elasped. Therefore, time represents the number of events.

If no timing data is desired, then the use of `Counter` will minimize the CPU overhead associated with the statistics collection. If parameter `CriticalPath` is set to true, then the event critical path is calculated and displayed as part of the event data summary. The critical path of a simulation is defined to be the longest sequence of events necessary to complete the simulation.

Table 17.7 shows the data collected from running the car and stop light simulation.

| **Event name** | **Nproc** | **Ncmtd** | **Tcpu** | **Tcmtd** | **<Tcmtd>** | **Eff** | **MaxProc** |
|---|---|---|---|---|---|---|---|
| Car_StopCar | 1000 | 200 | 1.006 | 0.261 | 0.001 | 0.260 | 0.2228 |
| Car_Go | 1207 | 244 | 0.009 | 0.002 | 1.1e-05 | 0.282 | 0.0001 |
| Car_Stop | 1168 | 240 | 0.040 | 0.009 | 4.1e-05 | 0.242 | 0.0002 |
| Car_RadioController | 375 | 37 | 0.018 | 0.002 | 5.7e-05 | 0.115 | 0.0002 |
| Car_RadioStation | 2561 | 228 | 0.023 | 0.002 | 9.6e-06 | 0.094 | 0.0001 |
| StopLight_TurnsRed | 478 | 60 | 0.036 | 0.004 | 8.2e-05 | 0.136 | 0.0007 |
| StopLight_TurnsGreen | 493 | 60 | 0.037 | 0.005 | 9.8e-05 | 0.157 | 0.0003 |
| Radio_Off | 38 | 3 | 0.000 | 3.8e-05 | 1.2e-05 | 0.079 | 1.5e-05 |
| Radio_On | 244 | 25 | 0.006 | 0.001 | 4.8e-05 | 0.194 | 0.0002 |
| Radio_Scan | 2474 | 211 | 0.087 | 0.008 | 4.2e-05 | 0.101 | 0.0004 |
| Radio_SendFrequency | 2572 | 228 | 0.036 | 0.003 | 1.6e-05 | 0.101 | 0.0001 |
| RollBack | 36 | 36 | 2.052 | 2.052 | 0.057 | 1 | 0.2180 |
| SpHandler | 36 | 36 | 0.176 | 0.176 | 0.004 | 1 | 0.1690 |
| SpCancel | 219 | 45 | 0.005 | 0.001 | 2.8e-05 | 0.226 | 0.0002 |
| Totals | 12901 | 1653 | 3.537 | 2.532 | 0.001 | 0.716 | 0 |

Critical path processing time = 1.504
Maximum possible speedup = 1.492

Table 17.7: Event Summary Diagnostic Output Data

Column 1. `Event name`:
The name of the event.

Column 2. `Nproc`:
The number of times an event of this type was processed.

Column 3. `Ncmtd`:
The number of times an event of this type was committed. The difference between this number and `Nproc` is the number of times this type of event was rolled back.

Column 4. `Tcpu`:
Total number of seconds spent processing events of this type including rollbacks. Although the name includes the phrase "CPU", the number will only sometimes represent CPU time. Other times, it may represent wall time or counts, depending on what the value of parameter `Timer` is in the `statistics` section of `speedes.par`.

Column 5. `Tcmtd`:
The same as `Tcpu`, except that `Tcmtd` does not include time spent on processing that was later rolled back.

Column 6. <`Tcmtd`>:
The average time spent committing an event of this type, or, $< \texttt{Tcmtd} >= \texttt{Tcmtd}/\texttt{Ncmtd}$. For example, SPEEDES spent a total of $3.8e-05$ seconds to commit all of the `Radio_Off` events. There were three `Radio_Off` events which means $< \texttt{Tcmtd} >$ was one-third of that figure: $1.2e-05$ seconds.

Column 7. `Eff`:

The efficiency of this event. The efficiency is defined by the quantity `Tcmtd`/`Tcpu`. This value gives a rough estimate of the proportion of useful CPU time spent on this event.

Column 8. `MaxProc`:

SPEEDES determines which event of this type took the longest to process, regardless of whether the event was rolled back. This parameter shows the time required to process that event.

This set of data often returns some of the most valuable information for simulation analysis, while also being the most difficult information to analyze. Columns `Tcpu` and `Tcmtd` give the total time spent processing this event and the total time spent committing the event, respectively. These may differ if the simulation is running on multiple nodes or if it is running with external interactions. For events which always take the same amount of time, `Nproc`/`Ncmtd` will be equal to `Tcpu`/`Tcmtd`. However, many, if not most events, will contain different code paths, hence different execution times. Therefore, in general, `Nproc`/`Ncmtd` may not be equal to `Tcpu`/`Tcmtd`, since event reexecution after a rollback can take a different path through the code.

### 17.4.4 Event Usage Statistics by Simulation Object

As a further refinement, the event data can be saved on a simulation object-by-event basis. To enable this feature, add section `StatsOnSimobjByEventBasis` to section `statistics` in `speedes.par`.

```
StatsOnSimobjByEventBasis {
  string  MyStatData        /tmp/output  // Directory name for the
                                         //  output data.
  logical PrintIntervalData T            // Interval data enable.
}
```

Parameter `MyStatData` specifies the directory name where all of the data files are to be saved. A file for each simulation object instance will be created, and all of its statistics will be written to these files. These file names are determined by the simulation object name. If simulation objects have the same names, then the output results will be concatenated into the same file. Subparameter `PrintInter-valData` specifies whether or not interval data is to be output (see Section 17.6.1). In general, this is an expensive run-time option to enable and may result in unacceptable performance loss.

Table 17.8 shows example output for simulation object `S_Car_MGR 0`. The column definitions are identical to those used in the event data summary (see Section 17.4.3).

| Event name | Nproc | Ncmtd | Tcpu | Tcmtd | <Tcmtd> | Eff | MaxProc |
|---|---|---|---|---|---|---|---|
| Car_StopCar | 173 | 20 | 0.03135 | 0.01384 | 0.00069 | 0.44154 | 0.01212 |
| Car_Go | 498 | 61 | 0.00800 | 0.00195 | 3.2e-05 | 0.24478 | 0.00025 |
| Car_Stop | 483 | 60 | 0.01611 | 0.00234 | 3.9e-05 | 0.14547 | 0.00012 |
| Car_RadioController | 224 | 10 | 0.00908 | 0.00044 | 4.4e-05 | 0.04855 | 6.9e-05 |
| Car_RadioStation | 1112 | 49 | 0.00971 | 0.00048 | 9.8e-06 | 0.04990 | 4.5e-05 |
| Radio_On | 125 | 6 | 0.00288 | 0.00027 | 4.6e-05 | 0.09717 | 0.00016 |
| Radio_Scan | 1063 | 46 | 0.03683 | 0.00173 | 3.7e-05 | 0.04705 | 8.2e-05 |
| Radio_SendFrequency | 1109 | 49 | 0.01548 | 0.00072 | 1.4e-05 | 0.04700 | 5.0e-05 |
| SpCancel | 406 | 43 | 0.00880 | 0.00114 | 2.6e-05 | 0.12955 | 4.1e-05 |

Table 17.8: Object and Event Data Summary Diagnostic Output Data

### 17.4.5 Automatic Lazy Re-evaluation Statistics

User designed lazy re-evaluation was discussed in detail in section 15.2.1. SPEEDES has the ability to automatically determine when an event may be lazily rolled forward and does this through a system of touch and depend analysis of every event.

Automatic lazy re-evaluation by default is not enabled. Compile time option of `-DUSE_AUTO_LAZY` will enable Automatic lazy re-evaluation. This causes rolled back events to be rolled forward without reprocessing if the final result of the event does not change. In addition, statistics for these types of events can be displayed. Once again, the code for capturing the statistics data must be compiled into the SPEEDES framework with compile time option of `-DAUTO_LAZY_PERF`. Once SPEEDES and the application have been compiled, the following parameters must be added to file `speedes.par`.

```
parameters {
  string  auto_lazy             ALL_EVENTS_ENABLED
  int     auto_lazy_threshold   10
}

statistics {
  logical AutoLazyEvaluation    T
}
```

Once automatic lazy re-evaluation has been enabled, SPEEDES begins the automatic lazy re-evaluation of all events. It does this by recording a depend every time a rollbackable variable or structure is read or accessed. It also records a depend whenever a rollbackable variable or structure is modified. With this information, SPEEDES is able to automatically determine whether a straggler touched variables upon which the current event depended. On the basis of that information, it can determine whether or not to roll the current event forward or to send anti-messages and reprocess the event.

SPEEDES automatic lazy re-evaluation algorithm is designed to be as efficient as possible, hence it only records touches when a straggler is processed rather than recording all touches all of the time. It also disables roll forward if an event has implemented the `exchange` method because these touches cannot be automatically tracked. Finally, event orderings may be different when automatic lazy re-evaluation is enabled because the automatic tie-breaking fields, as decribed in Section 6.1, cannot be properly implemented. However, this should only be an issue for events with identical floating point times.

Figure 17.9 and Figure 17.10 show the automatic lazy statistics output for the car and stop light example.

| Event Type | Processed | Roll Forwards | Reprocesses | # Events Sched | Add Depends | Add Touches | AutoLazy Evals |
|---|---|---|---|---|---|---|---|
| Car_StopCar | 200 | 0 | 0 | 0 | 1666 | 600 | 0 |
| Car_Go | 244 | 0 | 0 | 0 | 927 | 488 | 0 |
| Car_Stop | 240 | 0 | 0 | 0 | 1739 | 720 | 0 |
| Car_RadioController | 37 | 78 | 0 | 156 | 1462 | 1184 | 78 |
| Car_RadioStation | 228 | 541 | 0 | 0 | 45 | 0 | 541 |
| StopLight_TurnsRed | 60 | 198 | 0 | 838 | 440 | 60 | 198 |
| StopLight_TurnsGreen | 60 | 202 | 0 | 854 | 440 | 60 | 202 |
| Radio_Off | 3 | 13 | 0 | 0 | 0 | 3 | 13 |
| Radio_On | 25 | 54 | 0 | 54 | 87 | 75 | 54 |
| Radio_Scan | 211 | 512 | 0 | 1011 | 2105 | 1055 | 512 |
| Radio_SendFrequency | 228 | 542 | 0 | 542 | 932 | 228 | 542 |
| RollBack | 36 | 0 | 0 | 0 | 220 | 36 | 0 |
| SpHandler | 36 | 0 | 0 | 0 | 189 | 0 | 0 |
| SpCancel | 54 | 0 | 9 | 0 | 56 | 63 | 9 |
| Totals | 1662 | 2140 | 9 | 3455 | 10308 | 4572 | 2149 |

Table 17.9: Automatic Lazy Evaluation Statistics Part I

| Event Type | <T_proc> | T_depend | <T_depend> | T_touch | <T_touch> | T_eval | <T_eval> |
|---|---|---|---|---|---|---|---|
| Car_StopCar | 0.00015 | 0.00161 | 9.6e-07 | 0.00167 | 2.7e-06 | 0 | 0 |
| Car_Go | 2.8e-05 | 0.00099 | 1.0e-06 | 0.00127 | 2.6e-06 | 0 | 0 |
| Car_Stop | 7.6e-05 | 0.00168 | 9.6e-07 | 0.00225 | 3.1e-06 | 0 | 0 |
| Car_RadioController | 0.00022 | 0.00139 | 9.5e-07 | 0.00104 | 8.8e-07 | 0.00642 | 8.2e-05 |
| Car_RadioStation | 1.6e-05 | 4.4e-05 | 9.9e-07 | 0 | 0 | 0.00095 | 1.7e-06 |
| StopLight_TurnsRed | 9.3e-05 | 0.00044 | 1.0e-06 | 6.0e-05 | 1.0e-06 | 0.00040 | 2.0e-06 |
| StopLight_TurnsGreen | 9.1e-05 | 0.00043 | 9.8e-07 | 5.7e-05 | 9.5e-07 | 0.00043 | 2.1e-06 |
| Radio_Off | 2.7e-05 | 0 | 0 | 2.9e-06 | 9.6e-07 | 2.2e-05 | 1.7e-06 |
| Radio_On | 5.5e-05 | 8.3e-05 | 9.5e-07 | 7.7e-05 | 1.0e-06 | 9.7e-05 | 1.7e-06 |
| Radio_Scan | 7.8e-05 | 0.00202 | 9.6e-07 | 0.00103 | 9.8e-07 | 0.00778 | 1.5e-05 |
| Radio_SendFrequency | 3.1e-05 | 0.00091 | 9.7e-07 | 0.00023 | 1.0e-06 | 0.00366 | 6.7e-06 |
| RollBack | 0.01682 | 0.00024 | 1.0e-06 | 4.2e-05 | 1.1e-06 | 0 | 0 |
| SpHandler | 0.00021 | 0.00019 | 1.0e-06 | 0 | 0 | 0 | 0 |
| SpCancel | 4.3e-05 | 5.5e-05 | 9.9e-07 | 0.00018 | 2.9e-06 | 0 | 0 |
| Totals | 0.01796 | 0.01012 | 9.8e-07 | 0.00794 | 1.7e-06 | 0.01979 | 9.2e-06 |

Table 17.10: Automatic Lazy Evaluation Statistics Part II

Column header definitions for Table 17.9.

Column 1. `Event Type`:
The name of the event.

Column 2. `Processed::`
The number of times the event type was processed by the simulation. This number includes reprocesses of a single scheduled event, and should match `Nproc` from the `EventProcessing` statistics.

Column 3. `Roll Forwards`:
The number of times the event type was rolled forward instead of reprocessed following an automatic lazy re-evaluation.

Column 4. `Reprocesses`:
The number of times the event type was reprocessed instead of rolled forward following an automatic lazy re-evaluation.

Column 5. `# Events Sched`:
The number of events scheduled by events of the event type that have been re-evaluated by automatic lazy over the course of the simulation. For example, event `StopLight_TurnsRed` scheduled 838 events. Since this event was only processed 60 times, this event is a good event to roll forward to prevent all of the event cancelations required for its scheduled events.

Column 6. `Add Depends`:
The number of automatic lazy dependencies created by processing events of the event type over the course of the simulation. The average time for adding a dependency is given in the second half of the automatic lazy statistic table (shown in Table 17.10). In general, the largest overhead in automatic lazy are the dependency checks. Also, the more dependencies an event creates, the more likely it is the event will be reprocessed instead of rolled forward.

Column 7. `Add Touches`:
The number of automatic lazy touches, or modifications, created by processing events of the event type over the course of the simulation. The average time for adding touches for the event type is shown in the second half of the automatic lazy statistics table (shown in Table 17.10). If the number of touches for a particular event exceeds the `auto_lazy_threshold` parameter in `speedes.par`, events rolled back by that event will be automatically reprocessed instead of rolled forward. The threshold parameter defaults to 10. The idea is that, if a straggler event causes too many changes, it is assumed that there is going to be a match between the straggler's touches and the dependencies of the events that the straggler rolled back. Therefore, reprocess those events rather than wasting time looking for the match.

Column 8. `AutoLazy Evals`:
Automatic lazy evaluations result in either a roll forward or a reprocess. Therefore, this column is the sum of `Roll Forwards` and `Reprocesses`.

Column header definitions for Table 17.10.

Column 1. `Event Type`:
The name of the event.

Column 2. <T_proc>:
The average processing time spent per event of the event type.

Column 3. T_depend:
The total time spent adding automatic lazy dependencies while processing events of the event type.

Column 4. <T_depend>:
The average time spent adding automatic lazy dependencies while processing events of the event type.

Column 5. T_touch:
The total time spent adding automatic lazy touches while processing events of the event type.

Column 6. <T_touch>:
The average time spent adding automatic lazy touches while processing events of the event type.

Column 7. T_eval:
The total time spent conducting automatic lazy re-evaluations for events of the event type.

Column 8. <T_eval>:
The average time spent conducting automatic lazy re-evaluations for events of the event type.

## 17.5 Simulation Object Statistics

### 17.5.1 Simulation Object Data Summary

Similar to the event data summary diagnostic data (see Section 17.4.3), there is diagnostic data available for simulation objects. Instead of statistical information about each event type, this feature provides information about each simulation object type. To enable simulation object diagnostic data, add parameter `ObjectProcessing` to the `statistics` section in `speedes.par`, as shown below. Table 17.11 shows example output from the car and stop light simulation. Again, the names of the columns are the same as described in Section 17.4.3.

```
statistics {
  logical ObjectProcessing T  // Object processing statistics
}
```

| Object name | Nproc | Ncmtd | Tcpu | Tcmtd | <Tcmtd> | Eff | MaxProc |
|---|---|---|---|---|---|---|---|
| S_Car_MGR | 16326 | 1461 | 0.92095 | 0.10236 | 7.0e-05 | 0.11114 | 0.17468 |
| S_StopLight_MGR | 1107 | 156 | 0.31846 | 0.02893 | 0.00018 | 0.09085 | 0.22019 |
| S_RollBack_MGR | 36 | 36 | 1.38484 | 1.38484 | 0.03846 | 1 | 0.21344 |
| Totals | 17469 | 1653 | 2.62426 | 1.51614 | 0.00091 | 0.57774 | 0 |

Table 17.11: Simulation Object Summary Diagnostic Output Data

### 17.5.2 Object Placement Information

Using a block or scatter decomposition usually distributes the simulation objects in a relatively random fashion. However, this randomness makes it hard to identify where the objects are actually located. SPEEDES will print out this data so that users can identify interdependencies or help track down performance issues.

To enable this output, add a new section to the `statistics` section of `speedes.par` called `ObjNamesByNode` that looks like this:

```
statistics {
  ObjNamesByNode {
    string fileName ObjectMap
  }
}
```

This will generate files in the form of `ObjectMap_NODE_#`, where # is replaced by the node number. When the example in section 6.4 is run on two nodes with the above addition to `speedes.par`, it produces two files called `ObjectMap_NODE_0` and `ObjectMap_NODE_1`. These files are shown in Figures 17.5 and 17.6, respectively.

```
Only objects with names will be printed
Object manager is S_Car_MGR
 Object Name:"S_Car_MGR 0" Global Id:0 Object Handle:{0,0,0} Kindid:0
 Object Name:"S_Car_MGR 2" Global Id:2 Object Handle:{0,0,1} Kindid:2
Object manager is S_StopLight_MGR
 Object Name:"S_StopLight_MGR 0" Global Id:4 Object Handle:{0,1,0}
                                                              Kindid:0
```

Figure 17.5: File ObjectMap_NODE_0

```
Only objects with names will be printed
Object manager is S_Car_MGR
 Object Name:"S_Car_MGR 1" Global Id:1 Object Handle:{1,0,0} Kindid:1
 Object Name:"S_Car_MGR 3" Global Id:3 Object Handle:{1,0,1} Kindid:3
Object manager is S_StopLight_MGR
```

Figure 17.6: File ObjectMap_NODE_1

Some unnecessary white space was eliminated in the above text in order for the data to fit within the figure. This output shows that the cars with kind ids of 0 and 2 were created on node 0 while the cars with kind ids of 1 and 3 were created on node 1. Finally, the only stoplight was created on node 0.

## 17.6 Interval Statistics

### 17.6.1 Event and Object Statistics

Interval event and object processing statistics can be displayed as the simulation is executing by adding section `IntervalOutputTime` to section `statistics` in `speedes.par`, as shown below:

```
statistics {
  IntervalOutputTime {
    float SimTime 10.0                         // Interval output time
    OutputFileName {
      string StatisticsFileName  intervalStats // Result filename
    }
  }
}
```

This section indicates that the data should be collected approximately every 10.0 seconds. Section `OutputFileName` is an optional section. If it is present, the output of the simulation will be saved in the file specified by `StatisticsFileName`, which in this case is `intervalStats`. If it is not present, then the output goes to stdout. This output contains all the information described in Section 17.4 but only over the previous interval of time. This data can then be used to profile the behavior of specific events or objects over the life of the simulation. For example, suppose an event's average CPU time is less than 1 milliseconds (ms) over a 2 hour simulation run. However, this event may spike to over 100 ms or even 1000 ms at given times. This sort of behavior is masked by the global statistics, but becomes very visable when interval statistics are examined.

The overhead for displaying these statistics is usually quite small (usually less than a few milliseconds), and should not prohibit system analysts from using them. The tabulation of the data displayed occurs outside of the event processing cycle. Hence, these data point calculations do not have an affect on the displayed data.

### 17.6.2 Rollback Statistics

Information about all events which rollback other events can be saved off to output files by setting parameter `RollbackBreakdownByEventType` in section `statistics` in `speedes.par`, as shown below:

```
statistics {
  RollbackBreakdownByEventType {
    float OutputIntervalSimtime 100.0 // Minimum simulation time
                                      //  interval
    int   WidthOfPrintedMatrix  80    // Number of columns in printed
                                      //  table
    OutFileName {
      string RollbackBreakdownFileName rbTable // Base filename
    }
  }
}
```

Parameter `OutputIntervalSimtime` specifies the time interval at which the data will be output to the data file whose base name is specified by `RollbackBreakdownFileName`. This base name is appended by the node number to create the actual file name. Table 17.12 shows the output for the car and stop light simulation for node 0 at one time interval.

| **Event Name** | **Event Id** | **Rolled Back Event Id** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | **0** | **1** | **2** | **3** | **4** | **7** | **8** | **9** | **10** | **21** |
| Car_StopCar | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Car_Go | 1 | 9 | 12 | 14 | 2 | 18 | 0 | 2 | 16 | 18 | 4 |
| Car_Stop | 2 | 0 | 0 | 0 | 13 | 81 | 1 | 8 | 78 | 81 | 4 |
| Car_RadioController | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Car_RadioStation | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Radio_Off | 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Radio_On | 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Radio_Scan | 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Radio_SendFrequency | 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| SpHandler | 20 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| SpCancel | 21 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Table 17.12: Event Rollback Diagnostic Output Data

Each row specifies the event by name with its respective event integer id in the `Event Id` column. Each number at a specific column location specifies how many times the event in that row rolled back the event listed at the top of the column. For example, the quantity of 13 in row `Car_Stop` specifies that event `Car_Stop` rolled the event whose id is 3 (i.e. `Car_RadioController`), back 13 times.

# Part VII

# Appendix

# Appendix A

# Parallel Discrete-Event Simulation Technical Reference

## A.1 What is Discrete-Event Simulation

Discrete-event simulation is a very powerful technique for modeling the intricate interactions of complex systems.[1] The fundamental building blocks of discrete-event simulation are events. Once a discrete-event simulation has been initialized, all application-specific calculations evolve over time through the processing of time-stamped events that occur at discrete points in simulated time[2].

Each event is assigned a logical time stamp that determines when it is processed. To preserve causality, events are processed in ascending logical time order. An event queue data structure manages the correct event processing order in logical time. Events can potentially do two things:

- Modify the state of the system.
- Schedule new events for the present, or for the future, but not in the past.

One of the benefits of discrete-event simulation over time-stepped simulation [3] is that updates can be scheduled to occur only when necessary. This capability allows simulation developers to optimize their run-time performance by carefully coordinating when changes occur in the modeled system. Discrete-event simulation decouples complex models by allowing them to evolve independently over simulated time. Discrete-event simulation therefore supports model behavior in a more realistic and natural manner than time-stepped simulation.

## A.2 Sequential Discrete-Event Simulation (SDES)

Sequential Discrete-Event Simulation (SDES) simply means that events are processed sequentially on a single processor. This is in contrast to PDES where events are processed in parallel on multiple

[1]A system may be formally composed of sub-systems, which can further be composed into sub-sub-systems, etc., and models that interact in a coordinated manner to represent the physical system. Another way of saying this is that systems can be hierarchical, but they always decompose into models that represent the physical system.

[2]Simulation time and logical time are synonyms. These terms are equally used to specify the simulated time of an event.

[3]Time-stepped simulations normally update the entire state of the modeled system at regular time intervals. Time advances using a `for-next` loop with a constant step. In contrast, discrete-event simulations decouple processing through the use of irregularly time-stamped events that only updates what needs to be updated when it needs to be updated.

processors. The normal processing flow of a SDES is shown below in Figure A.1.

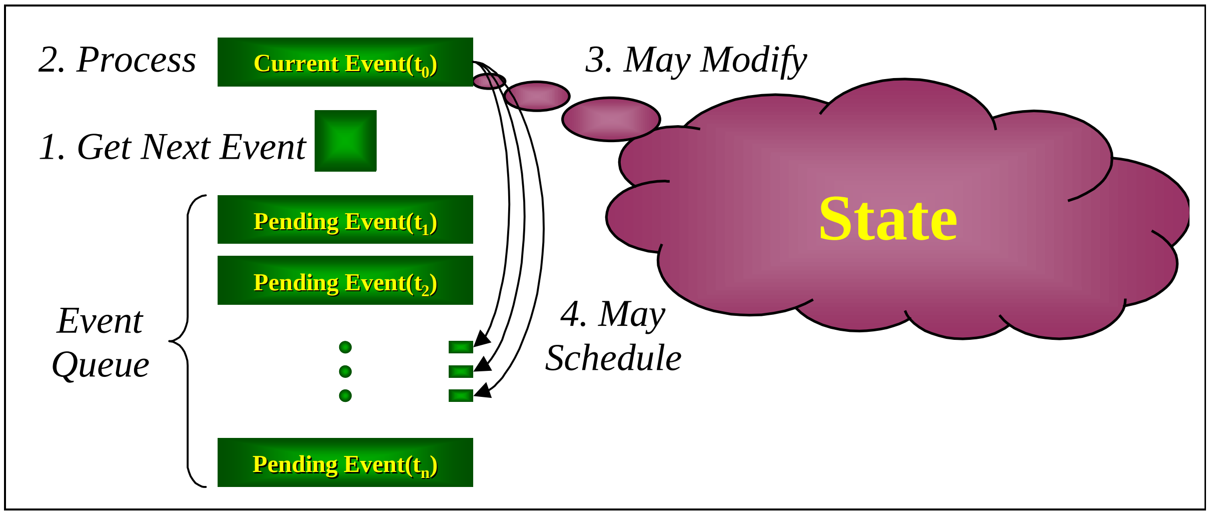


Figure A.1: Processing Flow of a Discrete-Event Simulation

All pending events are managed in an event queue. In step 1, the event with the earliest time stamp is removed from the event queue. The current simulation time of the system is advanced to this event's time value. In step 2, the event is processed. Steps 3 and 4 flow from the processing of the event. Events may modify the state of the system and they may schedule new events for the present or for future times. These four steps are continually repeated until the simulation either reaches its end time, or until some other termination condition is reached.

The event with the earliest time stamp is continually pulled out of the event queue and processed. This continues until either the simulation reaches its end time or until some other termination condition is reached. As the current event is processed, it may modify the state of the system and/or it may schedule new events for the present or for the future.

In SDES, events are free to access and/or modify any of the system's state variables. This is not normally allowed in PDES because the system's state is distributed across multiple processors. Furthermore, even different subsystems or models on the same processor are not normally allowed to access and/or modify each other's state because they may be at different simulation times.

## A.3 SDES and PDES Trade-Offs

There are several obvious reasons why large simulation applications might consider using the PDES paradigm. At the top of the list is computational speed-up. Computational speed-up is defined to be the execution completion time that an application takes to run on one processor divided by the same time it takes to run on $N$ processors. In other words, *speed-up* = $T_1/T_N$. The parallel processing efficiency is defined to be the computational speed-up divided by the number of processors. In other words, *efficiency = speed-up/N*.

However, it is important to understand that SDES are much simpler to implement than PDES. This equally applies to the supporting PDES infrastructure and to the actual application models representing the simulated system. There is no guarantee that every PDES application will achieve perfect speed-up. Some of the technical challenges involved in supporting PDES are listed in Table A.1. SPEEDES addresses all of the technical issues listed in Table A.1, while providing a rich object-oriented modeling framework for users to develop their application models. Some of the other potential benefits of PDES include:

- Faster execution run times.
- Larger address spaces.
- More disciplined object-oriented approach.
- Interoperatability with other distributed simulations.

| Issue | SDES | PDES |
|---|---|---|
| System State | Events can access or modify the entire state of the system. | Events can only access or modify the state of a single simulation object. This is because simulation objects are distributed across multiple processors and may also be at different times. |
| Object Interactions | Any simulation object, at anytime, can interact with any other simulation object without a performance penalty. | Simulation objects are either limited by how tightly they can interact in time, or they must use rollback-based techniques to recover from time accidents caused by straggler messages generated by events on other processors. |
| Communication | Events are scheduled locally. | Events scheduled for simulation objects on remote processors must send messages to be posted. |
| Synchronization | Simple next event processing approach. | Must coordinate event processing with messages received from other nodes while coordinating the advancement of global time. |
| Flow Control | None. | Must make sure that messages are routed through the network without excessive delays. Must also provide stability concerning synchronization messages, memory usage, and numbers of rollbacks. |

Table A.1: SDES vs. PDES

## A.4 PDES Challenges

The fundamental challenge of PDES is to efficiently process events concurrently on multiple processors, while preserving the overall causality of the system as it advances in simulated time. Each node must continually address the fundamental problem,

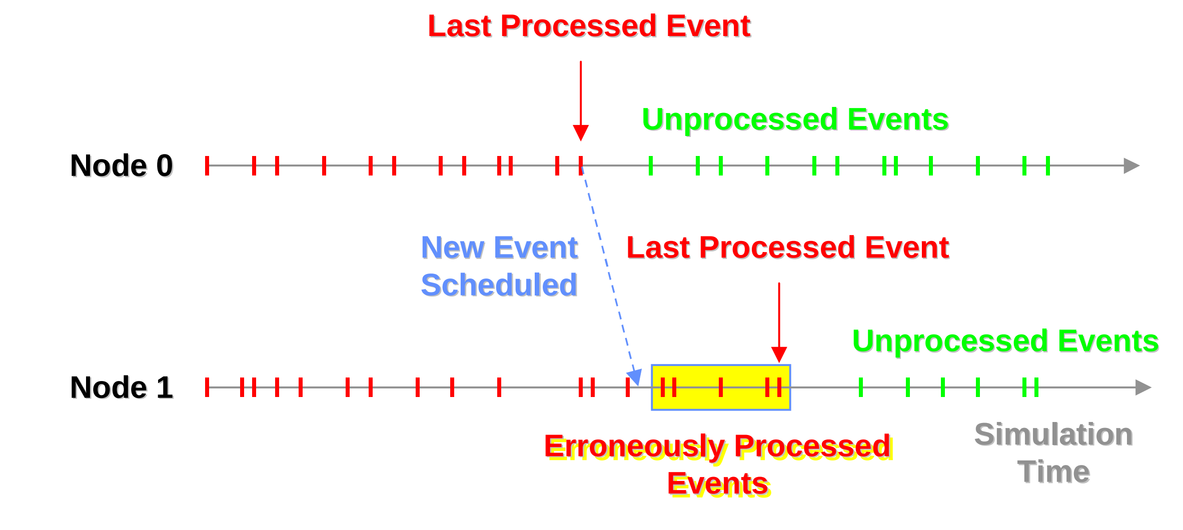
*Is it safe to process my next pending event, or will I receive an event message from another node with an earlier time stamp that must be processed first?*

In the most general situation, where any event can schedule an event for any other simulation object at any present or future time, addressing this problem may end up serializing the entire simulation if conservative techniques are used. The conservative approach encompasses any strategy that processes events only when it can be guaranteed that no straggler messages will arrive with an earlier time stamp. When considering all of the pending events waiting to be processed across all of the processors, only the event with the smallest time stamp can be safely processed. This is because it is still possible for that event to generate a new event with an earlier time stamp than all of the other pending events in the system.

Figure A.2 illustrates this problem. Imagine that two nodes are processing events without anytime synchronization. Almost certainly, one of the nodes will advance ahead of the other in simulated time.

Imagine now that an event on the slower node schedules an event for the faster node in its logical past. This would violate causality since events must be processed in their correct ascending time order.

The challenge of PDES is to efficiently process events concurrently while preserving the overall causality of the simulated system. Two approaches are commonly used to solve this problem. The conservative approach only processes events when there is no chance of receiving a straggler message. The optimistic approach uses rollback techniques to undo the damage done when processing events out of order.



*Conservative approach:* ***Never allows events to be processed if it is possible for "straggler" messages to arrive from other nodes***

*Optimistic approach:* ***Fixes straggler message problem by rolling back state and canceling generated events***

Figure A.2: Synchronization in Optimistic PDES.

## A.5 Conservative Time Management

There are basically only two ways to synchronize PDES. The first approach is to ensure that each node never processes an event when it is possible to receive an event with an earlier time stamp from another node. This general strategy is often called the conservative approach. Conservative approaches place limitations on how simulated objects interact. Some of these limitations are described in Table A.2.

| Technique | Limitation |
|---|---|
| Lookahead | This approach limits how tightly simulation objects on different nodes are allowed to interact in simulation time. Assuming a lookahead value, *L*, a simulation object at time, *T*, would not be allowed to schedule an event for another simulation object on another node any earlier than time *T*+*L*. This limitation can degrade the fidelity of the simulation. Trying to work around this limitation by predicting when events occur only complicates the application software and still does not fix the problem when such predictions are wrong. |
| Topology | This approach limits which simulation objects can interact with which other simulation objects by defining a rigid interaction topology between them. A simulation object can only schedule an event for another simulation object if it is connected to it in the topology graph. For many applications, this constraint does not impose any real limitations. However, when virtually any simulation object can interact with any other simulation object, this constraint is unacceptable. |
| FIFO Interactions | The First In First Out (FIFO) technique imposes a rule for ordering event scheduling between simulation objects. All events scheduled by one simulation object for another simulation object must follow in ascending order as time evolves. An example of how this rule can be violated is if a simulation object, A, at time, 100, schedules an event for another object, B, at time, 200. Then at time 110, object A schedules another event for object B at time, 150. In this example, the event scheduling streams are crossed, thus violating the FIFO constraint. |

Table A.2: Frequent Limitations Imposed by Conservative Techniques

## A.6 Optimistic Time Management

A second, and more aggressive, approach allows nodes to process events optimistically, but then relies on a rollback mechanism to undo any erroneously processed events as stragglers are received. This is called the optimistic approach. Rolling back an event requires an advanced simulation engine that is capable of undoing changes made to state variables, while also retracting any events that were scheduled. SPEEDES applications handle this through use the built-in rollbackable data types (i.e., atomic data types, container classes, and dynamic memory operations) to implement their state in a rollbackable manner. SPEEDES then transparently performs rollback operations for applications as straggler messages arrive from other nodes.

When a SPEEDES node receives a straggler message, it does not rollback all of the locally processed events that have a greater time value. Instead, SPEEDES only rolls back the necessary events that were processed by the target simulation object. This is shown in Figure A.3. By limiting rollbacks in this manner, more parallelism is achieved. This results in fewer rollbacks and better overall parallel performance. However, it does mean that events must be associated with one, and only one, simulation object. Accessing and/or modifying the state variables of another simulation object in an event is forbidden.

To first order, rollbacks only affect the target simulation object. Notice in Figure A.3 that even though Node 1 receives a straggler message in its past from Node 0, only Object 7 rolls back its one event that was processed out of time order. Of course, the event rolled back might induce further rollbacks. Also, it is possible that when this straggler event is processed, it may generate more events that cause further rollbacks, but this is also a second order effect. In this figure, GVT stands for Global Virtual Time, and LVT stands for Local Virtual Time. GVT is globally advanced by all of the nodes in a periodic manner. LVT is locally known by each of the nodes and can sometimes go backwards in time.

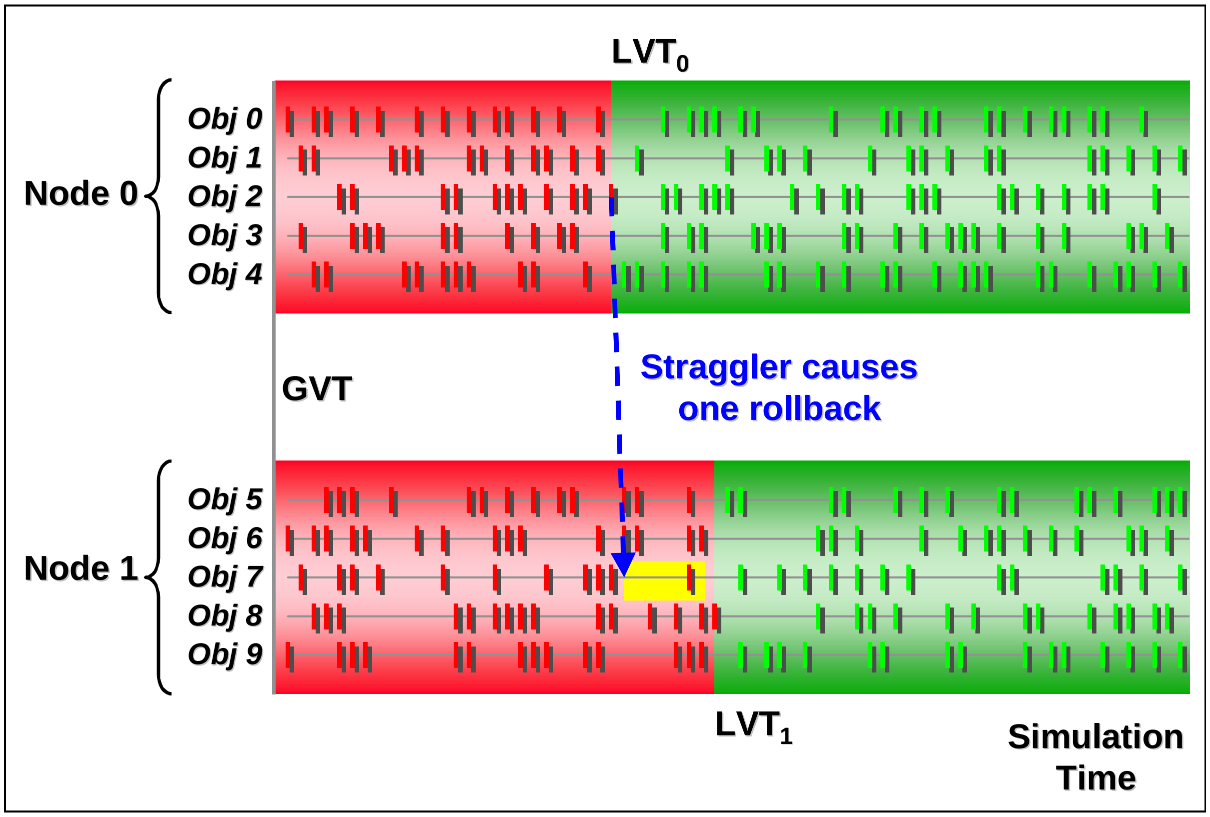


Figure A.3: Stragglers only Rollback a Single Object Rather Than Whole Node

Figure A.4 shows how rollbacks are actually implemented. When a simulation object receives a straggler message, only the optimistically processed events with time stamps greater than the incoming straggler are rolled back Events are always rolled back one at a time and in reverse order. This is very similar to commercial software applications that provide undo capabilities. By undoing a series of actions, users can restore their work to earlier states. State variables affected by each event are restored to their original values and then all generated events are retracted. SPEEDES automatically retracts locally generated events directly through pointers and uses anti-messages to cancel events generated for simulation objects on other nodes.

If those events scheduled other events, then those scheduled events are retracted either directly for locally scheduled events, or by sending anti-messages to cancel events that were scheduled for simulation objects residing on other nodes. It is possible for a simulation to receive an anti-message for an event that it has already processed. In this case, more rollbacks may be generated, which may generate more anti-messages in a cascading manner. SPEEDES provides flow control over optimism and message-sending risk to provide stability and eliminate rollback explosions.

Anti-messages may induce further rollbacks, which in turn may generate even more rollbacks, etc. Such cascading rollbacks may become unstable if flow control is not provided by the simulation infrastructure. SPEEDES provides this flow control by limiting optimism and message-sending risk.

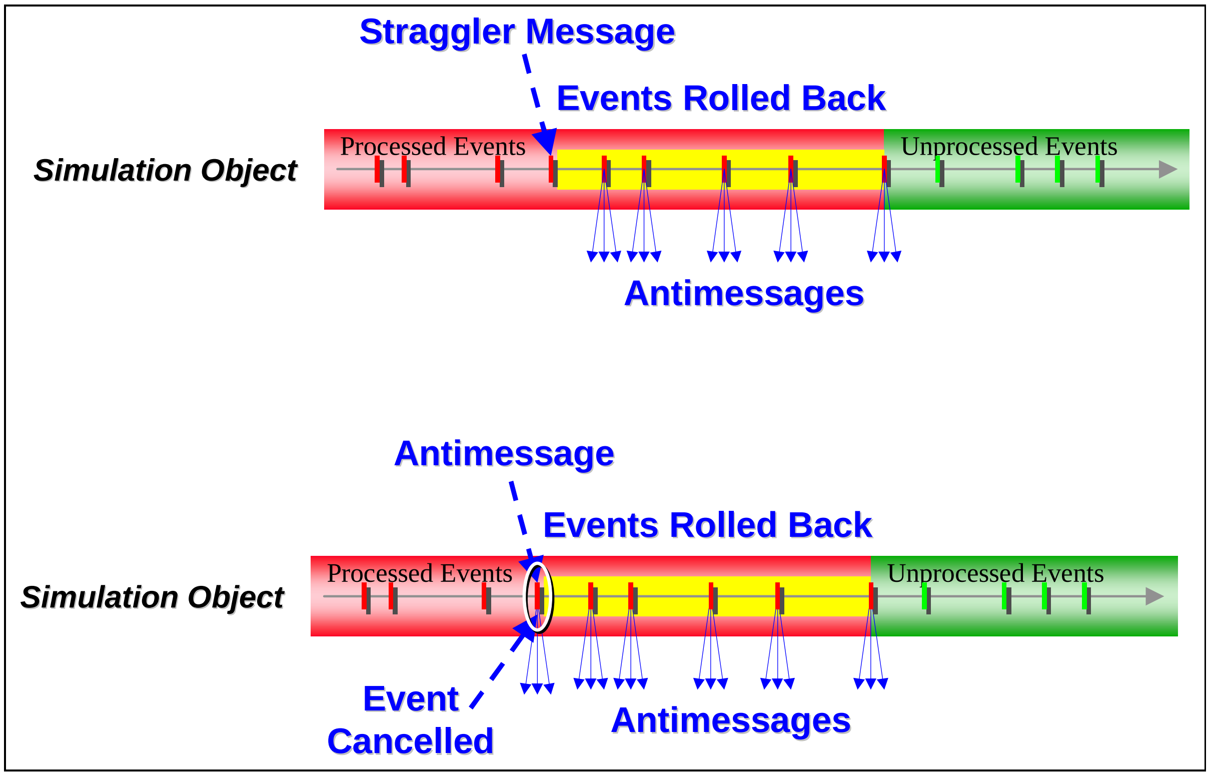


Figure A.4: Rolling Back a Simulation Object

## A.7 Time Warp and Breathing Time Buckets

The study of optimistic time management algorithms began in 1984. The first of these algorithms, called Time Warp, showed great promise. However, for real-world kinds of problems, the Time Warp algorithm often showed signs of instability due to excessive overheads and poor workload imbalances.

In 1990, SPEEDES began by investigating how optimistic event processing could be accomplished without requiring anti-messages. The Breathing Time Buckets algorithm processed events optimistically, but only released off-node messages when it was safe to do so. The key to the Breathing Time Buckets algorithm is the Event Horizon, shown in Figure A.5.

In Figure A.5, three event horizon cycles are shown. Each cycle processes its pending events while collecting newly generated events in an auxiliary event queue. When the next event to be processed is in the auxiliary event queue, the auxiliary event queue is sorted, merged into the primary event queue and then the next cycle begins. Cycle 3 shows that, even when an event schedules a new event for the same time as itself, the event horizon algorithm always processes at least one event. Thus, there is no way for a deadlock situation to occur when processing events in cycles defined by the event horizon.

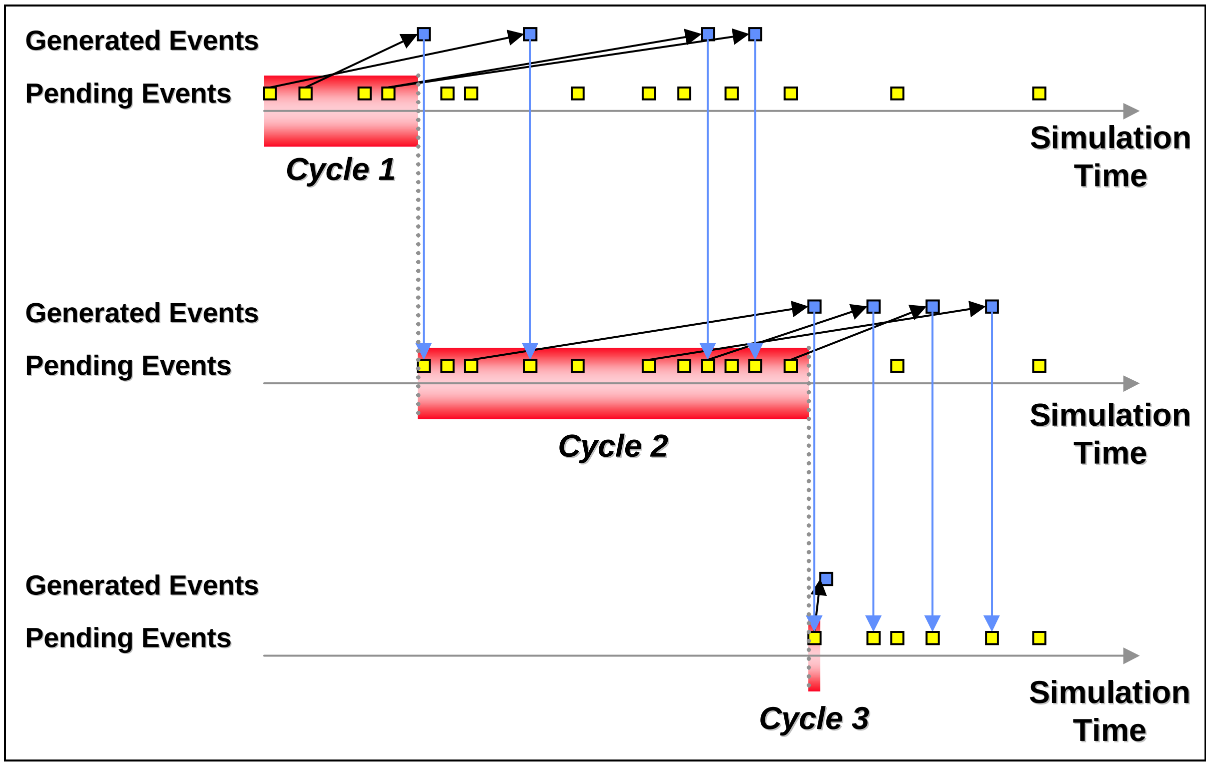


Figure A.5: The Event Horizon

The event horizon is a concept that can first be understood without referring to parallel processing. Imagine first that all pending events are mapped into a single event queue. As each event is processed, it may generate zero or more new events with arbitrary time stamps. To maintain complete generality, it is assumed that the simulation engine has no way to predict what each event will do until it is processed. All newly generated events are maintained in a special auxiliary event queue. At some point in time, the next event to be processed will be in the auxiliary event queue. This point in time is called the event horizon.

One of the key features of the event horizon is that, by definition, events in a given cycle are not disturbed by other events generated in the same cycle. If the events were distributed to multiple processors, and if the event horizon was known before hand, it would be possible to process events in parallel without ever receiving a straggler message. However, the event horizon is not determined until the events in the current cycle are actually processed. The Breathing Time Buckets algorithm resolved this dilemma by merging optimistic processing with risk-free messaging sending. The basic algorithm is shown in Figure A.6.

Figure A.6 shows each node starting the cycle using the same global time value. Events are then processed optimistically without releasing any of their generated messages. Each node determines its local event horizon using its own set of pending events and generated messages. The global event horizon is the minimum of all local event horizons. Once the global event horizon is determined, the Breathing Time Buckets algorithm safely releases all messages generated by events with time stamps less than or equal to that value. As shown for Node 0, it is still possible for events to be rolled back by stragglers. When this occurs, state information is restored and unsent messages are thrown away. Anti-messages are never necessary because messages are only released when events are committed.

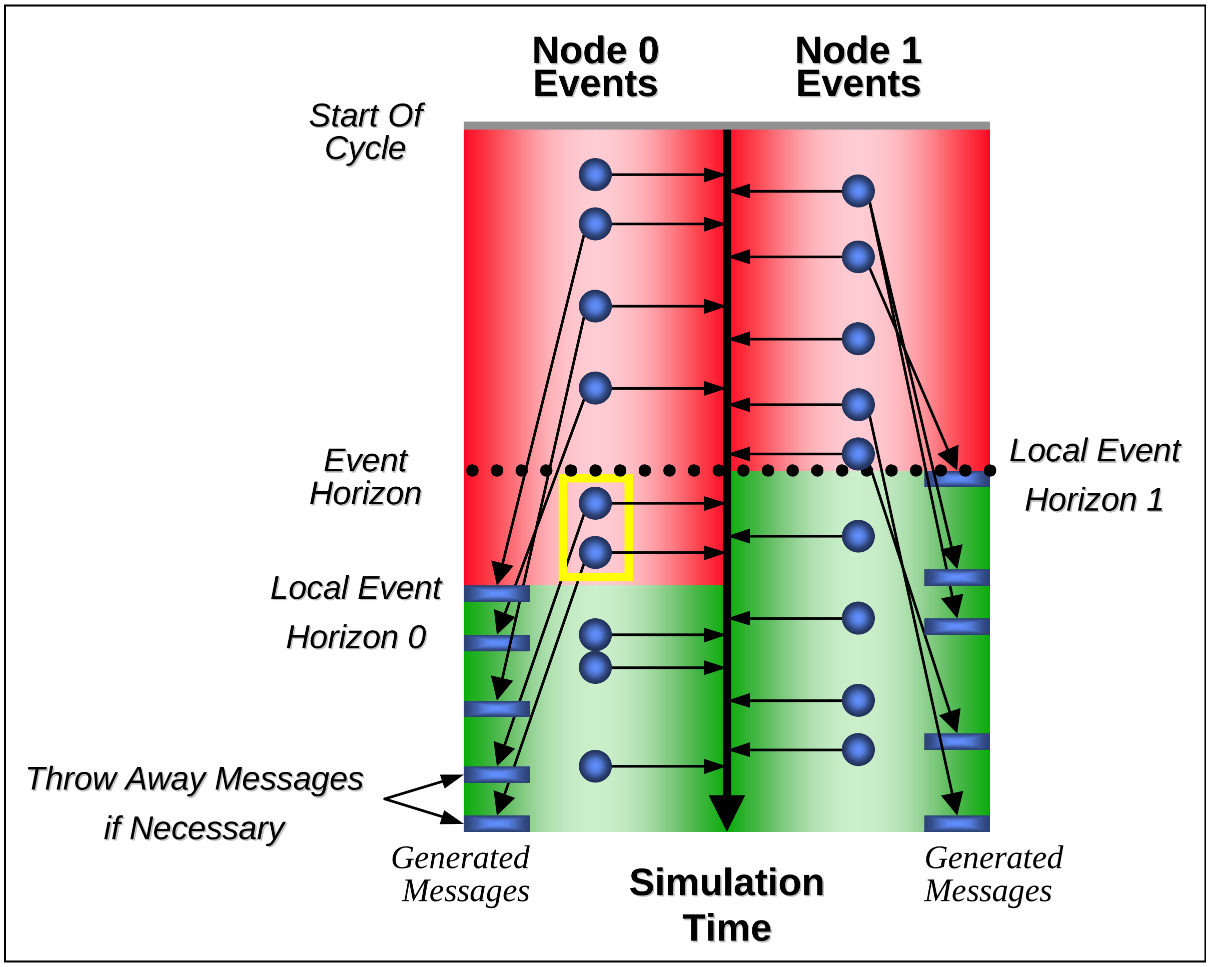


Figure A.6: The Breathing Time Buckets Algorithm Processes in Event Horizon Cycles

Breathing Time Buckets starts each cycle by processing events optimistically without releasing messages. Each node determines its local event horizon by tracking the time value of its earliest unsent message. Locally scheduled events do not count in the event horizon determination. This greatly extends the number of events processed in each event horizon cycle without causing excessive overheads. This value is called $T_{min}$. When the node's next event has a time value greater than $T_{min}$, the node defines its local event horizon to be $T_{min}$. The true global event horizon is the minimum of each node's local event horizon.

Once a node crosses its local event horizon, it then broadcasts its local event horizon time to the other nodes. This may cause other nodes to realize that they too have crossed the event horizon. If another node later determines that its local event horizon has an earlier time value, then it, too, broadcasts its local event horizon to the other nodes. SPEEDES provides flow control by staggering these broadcasts to ensure scalability.[4]

Once all of the nodes have crossed the event horizon, they simultaneously break out of their optimistic event processing and begin to safely release all unsent messages generated by events with time tags less than or equal to the event horizon. This step is where events are committed. A coordinated message-sending algorithm ensures that each node receives all of its incoming messages before beginning the next cycle. Rollbacks may occur as straggler messages arrive, but their effect on rollbacks is limited

[4]The time between asynchronous broadcasts is automatically staggered by SPEEDES and can be set by users. `Tasb` asynchronous broadcasts can also be disabled if desired.

locally. Events that are rolled back simply restore their state and then discard their unsent messages.

The one drawback of Breathing Time Buckets is the possibility that not enough events will be processed on the average in each cycle to remain efficient. The Breathing Time Buckets algorithm can suffer from the problem of too many synchronizations (see cycle 3 in Figure A.5). Keep in mind, however, that Breathing Time Buckets will always process at least one event, so there is no chance for deadlocks to occur. Like Time Warp, Breathing Time Buckets places no event scheduling constraints on the programmer (e.g. global lookahead or limited object interactions), but its performance will be poor if very few events are processed on the average per cycle. Simulations that frequently model interactions between objects on different nodes with very tight time scales exacerbate this problem.

## A.8 Breathing Time Warp and Flow Control

Time Warp and Breathing Time Buckets are limited in their ability to support general-purpose PDES applications. However, these two approaches actually suffer from opposite problems.

- Time Warp suffers from cascading anti-message explosions that can easily become unstable. In other words, Time Warp sends its messages in a manner that is too risky.
- Breathing Time Buckets suffers from holding back too many of its messages, which can result in too many synchronizations. In other words, Breathing Time Buckets does not provide enough risk in its message sending.

The obvious solution to the inherent problems of each approach is to combine their strengths into one algorithm that tunes the message-sending risk. This is precisely what BTW does.

BTW starts out each cycle by processing events using the Time Warp approach. In other words, messages are sent with risk[5]. Each node starts holding back its messages once the number of optimistically processed (but uncommitted) events gets beyond $N_{risk}$. At this point, the Breathing Time Buckets algorithm kicks in. Once GVT is updated, messages from events with time stamps less than or equal to GVT are released. The basic BTW event processing cycle is shown in Figure A.7.

Figure A.7 shows the cycle starting out with each node processing events with message-sending risk using the Time Warp algorithm. If any these events are rolled back, anti-messages may be required. After a node optimistically processes `Nrisk` (configurable in `speedes.par`) events, it switches over to the Breathing Time Buckets algorithm where messages are held back. Here, events that are rolled back do not need to send anti-messages, since their messages were never released. At some point in time, it is determined to update GVT. Once GVT is updated, the Commit phase releases all unsent messages, frees state-saving resources, and is permitted to send data to the outside world.

[5]This means that anti-messages are required to retract messages that were sent with risk if the event is rolled back.

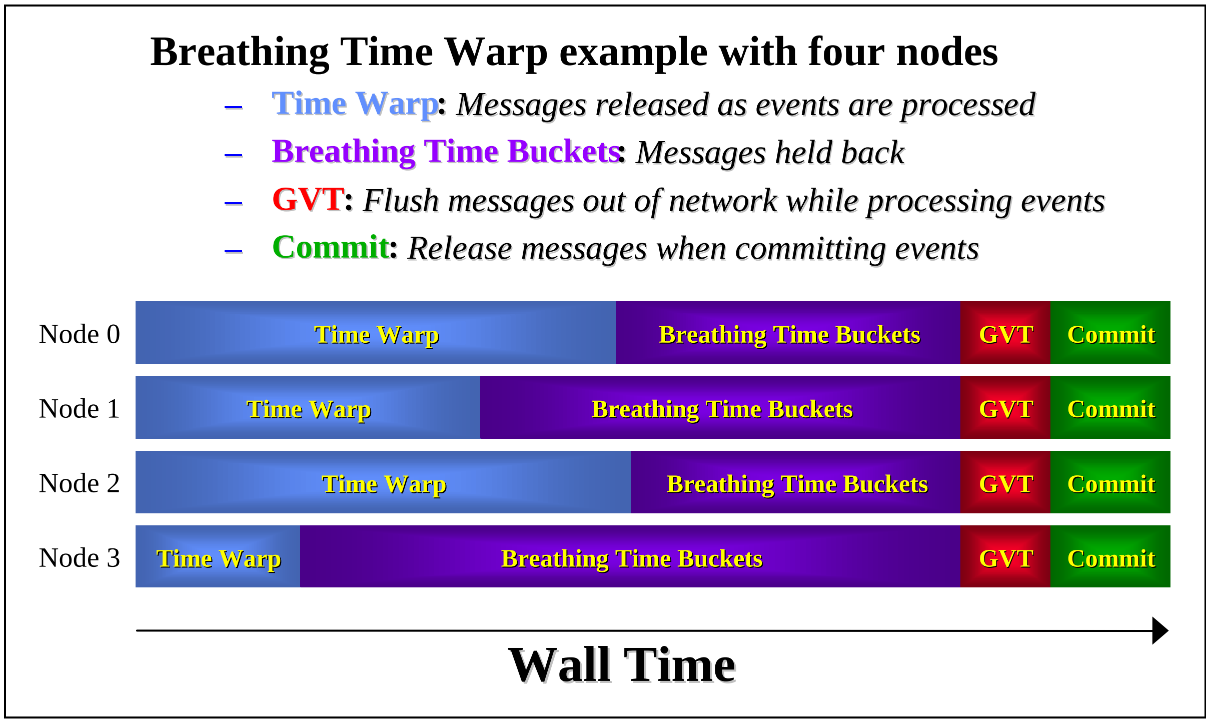

Breathing Time Warp example with four nodes
– Time Warp: Messages released as events are processed
– Breathing Time Buckets: Messages held back
– GVT: Flush messages out of network while processing events
– Commit: Release messages when committing events
Node 0
Node 1
Node 2
Node 3
Time Warp
Breathing Time Buckets
GVT
Commit
Wall Time

Figure A.7: The Breathing Time Warp GVT Cycle

A number of conditions may cause a GVT update. If any of the following conditions occur, BTW breaks out of its event processing and updates GVT.

- A specified amount of wall time, $T_{GVT}$ elapses (`Tgvt` in `speedes.par`).
- Each node has $N_{GVT}$ processed, but uncommitted events.
- The event horizon is crossed by all of the nodes
- There are no more events to process.

One common question raised by many SPEEDES users is, "what is the optimal setting for $N_{risk}$"? There are two ways to understand the effect of $N_{risk}$ on the BTW algorithm.

- By allowing $N_{risk}$ events to be processed before starting the Breathing Time Buckets algorithm, the problem of too many synchronizations is solved. At least $N_{risk}$ events are processed during each cycle. Even though at least events are processed during each cycle, this does not guarantee that $N_{risk}$ events are committed during each cycle. Rollbacks may limit the actual number of events committed. Notice that if $N_{risk}$ is set to zero, BTW reduces to Breathing Time Buckets.
- By providing a mechanism to reduce the message sending risk in Time Warp, cascading anti-messages are effectively limited. $N_{risk}$ prevents Time Warp from becoming unstable. Notice that if $N_{risk}$ is set to infinity, BTW reduces to Time Warp.

In most instances, $N_{risk}$ can be set somewhere between 50 to 100 without losing significant performance due to excessive event horizon synchronizations. On the other hand, $N_{risk}$ can usually be set as high as 2000 without losing performance due to cascading anti-message explosions. This bathtub effect means that users are not required to fine-tune their settings of $N_{risk}$. In other words, a wide range of settings will work fine.

As applications use more processing nodes, the potential for rollbacks increases. For example, the same simulation running on 4 nodes almost always experiences fewer rollbacks than when executing on 16 nodes. This means that the value for $N_{risk}$ may need to be decreased to throttle anti-messages when executing on large numbers of nodes. At some point, $N_{risk}$ may be so low that it impacts performance. In such cases, the application is most likely approaching its maximum speed-up, which is dictated by the critical path of its run-time execution.

# Appendix B

# The Conservative Time Management Algorithm

The conservative time management algorithm offers a simpler, non-patented alternative to the Breathing Time Warp algorithm. The term ”conservative” is in contrast to ”optimistic.” The essential difference between the two is that optimistic algorithms process events without certainty that they are valid (and therefore they sometimes need to roll back events), while conservative algorithms process only valid events, thereby never needing to roll them back. The advantage to the user is that he needs not write rollbackable simulations. The disadvantage to the user is that he must specify a minimum lookahead value (`float MinLookahead` in the `parameters` section of the `speedes.par` file), representing the simulation time interval between the current event's simulation time and the simulation time at which events are scheduled by the current event.

## B.1 Setting the Appropriate Parameters in the `speedes.par` File

Running SPEEDES using the conservative time management algorithm requires that two parameters are set in the `parameters` section of the `speedes.par` file, namely `mode`mode and `MinLookahead`. The `mode` parameter must be set to `CONSERVATIVE`, while the `MinLookahead` parameter must be set to a non-zero, positive floating point number, representing the minimum time interval (defined as the ”lookahead”) between the current event and any events that it schedules (with the exception of self-scheduled events, which do not need a minimum lookahead). While a non-zero minimum lookahead parameter is required when running in conservative mode, it can also be specified in any other time management mode as well.

If and when events are scheduled in the past or with less lookahead than the minimum lookahead required, SPEEDES automatically increases the schedule time so as to equal the minimum lookahead required. To determine whether to be warned of these automatic corrections, an optional parameter, `LookaheadAutoCorrectDebug` may be set in the `DebugOutput`ection of `speedes.par`. The default setting for `LookaheadAutoCorrectDebug`, `NON_TERMINAL_WARNING`, specifies SPEEDES to print a warning message like the following:

```
---| WARNING: Event scheduled with insufficient lookahead
---| Scheduler Event ID = 0
---| Scheduler Event Time = {2, 0, 0, 2, 0}
---| Scheduled Event ID = 0
```

```
---| Scheduled Event Time = {3, 0, 0, 3, 2}
---| Lookahead Requirement = 2
---| Corrected Scheduled Event Time = {4, 0, 0, 3, 2}
```

When an event is scheduled in the past, the message looks similar, except that the first line says `---| WARNING: Event scheduled in the past`.

If you would prefer to have the simulation terminate whenever there is insufficient lookahead, set `LookaheadAutoCorrectDebug` to `TERMINAL_ERROR`. In this case, SPEEDES prints a message like the following just before terminating:

```
---| TERMINAL ERROR: Event scheduled with insufficient lookahead
---| Scheduler Event ID = 0
---| Scheduler Event Time = {2, 0, 0, 2, 0}
---| Scheduled Event ID = 0
---| Scheduled Event Time = {3, 0, 0, 3, 2}
---| Lookahead Requirement = 2
```

Finally, if you would prefer SPEEDES to just correct schedule times as needed, continuing without printing warnings at all, set `LookaheadAutoCorrectDebug` to `NONE`.

## B.2 Minimum Lookahead Global Functions

`SpGlobalFunctions.H` provides two useful functions when using minimum lookahead. First, to obtain what the minimum lookahead time is (as specified in `speedes.par`), call `SpGetMinLookahead()`. To obtain the simulation time equal to the current time (including the user priority fields) plus the minimum lookahead, call `SpGetMinLookaheadTime()`. These functions are useful when writing generic events that may be used with any minimum lookahead setting, since they relieve the need to hard-code the lookahead used when scheduling events.

## B.3 Proxy Updates, Undirected Handlers, and DDM

Proxies updates are always delivered with minimum lookahead with respect to when the attribute they reflect was modified. Undirected handlers, like all other non-self-scheduled events, must be scheduled with minimum lookahead, and all handler will be invoked at that time (including handlers invoked on the same simulation object that scheduled them). The lookahead added for all DDM events is undefined.

## B.4 Special Event Plug-in Calls (Advanced Topic)

When requiring a non-zero minimum lookahead, there are two alternatives to plugging in events using `PLUG_IN_EVENT`. The first alternative, `PLUG_IN_EVENT_WITH_LOOKAHEAD_BYPASS`, can be employed for events that are scheduled locally (meaning on the same node). Such events will not be required to be scheduled with minimum lookahead. This feature takes advantage of the fact that the conservative algorithm does not technically require minimum lookahead for local events, but only for events that are sent from one node to another. The reason SPEEDES normally requires minimum lookahead even for local events is for the sake of consistency, and so that runs will be repeatable when changing the number of nodes used (and thus the node placement of simulation objects). This plug-in call is not recommended unless one or more of the following are true: 1) the simulation does not require being repeatable when changing the number of nodes, 2) the event is always scheduled between simulation objects that are guaranteed to always be on the same node, guaranteeing repeatability (by means of the node placement feature, for instance), or 3) the event is scheduled on a simulation object which is always local, regardless of the number of nodes, guaranteeing repeatability.

The second alternative, `PLUG_IN_EVENT_WITH_LOOKAHEAD_ADJUSTMENT` can be used when an event is expected to be scheduled without enough lookahead, and the correction that SPEEDES makes is considered expected behavior. This plug-in call forces there to be no error or warning whenever SPEEDES makes the needed lookahead corrections, regardless of the setting for `LookaheadAuto-CorrectDebug` in `speedes.par`. However, if possible, it is preferable to schedule such events with a schedule time obtained with `SpGetMinLookaheadTime()`. This way, they are guaranteed never to need a lookahead adjustment.

# Appendix C

# SPEEDES Parameter File Configuration

The `speedes.par` file configures many of the run-time parameters for SPEEDES. The file itself, along with all of the sections and parameters contained in the file, are optional. Except where noted, the default values are presented.

## C.1 High Level SPEEDES Configuration (section parameters)

The `parameters` section controls the general behavior of the simulation.

```
parameters {
  string  mode                BREATHING_TIME_WARP
  int     n_nodes             1
  float   tend                3600.0
  float   lookahead           0.0
  logical scaled_time         F
  float   scaler              1.0
  logical statistics          T
  logical lazy                F
  float   spin                0.0
  logical optimize_sequential T
  string  auto_lazy           NO_OVERRIDE
  int     auto_lazy_threshold 10
}
```

| Parameter Name | Description |
|---|---|
| mode | This string defines the time management mode for the simulation. It can take one of the following values: `CONSERVATIVE`, `BREATHING_TIME_BUCKETS`, `TIME_WARP`, or `BREATHING_TIME_WARP`. |
| n_nodes | The number of nodes to be used. It can be overridden by the command line option `-n #` where # is the number of nodes. |
| tend | The simulation end time. |
| lookahead | If the value of `mode` is `CONSERVATIVE`, all events that are scheduled off-node will be scheduled at least `lookahead` seconds into the future. If this is violated, the time of the event is moved out to `lookahead` seconds. This parameter is required only if the value of `mode` is `CONSERVATIVE`. |

| Parameter Name | Description |
|---|---|
| scaled_time | If set to `T`, the simulation will run at the rate specified by the value of `scaler` (next definition). |
| scaler | This parameter, which is required only if `scaled_time` is set to `T`, specifies the multiple of wall clock time to which the rate of the simulation will be locked. If unable to maintain the specified rate, the simulation will not attempt to catch up. |
| statistics | If set to `T`, statistics, as specified in the `statistics` section (see Section C.4), will be printed on the command line. If set to `F`, only a minimal set of statistics will be printed, in particular, cycle number, wall clock time, and GVT. |
| lazy | If set to `T`, lazy evaluation of events will be determined on an event-by-event basis. Otherwise, no lazy evaluation is performed. |
| spin | Setting this parameter causes SPEEDES to spin in a busy loop for the specified number of seconds after every event. |
| optimize_sequential | If set to `T`, a specialized sequential algorithm is used when running on one node. Note: All external interactions may behave incorrectly if this flag is set to `T`. |
| auto_lazy | The string value of this parameter determines the automatic lazy re-evaluation mode for events. It can take any of the values `NO_OVERRIDE`, `ALL_EVENTS_ENABLED`, or `ALL_EVENTS_DISABLED`; corresponding to "use default settings", "automatic lazy enabled for all events", or "automatic lazy disabled for all events", respectively. Automatic lazy re-evaluation is not enabled by default in SPEEDES. It must be compiled in with the compile time option `USE_AUTO_LAZY`. |
| auto_lazy_threshold | Adds upper limits on the number of automatic lazy touches recorded automatic lazy re-evaluation diagnostic output. Automatic lazy re-evaluation diagnostics is not enabled by default in SPEEDES. It must be compiled in with the compile time option of `USE_AUTO_LAZY`. |

Table C.1: The "parameters" section of the speedes.par file

## C.2 GVT Configuration (section gvt_parameters)

The `gvt_parameters` section controls the behavior of the GVT update algorithm.

```
gvt_parameters {
  float Tgvt              1.0
  float Tasb              1.0
  int   Ngvt              100
  int   Nrisk             500
  int   Nopt              1000
  int   MaxGvtUpdateCycles INT_MAX
}
```

| Parameter Name | Description |
|---|---|
| Tgvt | This parameter specifies the time interval for GVT updates (if possible). For example, for a two node simulation with `Tgvt` set to `1.0`, one node may ask the other node for a GVT update once 1.0 seconds has elapsed since the last GVT update. However, the other node could be busy processing a computationally intensive event, preventing it from making its request for a GVT update until it has completed processing the current event. Hence, this node can exceed time specified by `Tgvt`. |

| Parameter Name | Description |
|---|---|
| Tasb | Time between asynchronous broadcasts of the event horizon. |
| Ngvt | The value of `Ngvt` represents the number of events that a node may process before requesting a GVT update. Once the current node has reached this limit (or `Tgvt`, whichever occurs first), it will request a GVT update. While waiting for the other nodes to make their requests, the first node will continue to process events optimistically . |
| Nrisk | This parameter specifies the maximum number of events a node may process with risk. |
| Nopt | The value of `Nopt` specifies the maximum number of events a node may process past GVT. Its value must be greater than that of `Nrisk`. |
| MaxGvtUpdateCycles | This parameter designates the maximum number of GVT update cycles that will be processed before assuming messages have been irrevocably lost. This number must be at least 2. The default value for this parameter is `INT_MAX`, the maximum value a signed integer can hold (This value is hardware specific and is typically found in the include file `limits.h`). |

Table C.2: The "gvt_parameters" section of the speedes.par file

## C.3 External Connections Configuration (section SpeedesServer)

The `SpeedesServer` section configures SPEEDES such that it can create external communication paths to external interfaces and allow each node of a multiple node SPEEDES application to run on different target machines. Usually, the default values for the parameters are used. The `SpeedesServer` contains two parts:

- The subsection `SpeedesComm` handles communication between nodes, which reside on different target machines.
- The subsection `HostRouter` handles communication between nodes and external interfaces.

Usually `SpeedesServer` handles both of the cases at the same time. However, users are allowed to break this functionality apart by setting the appropriate parameter listed below.

```
SpeedesServer {
  string  DefaultMachineName localhost
  int     DefaultPort        -1
  logical DefaultStatistics  F
  int     Group              0

  SpeedesComm {
    logical UseDefaults T
    string  MachineName localhost
    int     Port        -1
    logical Statistics  T
  }

  HostRouter {
    logical UseDefaults T
    string  MachineName localhost
    int     Port        -1
```

```
      logical Statistics  F
    }
  }
```

| Parameter Name | Description |
|---|---|
| DefaultMachineName | Specifies the target machine where the application `SpeedesServer` is executing. It can be specified by either host name or Internet Protocol (IP) address. |
| DefaultPort | The value of this parameter specifies the number of the port through which the application will contact the `SpeedesServer`. When the value of `DefaultPort` is set to -1, the chosen port will have a port number equal to 10000 plus the value of the user's (unique) UNIX user id (as displayed by the UNIX command `id -u`). |
| DefaultStatistics | When set to `T`, statistics will be generated for the `SpeedesServer` and `HostRouter`. |
| Group | In order to allow multiple simulations to run simultaneously with one `SpeedesServer`, users can specify unique group numbers for each simulation. |
| SpeedesComm | This section is used if the parameter `UseDefaults` below is set to `F`. |
| UseDefaults | If set to `T`, then the default `SpeedesServer` parameters listed above are used. If set to `F`, then the parameters contained within this section are used. |
| MachineName | The value of this parameter specifies the machine on which the `SpeedesServer` is running. It can be either an alphabetic host name or numeric IP address. |
| Port | The value of this parameter specifies the number of the port through which the application will contact the `SpeedesServer`. When the value of `DefaultPort` is set to -1, communication will take place through a port with a port number equal to 10000 plus the (unique) value of the user's UNIX user id (as displayed by the UNIX command `id -u`). |
| Statistics | The value of this parameter indicates whether or not statistics are to be generated for the `SpeedesServer` or `HostRouter`. |
| HostRouter | This section is used if the parameter `UseDefaults` below is set to `F`. |
| UseDefaults | If set to `T`, then the default `SpeedesServer` parameters listed above are used. If set to `F`, then the parameters contained within this section are used. |
| MachineName | The value of this parameter specifies the machine on which the `SpeedesServer` is running. It can be either an alphabetic host name or numeric IP address. |
| Port | The value of this parameter specifies the number of the port through which the application will contact the `SpeedesServer`. When the value of `DefaultPort` is set to -1, communication will take place through a port with a port number equal to 10000 plus the (unique) value of the user's UNIX user id (as displayed by the UNIX command `id -u`). |
| Statistics | The value of this parameter indicates whether or not statistics are to be generated for the `SpeedesServer` or `HostRouter`. |

Table C.3: The “SpeedesServer” section of the speedes.par file

## C.4 Output Data Configuration (section statistics)

The `statistics` section controls and selects the different types of data that can be output by the SPEEDES framework.

```
  statistics {
    logical CYCLE               T
    logical GVT                 T
    logical LVT                 T
    logical BTW                 T
    logical CPU                 T
    logical WALL                T
    logical STAR                T
    logical PROC                T
    logical COMMIT              T
    logical PROCEFF             T
    logical PHASE1              T
    logical PHASE2              T
    logical EVENTS              T
    logical EVTGVT              T
    logical EVENTSCYCLE         T
    logical ROLLBACKS           T
    logical MESSAGES            T
    logical ANTIMESSAGES        T
    logical CANCELS             T
    logical NUM_EVENTS_IN_QUEUE F
    int     ReportTime          1

    logical WriteGvtStatistics  F
    string  FileName            GvtStatistics

    string  Timer               WallClock
    logical MemoryUsage         F
    logical MessageSending      F
    logical EventProcessing     F
    logical ObjectProcessing    F
    logical AutoLazyEvaluation  F
    logical CriticalPath        F

    IntervalOutputTime {
      float SimTime             10.0
      OutFileName {
        string StatisticsFileName intervalStats
      }
    }

    RollbackBreakdownByEventType {
      float OutputIntervalSimtime 100.0
      int   WidthOfPrintedMatrix  80
      OutFileName {
        string RollbackBreakdownFileName rbTable
       }
    }

    StatsOnSimobjByEventBasis {
      string  MyStatData        /tmp/output
      logical PrintIntervalData T
    }

    string Output_Method        stderr
    string Stat_Output_Filename statq
    string Stat_Displayer       localhost
    int    Port                 1029

    ObjNamesByNode {
      string fileName Objectmap
    }
  }
```

| Parameter Name | Description |
|---|---|
| CYCLE | If set to T, display the GVT cycle number. |
| GVT | If set to T, display the value for GVT. |
| LVT | If set to T, add the value for LVT to the GVT statistics file. |
| BTW | If set to T, add the times for BTW phases to the GVT statistics file. |
| CPU | If set to T, display the total CPU time spent. |
| WALL | If set to T, display the total wall clock time spent. |
| STAR | If set to T, display the Simulation Time Advancement Rate (STAR) since the last statistics display. |
| PROC | If set to T, display the total time spent processing events. |
| COMMIT | If set to T, display the total time spent committing events. |
| PROCEFF | If set to T, display the ratio of committed events to processed events over the last cycle. This ratio may be greater than 1. |
| PHASE1 | If set to T, display the number of events processed optimistically in the last cycle. |
| PHASE2 | If set to T, display the number of events committed in the last cycle. |
| EVENTS | If set to T, display the total number of events committed since the start of the simulation. |
| EVTGVT | If set to T, display the number of events processed during the last GVT cycle. |
| EVENTSCYCLE | If set to T, display the number of events processed during the last cycle. |
| ROLLBACKS | If set to T, display the number of events rolled back since the start of the simulation. |
| MESSAGES | If set to T, display the number of messages sent between nodes since the start of the simulation. |
| ANTIMESSAGES | If set to T, display the number of anti-messages sent between nodes since the start of the simulation. |
| CANCELS | If set to T, display the number of events canceled since the start of the simulation. |
| NUM_EVENTS_IN_QUEUE | If set to T, add the number of events in the queue to the GVT statistics file. |
| ReportTime | The period, in wall clock time, at which the statistics are written to the screen and to the GVT statistics file. A value of 0 instructs SPEEDES to write statistics at every cycle. |
| WriteGvtStatistics | If set to T, write a GVT statistics file for every node. |
| FileName | The value of this parameter specifies the common prefix used in names of the GVT statistics output files. The full file name will consist of the specified prefix suffixed by the node number. |

| Parameter Name | Description |
|---|---|
| Timer | This parameter, whose value can be either `CPU`, `WallClock`, or `Counter`, determines the particular method used for timing events. Displayed time spent processing events is given either in CPU seconds or in seconds of wall clock time, depending on whether `Timer` is set to `CPU` or `WallClock`. Setting `Timer` to `Counter` simply increments by 1 each time the timer is called (thus providing an event count on an event type basis). |
| MemoryUsage | If set to `T`, memory usage for the event and message free lists will be printed to the terminal. |
| MessageSending | If set to `T` and the number of nodes is greater that 1, the statistics on message sending will be printed to the terminal. |
| EventProcessing | If set to `T`, statistics on event processing will be printed to the terminal. |
| ObjectProcessing | If set to `T`, statistics on event processing for objects by type will be printed to the terminal. |
| AutoLazyEvaluation | If set to `T`, and automatic lazy re-evaluation has been turn on, then statistics for automatic lazy re-evaluationperformance performance are printed to the terminal. Note: compile time options of `USE_AUTO_LAZY` and `AUTO_LAZY_PERF` must have been used during SPEEDES and application compilation. |
| CriticalPath | If set to `T`, the critical path and maximum speedup will be calculated and printed to the terminal. |
| IntervalOutputTime | When this section is present, then the `MemoryUsage`, `MessageSending`, `EventProcessing`, and `ObjectProcessing` statistics are printed at the interval specified by `IntervalOutputTime`. |
| SimTime | The value of `SimTime` specifies the approximate time intervals at which statistics data should be recorded. |
| OutFileName | Inclusion of this parameter forces printing of interval data to a file, whose name is given by `StatisticsFileName`, rather than the screen. |
| StatisticsFileName | The value of `StatisticsFileName` names the file to which interval statistics data should be written. |
| RollbackBreakdown-ByEventType | Inclusion of a `RollbackBreakdownByEventType` section will cause the printing of a matrix describing event rollbacks. The entry in row $i$ and column $j$ indicates the number of times an event of type $i$ rolled back an event of type $j$. |
| OutputIntervalSimtime | Optionally, print the rollback matrix at fixed approximate intervals specified by this value. |
| WidthOfPrintedMatrix | Since the rollback matrix may be too large to fit on the screen, the user may wish to break it into subtables. The value of this parameter defines the width, in characters, of the subtables. |
| OutFileName | Optionally, the user may wish to output the rollback matrix to files (one file per node), rather than the screen. Names for these output files may be chosen by setting the parameter `RollbackBreakdownFileName`. |
| RollbackBreakdown-FileName | The value of this parameter specifies the common prefix used in the names of the rollback matrix output files. The full file name will consist of the specified string suffixed by a pattern of the form `node#`, where # is the node number. |
| StatsOnSimobjBy-EventBasis | Optionally, the user may wish to output event usage statistics on an individual object basis. The `StatsOnSimobjByEventBasis` subsection allows such configuration. |
| MyStatData | This parameter indicates the directory to which individual object data will be written. |
| PrintIntervalData | Setting this logical parameter to `T` turns on interval output based on the settings found in `IntervalOutputTime`. |

| Parameter Name | Description |
|---|---|
| Output_Method | Setting this parameter, to either `stderr`, `File`, or `Socket`, offers the user control over the choice of data stream to which statistics are printed. The values `stderr` and `File` are self-explanatory (`File` is defined in parameter `Stat_Output_Filename`), while a value of `Socket` instructs SPEEDES to send the statistics data to a different machine, as defined by parameter `Stat_Displayer`. |
| Stat_Output_Filename | This parameter is required if the value of `Output_Method` is `File`. The value of `Stat_Output_Filename` names the file to which data should be written. |
| Stat_Displayer | This parameter is required if the value of `Output_Method` is `Socket`. The value of `Stat_Displayer` specifies the machine to which data should be transferred. |
| Port | This parameter is required if the value of `Output_Method` is `Socket`. The value of `Port` specifies the port on the machine indicated by the value of `Stat_Displayer` to which data should be transferred. |
| ObjNamesByNode | This section enables printing of the object name, handles, and kind ids in order to determine what node each object lies on. |
| fileName | This string specifies the prefix for the file name of output of the lists of objects and the nodes on which they lie. |

Table C.4: The "statistics" section of the speedes.par file

## C.5 Flying Trace Diagnostic Configuration (section FlyingTraceOutput)

If SPEEDES was compiled with flying trace enabled (see Section 17.3), then this section, if it exists, enables the flying trace output.

```
FlyingTraceOutput {
  string TraceFileName MyFlyingTraceFile
}
```

The parameter `TraceFileName` specifies the file name for the flying trace output data.

## C.6 Trace Diagnostic Configuration (section trace)

When enabled, this section specifies that trace files are to be created and saved under the specified file name.

```
trace {
  logical trace     F
  string  tracefile Trace
}
```

| Parameter Name | Description |
|---|---|
| trace | If set to `T`, trace files will be created. |
| tracefile | The value of this parameter specifies the common prefix used in names of the trace files. The full file name will consist of the specified prefix suffixed by the node number. |

| **Parameter Name** | **Description** |
|---|---|

Table C.5: The “trace” section of the speedes.par file

## C.7 Simulation Named Pauses (section NamedPauses)

Each item listed in this section corresponds to a pause that must be removed using the `SpResume` command line utility (see Section 13.6), or by using the `SpEmHostUser` (see Chapter 12). This section can contain as many user-defined pauses, at anytime, as required for the simulation. Then, names of the pauses, for eaxmple `Pause1` and `Pause2` as shown below, can contain any legal character as allowed for parser input parameters (i.e. alphanumeric and underscore characters).

```
NamedPauses {
  define Pause1 0.0
  define Pause2 50.0
}
```

## C.8 Checkpoint/Restart Configuration (section Checkpoint)

If checkpoint/restart is desired, inclusion of a `Checkpoint` section will enable the creation of checkpoint files.

```
Checkpoint {
  logical Enable           F
  float   WallTimeInterval 600.0
  float   SimTimeInterval  100.0
  string  CheckpointPath   ./Checkpoints
}
```

| Parameter Name | Description |
|---|---|
| Enable | Setting this parameter to `T` will enable the checkpoint capability. |
| WallTimeInterval | The value of this parameter defines the approximate wall clock time interval at which checkpoints should be created. If this value is less than or equal to zero, then this parameter is not used when determining whether or not it is time to perform a checkpoint. |
| SimTimeInterval | The value of this parameter defines the approximate simulation time interval at which checkpoints should be created. If this value is less than or equal to zero, then this parameter is not used when determining whether or not it is time to perform a checkpoint. |
| CheckpointPath | The value of `CheckpointPath` specifies the directory in which checkpoint files will be created. |

Table C.6: The “Checkpoint” section of the speedes.par file

## C.9 HLA (section HLA)

This optional section enables the use of the HLA Gateway.

```
HLA {
  logical Gateway T
}
```

# Appendix D

# Acronyms and Abbreviations

| | |
|---|---|
| AFB | Air Force Base |
| AltItem | Alterable Item |
| ANSI | American National Standards Institute |
| API | Application Program Interface |
| BTW | Breathing Time Warp |
| CPU | Central Processing Unit |
| DDM | Data Distribution Management |
| DM | Declaration Management |
| ECI | Earth Centered Inertial |
| ECR | Earth Centered Rotating |
| FIFO | First In First Out |
| GHz | gigahertz |
| GNU | GNU’s Not Unix |
| GPS | Global Positioning System |
| GVT | Global Virtual Time |
| HLA | High Level Architecture |
| id | identifier |
| IP | Internet Protocol |
| IMPORT | Integrated Modeling and Persistent Object Relations Technology |
| km | kilometers |
| LVT | Local Virtual Time |
| MODSIM | Modular Simulation |
| ms | milliseconds |
| PDES | Parallel Discrete-Event Simulation |
| SDES | Sequential Discrete-Event Simulation |
| SOW | Statement of Work |
| SPEEDES | Synchronous Parallel Environment for Emulation and Discrete-Event Simulation |
| STAR | Simulation Time Advancement Rate |
| SDRL | Subcontractor Data Requirement List |
| vtable | virtual function table |

# Index