SPADES
System for Parallel Agent Discrete Event Simulation

User's Guide and Reference Manual

For Version 0.91

`http://spades-sim.sourceforge.net`

Patrick Riley
`pfr+@cs.cmu.edu`

December 7, 2003

Contents

List of Figures

List of Tables

Acknowledgments

SPADES originally grew from a graduate class project in 15-712: Advanced Operating Systems and Distributed Systems at Carnegie Mellon University. I am grateful to my partner Emil Talpes who worked with me on the initial implementation. I would also like to thank the instructor Dr. Greg Ganger and the TA Jay Wylie for their comments and advice. Lastly, I would like to thank my adviser Dr. Manuela Veloso for permitting this hopefully small side project while I work on my own Ph.D. thesis.

Typographical Conventions

The following typographical conventions are used throughout the text.

- All program parameters appear in slanted type, e.g. *my_program_parameter*
- All method and class names appear in monospaced type, e.g. `myMethod`, `MyClass`
- All exact code or things to be typed appear in monospaced type (e.g. `type me`), and variables are then italicized (e.g. `My name is` *`yourname`*).
- Specialized SPADES terms are in sans serif (e.g. my new term). Note that these terms are formatted in regular font in the index.

Chapter 1

Introduction

1.1 What is SPADES?

The System for Parallel Agent Discrete Agent Simulation (SPADES) is a middleware system for agent-based distributed simulation. It is targeted mainly at the artificial intelligence community where the "thinking" of an agent is a significant amount of the computation involved in a simulation. SPADES is designed to ease the creation of agent-based simulations which can be distributed across machines while still being repeatable and efficient.

The agent is a central feature of SPADES. By "agent," we simply mean a computational entity that receives sensations from the simulated world, goes through some amount of computation, then returns some actions to be executed. SPADES explicitly tracks the latencies inherent in sensing, thinking, and acting and reflects those in the simulation.

SPADES provides an abstraction that allows the designers of world models and designers of agents to ignore networking issues and reasoning about distributed event distribution. The world model operates with a view of simulation events being realized one by one, and the agents simply receive sensations and send actions.

Several of the important features of SPADES are:

- Agent based execution, including explicit support for modeling latencies in sensation, thinking, and acting. The modeling of the thinking latency (the time to return a set of actions after receiving a sensation) is done by tracking the amount of computation done in response to a sensation being sent.

- Agents can be distributed among multiple machines. SPADES does all the necessary reasoning so that world model and agent designers do not need to do anything different based on how many computers are being used for the simulation.

- The result of the simulation is unaffected by network delays or load variations among the machines. Naturally, these factors can affect the speed of the simulation.

- The architecture for the agents is unconstrained. SPADES does not require that the agents are written in a particular programming language. The only requirement is that the agent processes can read and write to pipes.
- The agents' actions do not have to be synchronized in the domain. Unlike many simulation environments for the artificial intelligence community, the agents do not take a single joint action at a particular time. Rather, their actions can take effect at varying times in the simulation.

1.2 What SPADES Provides

SPADES is a simulation middleware that is not tied to the simulation of any one particular domain. There are two pieces that need to be added to SPADES to make a complete simulation: agents (computational entities) which sense, think, and act; and the world model which simulates the "ground truth" of the world and all agent interactions.

SPADES provides the following to aid the creation of simulations:

- An abstraction for an event based simulation is provided to the world model. A simulation consists of events being realized in time order (this is known as discrete event simulation). The world model can create simulation events and do arbitrary processing as the events are realized (i.e. have their effect on the world).
- An abstraction of sensations and actions is provided to the agent. The agent simply needs to read data from a pipe (in a format determined by the world model) and send actions back on another pipe. The agent does not need to do networking or explicit time management, except perhaps to limit the amount of computation done.
- SPADES tracks the amount of computation used by agents in responding to a sensation. This is done to simulate the fact that an agent which uses more computation will respond more slowly.
- All networking is managed. Neither the world model nor the agents need to create sockets, know network addresses, or do any other network management.
- All reasoning to manage distribution among machines is done. It is a tricky problem to distribute a simulation across machines in a way that maintains correctness (does not violate causality among events) and efficiently uses the resources available. The results of a SPADES based simulation are not affected by variations in number of machines used, network load, or machine load. The world model and agent designers can largely ignore all issues of distribution.
- Various logging facilities are provided. Logging for errors and traces of execution, as well as a record of the states of the simulations can all be managed by SPADES.

1.3 How to Use This Manual

This manual is the primary documentation of SPADES and is targeted at a variety of users. Here is the recommended way to read this manual for different uses:

User of a complete simulation If you have a complete SPADES based simulation that you want to use, you may still need some basic information about SPADES. Chapter 2 gives an overview and Chapter 3 talks about how to get started running the system. You will need to understand the agent database described in Sections 4.5 and 6.2. Lastly, you might be interested in the parameters described in Section 6.1.

Agent designer If you would like to create an agent to function in a SPADES simulation which already exists, you should also start with Chapter 2 to get an overview of the system, then read Chapter 3 for how to get started. You will probably then be most interested in the Sections 4.5, 4.6, 4.7 5.5, 6.2, 6.3, 6.4, and 6.7

World model designer If you want to create a new simulation using SPADES, you will need to create a new world model. You should first read Chapters 2 and 3 to get an overview of the system, and then Chapter 4 is your primary source of information. You should refer to the later chapters as needed, so a quick scan to see what is there is probably in order.

Simulation researcher A more concise summary of SPADES is provided in technical papers along with experimental results. Riley [2003] and Riley and Riley [2003] are probably better places to start. In this document, you might want to read Chapter 2, then Section 6.8.

Programmer interested in working on SPADES You should read the whole thing, probably in order. Then, you'll need to start reading lots of source code and the comments in the code. Then, you may still be confused and you'll have to discuss it on the developer mailing list:

`spades-sim-devel@lists.sourceforge.net`

Chapter 2

System Structure

This chapter describes the basic structure of all SPADES based simulations. This chapter is a good place to start for anyone who is working with SPADES.

2.1 Component Organization

Figure 2.1 gives an overview of the structure of the entire SPADES system, along with the components users of the system must supply (shaded in the diagram). The simulation engine and the communication server are supplied as part of SPADES. The world model and the agents are created by a user to simulate a particular environment.

The simulation engine is the heart of the discrete event simulator. All pending events are queued here, and the engine coordinates all network communication. A communication server must be run on each machine on which agents run. The communication server manages all communication with the agents (through a Unix pipe interface) as well as tracking the CPU usage of the agents to calculate the thinking latency (see Section 6.7.1). The communication server and simulation

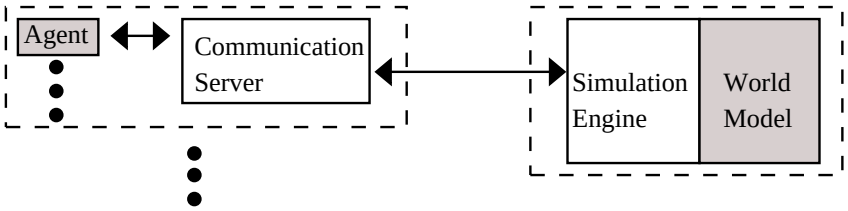

Figure 2.1: Overview of the architecture of the SPADES system. The shaded components are provided by the users of the system, not by SPADES itself. The dotted lines denote machine boundaries.

engine communicate over a TCP/IP connection.[1] Note that if you want a communication server and the simulation engine to run on the same machine, you can run an integrated communucation server which runs in the same process as the simulation engine.

The world model is created by a user of SPADES to create a simulation model of a particular environment. The simulation engine is a library to which the world model must link, so the simulation engine and world model exist in the same process. The world model must provide such functionality as advancing the state of the world to a particular time and realizing an event (changing the state of the world in response to an event occurring). SPADES provides a collection of C++ classes from which objects in the world model can inherit in order to interact with the simulation engine.

Lastly, the agents are the computational entities whose interactions are being simulated. Section 2.3 describes their interaction with the system in more detail, but the agents communicate with the communication server via pipes, therefore the agents are free to use any programming language and any architecture as long as they can read and write to pipes.

2.2 Event-Based Simulation

SPADES is a hybrid of a continuous and discrete event simulator. A continuous simulation is one in which a small time quanta is chosen and the simulation advances by processing for each time quanta in turn. A discrete event simulation is one in which the state of the simulation is changed by a series of events with each taking place at a discrete time.

In the interaction with the world model, SPADES is a continuous simulator. The simulation engine calls to the world model to advance some number of time quanta. This is especially useful for simulating an underlying continuous process such as physical interactions.

In the interaction with the agents (and all reasoning about distribution), SPADES is a discrete event simulator. The assumption is that agents are doing nothing until they receive sensations from the world (though note that agents can request a sensation at a particular time with a request time notify message). Events (e.g. sensations and actions) form the basis for an agent to be affected by and to affect the simulation.

The world model has several primary jobs in the simulation. The first is to advance the state of the simulation up to the time of the next event as requested by the simulation engine. The second is to "realize" an event by putting the effects of that event into the state of the world. Lastly, the world model needs to generate sensations to be sent to the agents and create events based on the responses of the agents.

SPADES guarantees that the order of event realization will not violate causality, i.e. no events which are causally related will be realized out of order. This means that events may not be realized

[1] Since the simulation needs the lowest latency traffic possible in order to achieve efficient simulation, Nagle's algorithm is turned off on the TCP sockets (using the `TCP_NODELAY` socket option in the Linux socket interface). Manual bundling of messages is done to avoid sending an excessive number of small packets. See the parameters *internal_tcp_packet_size* and *agent_packet_size*.

in strict time order. Section 4.2 describes the exact guarantees that SPADES provides as well as further details about events.

2.3 Sense-Think-Act

SPADES views an agent as a computational entity which receives sensation messages, does some thinking (i.e. processing), and returns some actions. The agent is assumed to do no processing outside of the receipt of a sensation (though note that agents can request their own sensations). Therefore, from the agent's perspective, the interaction with the simulation is fairly simple:

1. Wait for a sensation to be received
2. Decide on a set of actions and send them to the communication server
3. Send a done thinking message to indicate that all actions were sent

Since SPADES restricts an agent to only send actions in response to a sensation, the system gives an agent a special action called request time notify. A time notify is essentially an empty sensation to which the agent can respond with actions as with any other sensation. For example, this allows an agent to send an action at a particular time, even if no sensation would normally be received then.

Another important concept for the sense–think–act cycle is the latencies in each step of that cycle. Figure 2.2 represents a timeline for executions within a cycle. Consider the topmost timeline. Time point A represents the point at which a sensation is generated, such as the frame of video from a robot's camera or a snapshot of stock market prices. The time between points A and B represents the time to transfer and process a sensation to be used by the thinking component starting at point B. For example, this could represent the time for a robot to capture a frame from its camera and extract information from it. Between points B and C, some computation is done to determine which actions must be taken in response to the sensation. Between points C and D, the action messages are sent to the effectors and those actions begin to take effect at point D. For examples, this latency could represent the time to transfer information to effectors. Note that we are not claiming that there is a fundamental difference between the computation that happens in the sense and act components of the cycle and the think component. However, a simulation system must necessarily create an abstraction over whatever world is being modeled. Since the simulation provides information and receives actions in a processed form, the use of an explicit latency allows the simulation to account for the time that that would be needed in the real world to convert to and from that processed form. Note that SPADES does *not* require that all sensations and actions have the same latency.

In many agents, the sense, think, and act components can be overlapped in time as depicted in Figure 2.2. SPADES explicitly allows all overlaps with one major exception. In our environment, the think cycles for a single agent may never overlap. This is a reasonable model of real–world agents, given that these agents are typically implemented using a single processing unit per agent.

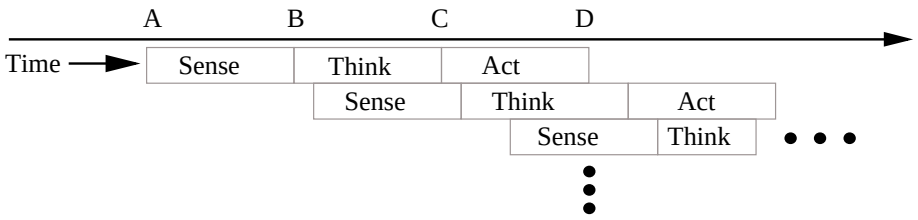

Figure 2.2: Example timeline for the sense-think-act loop of an agent

It would not make sense to allow our model of agent processing to think about several sensations at the same time, when the actual implementation of such an agent cannot do so.

One of the communication server's primary jobs is to track the thinking time of the agent to model the thinking latency. When sending a sensation to an agent, the communication server begins tracking the CPU time used by the agent, which is provided by the Linux kernel. When the done thinking message is received, the communication server calculates the total amount of CPU time used to produce these actions and the CPU time is translated into simulation time (see Section 6.7.1). All actions are given the same time stamp of the end of the think phase.

Also, if the jiffies timer is used, the current time slice reported by the Linux kernel is in 10ms increments known as jiffies. With the randomness in interrupts and other system activity, CPU usage numbers are unfortunately not perfectly repeatable, in contrast to a more language specific time tracking system like Anderson [1995, 1997]. However, our experiments indicate this effect is small [Riley and Riley, 2003].

The use of the perfctr timer can alleviate this problem by tracking the number of instructions processed. However, use of the perfctr timer currently requires a patched kernel.

Chapter 3

Getting Started

This chapter is a primer for starting with SPADES. It discusses the configuration and installation steps as well as a brief overview of the sample world model and agents provided.

3.1 Configuration and Installation

SPADES uses the GNU autotools.[1] Therefore, installation is (maybe) as easy as doing:

```
./configure
make
make install
```

This section covers some other features of configuration and installation.

3.1.1 Configuration Parameters

SPADES attempts to auto-detect a number of features. If these fail, you may need to provide some help to `configure`. You can also adjust what is built and installation directories and such. This section will cover the SPADES specific options in configure. For a discussion of the general autotools options, please see `http://www.gnu.org/software/autoconf/manual/autoconf-2.57/autoconf.html`. Note that all options that begin with `enable` also have a `disable` variant, and the `with` options also have a `without` variant.

enable-debug (default is off) If this is on, then action logging (see Section 5.1) is enabled (note that error logging is always enabled). It is recommended that you use this anytime you are developing a world model. However, once the world model is done, disabling this will slightly reduce the SPADES code size and increase execution speed.

[1] `http://www.gnu.org/software/autoconf/`, `http://www.gnu.org/software/automake/`, `http://www.gnu.org/software/libtool/`

enable-sample-code (default is on) If this is on, then the sample world model and agents are built. This is mostly useful to test the system. Definitely recommended for beginning users, but this is not necessary for SPADES to work with your own world model and agents.

enable-sample-code-monitor (default is on) Determines whether the monitor for the sample world model is built. The monitor is written in Java (everything else is C++), so there are some installations for which this can not be built. Disabling this still allows you to run SPADES and the sample world model and agents. However, it will be more difficult to see what is going on in the sample world model.

with-perfctr=DIR The perfctr timer requires the perfctr system which is a patch to the Linux kernel and a user library to access the information. If the user library is not in your standard include path, you can use this option to indicate where it is. If the headers for the patched kernel are in an unusual place, you will need to use the `CPPFLAGS` variable to help SPADES find them.

If this variable is not specified, then `configure` will try to detect whether perfctr is present and enable/disable the timer as a result. If you do give this option, `configure` will exit if perfctr is not found.

SPADES can still function correctly without the perfctr timer; there are other timers that can be used (see Section 4.7.1). However, it can affect reproducibility (see Section 4.11).

with-fork-dynlib-loc=PATHTOLIB In order to do the interception of dynamic library calls to `fork` and related functions (see Section 4.7.2), SPADES needs to locate the dynamic library that defines the `fork` function. It attempts to do this by compiling a small program and running `ldd` and `nm`. If you do not have these programs (or their output is in a format that `configure` does not understand), you may have to tell SPADES exactly which dynamic library defines `fork`. Note that SPADES assumes that `clone` is defined in the same library.

with-jiffies (default is yes) Determines whether to build the jiffies timer. This timer uses the `/proc` file system. If for some reason this is not available or SPADES can not use it, you can disable it (if this happens, please post of bug report/feature request on the SPADES web site). SPADES can function without this timer, but if you do not build either the perfctr timer or the jiffies timer, you will not be able to track the actual computation used.

with-java-prefix=PFX This is only relevant if you are building the sample world monitor. Gives a prefix where Java runtime is installed.

with-javac-flags=FLAGS This is only relevant if you are building the sample world monitor. Gives additional flags to give to the java compiler.

with-java-flags=FLAGS This is only relevant if you are building the sample world monitor. Gives additional flags to give to the java virtual machine.

3.1.2 Files Installed

This section gives an overview of what SPADES installs, but it does *not* give a complete file listing. All paths are relative to the prefix given during installation (`/usr/local` by default).

`bin/commserver` This is the communication server executable. Since the communication server is not specialized to a particular simulation, the same executable can be used with any world model.

`bin/show_tttimes.pl` This perl script processes a recorded agent think time file into a more human readable format (see Section 4.7.1).

`bin/run_exp.pl` This perl script provides a way to run SPADES many times for testing. It is perhaps of limited use without modifying the script. No further documentation for this script is provided in this manual.

`include/spades/` All header files which are needed to compile your world model or agents are installed here.

`lib/` The library to which the world model must link and the optional agent library for agent designers are put here. For information about the agent interface library, see the agent library manual on the SourceForge site.

`lib/spades/` Libraries which are used internally by spades are installed here. You may need to know this to set the correct value of *agent_intercept_library*, though in general, this should be done for you.

`share/spades/agent.conf` A configuration file which gives some reasonable default values for the parameters controlling how SPADES manages agents. This is useful to be included by the communication server and the simulation engine (if it is running an integrated communication server). You will likely need to copy and tweak for your simulation.

`share/spades/commserver.conf` A configuration file giving reasonable default values for the communication server, though you may need to copy and tweak for your simulation. Note that this file includes `agent.conf`.

`share/spades/agentdb.xsd` The XML schema which describes the format for the agent database.

3.2 Sample World Model and Agents

If this is your first time installing SPADES or if you want to check that everything is working the way it is supposed to, it is a good idea to try running the sample world model and agents.

3.2.1 Description

The sample world model is known as "Ball World." The simulated world in the sample world model is a two dimensional rectangle where opposite sides are connected (i.e. "wrap-around"). Each agent is a "ball" in this world. Each sensation each agent receives contains the positions of all agents in the simulation, and the only action of each agent is to request a particular velocity vector. The world model provides the correct accelerations to achieve this velocity. The dynamics and movement properties are reasonable if not exactly correct for small omni-directional robots moving on carpet, except that collisions are not modeled. Note that the parameters used approximate those of the CM Dragons from Carnegie Mellon which competed at RoboCup2001 [Birk et al., 2002]. The world model advances in 1ms increments.

There are two kinds of agents. The "wanderer" moves randomly around the world. The "chaser" chases one of the wanderers by setting its requested velocity directly towards the current observed location of that agent. Note that the agents do not try to predict ahead the latency between the time the sensation was generated at the action will take effect. If you watch the performance of the agents, you will notice this effect.

The agents can also be told to take differing amounts of computation time.

3.2.2 Running

The sample world model is not installed, so you will need to run it from the compilation directory. First, change to the `sample_world_model` directory. Then, to run the sample world model, do this:

```
./ballworld --file ballworld.conf
```

After printing the copyright notice and other startup information, you should see lines like this:

```
Engine Status: PAUSED simtime=0 events=0       mean simtime/sec=0
```

You should not receive any errors or warnings. If you do, you probably need to figure out what went wrong.

At this point, the world model and simulation engine are waiting for communication servers to connect. To connect a communication server from the local machine, just do this (all on one line):

```
commserver --file /usr/local/share/spades/commserver.conf
--agent_db_fn ../sample_agent/agentdb.xml
```

Note that this requires that you are still in the directory with the sample world model and that the installation bin directory is in your path.

If you want to connect several computers to the simulation, then you need to do the following. Let's say there are *N* computers that you would like to connect. Run the sample world model like this:

```
./ballworld --file ballworld.conf --num_comm_servers_wanted N
```

If the machine running the simulation engine is named *HOST*, then each communication server should be started like this (all on one line):

```
commserver --file /usr/local/share/spades/commserver.conf
--agent_db_fn ../sample_agent/agentdb.xml --engine_host HOST
```

If you want to run just on one machine, you can run with the integrated communication server like this:

```
./ballworld --file ballworld.conf --run_integrated_commserver
```

In any of these cases, you can connect a monitor to the simulation to watch it run. From the build directory:

```
cd sample_world_monitor/monitor
./connect
```

This will connect a monitor to a sample world model running on the localhost. If you need to connect to a remote machine, look at the way the `connect` script works and call java appropriately.

If you have a fast machine, the simulation may be over before you can connect a monitor. If this happens, try increasing *simulation_length* or connecting the monitor before the communication server.

There are a number of parameters that affect the way the sample world model runs. These are presented in detail in Section 6.1.4.

3.2.3 Log files

By default, SPADES produces a number of log files. These all appear, by default, in the `Logfiles` directory in the current working directory.

`agent*-stdout.log, agent*-stderr.log` The standard out and standard error of the agents. See Section 4.6.1

`agent*-ttimes.log` This file is a record of the agent think times. See Section 4.7.1.

`actions.log` This is the action log of the simulation engine. See Section 5.1.

`events_text.log` This is a human readable list of all the events realized during the simulation. Currently, SPADES can not read this file in, so it is output only. The parameter *use_text_event_log* controls whether this file is created and *text_event_log_fn* controls the location and file name.

`monitor.log` While this file is normally created by SPADES, its format is completely determined by the world model (see Section 4.8).

For the sample world model, you can graphically examine the contents of thie file with the sample world model monitor. From the build directory:

```
cd sample_world_model/monitor
./playlog
```

This plays the file `../Logfiles/monitor.log`. If you need to play some other file, look at the `playlog` script and modify appropriately.

Chapter 4

Creating a SPADES Simulation

This chapter provides details on what you must do to create a new simulation using SPADES. Several features which are not strictly required are discussed later in Chapter 5 and many technical details are given in Chapter 6, which is more of a reference than this chapter.

Further, there is a sample world model and sample agent distributed with SPADES in the `sample_world_model` and `sample_agent` directories. These should provide valuable examples from which to build.

4.1 Basic Simulation Process

This section will describe the overall process of simulation both from the perspective of the end user of the simulation and the perspective of the world model. Further details about most of this process are contained in the following sections of this chapter.

4.1.1 Running a Simulation

SPADES is designed to run a simulation across multiple machines. One machine will be the center of coordination and communication and will run the simulation engine and world model as a single process. The designer of the world model must link to the simulation engine library to produce one executable. This executable must be started first.

Then, communication servers must be started on every machine which will participate in the simulation. The communication server is given the host machine on which the simulation engine is running through the *engine_host* parameter. Note that for a communication server running on the same machine as the simulation, the integrated communication server can be used (see Section 5.4).

The communication server is responsible for starting up and monitoring agents. Each communication server must be given an agent database in order to know how to start agent processes. When the simulation is finished (as controlled by the world model), each communication server monitors the shutdown of the agents, then exits.

4.1.2 World Model's Perspective

First, a couple concepts must be described. SPADES is a combination of an event based and continuous simulation. The simulation proceeds by first advancing the world model to the time of the next event, then realizing an event (i.e. make the effects of the event in the world model).

Time is a discrete dimension for SPADES as represented by the `SimTime` type. The meaning of the simulation time step is left up to the world model.

The user of the spades simulation is responsible for creating the `main` function. Only two things are required by SPADES from the `main` function. First, an object of type `WorldModel` must be allocated. This object will be the primary interface between SPADES and the user of SPADES. Second, the function `SimulationEngineMain` must be called. Note that argument processing should not be done before calling `SimulationEngineMain`.

Once in `SimulationEngineMain`, the method `parseParameters` of the `WorldModel` object is called. Note that this function must return an object of type `EngineParam`. The recommended way to do argument processing is by subclassing `EngineParam` and calling the `getOptions` method.

Once the parameters are parsed, the logging facilities are initialized (see Section 5.1). Note that this means anything sent to the action logging facilities before this time will not be recorded. Messages sent to the error log will still appear on standard error, but not in the action log.

The simulation then enters the main simulation loop. The engine can be in various simulation modes, which are:

`SM_RunNormal` This is the mode in which the simulation spends most of its time. From the world model's perspective, the method `simToTime` will be called to advance the time of the world model to a given time. Then, an event will be realized. The simulation does some work to accomplish this, but the world model has an opportunity to realize the event through the `realizeEventWorldModel` method of `Event`. Note that there is no call to any `WorldModel` methods for event realization. It is expected that events which are enqueued will be subclasses of the standard types given by SPADES.

Note that the events may not be realized in strict time order, though causality is guaranteed not to be violated. Section 4.2 describes the details.

`SM_RunLimitedRate` This is like the normal run mode, except that the speed of the simulation is limited. Section 5.6 describes this in detail.

`SM_PausedInitial` There are three paused modes. In all of them, simulation time is not advanced, nor are any events realized. Messages are received from communication servers and then the `pauseModeCallback` method of the `WorldModel` is called.

If the simulation engine remains in paused mode long enough for *max_pause_mode_seconds* to pass, then the simulation is shut down.

This paused mode is the initial mode that the simulation starts in. You should never return to this mode once the simulation is started.

SM_PausedMonitor Like SM_PausedInitial except that this code is used when a monitor requests that the simulation is paused.

SM_PausedWorldModel Like SM_PausedInitial except that this code is used when the world model requests that the simulation is paused.

SM_Shutdown Indicates that the simulation is in the process of shutting down.

Note that the world model can control the change of simulation mode with calls to the changeSimulationMode method of the SimEngine class.

4.2 Events

This section further defines events as the basic building blocks for the simulation and then describes the event classes provided by SPADES upon which a world designer should build.

4.2.1 Definition

Events are one of the primary objects of the simulation. As described in Section 4.1, the simulation proceeds by advancing the world model to a particular time, then realizing an event (i.e. have an event take its effect on the world).

SPADES provides a class structure from which you should subclass in order to create events for your simulation. This section will describe these classes and how their methods will be used by the simulation.

There is one special subclass of events, namely fixed agent events. Knowledge of fixed agent events is used to achieve more parallelism among agents. Fixed agent events have the following properties:

1. They do not depend on the current state of the world.
2. They affect only a single agent, possibly by sending a message to the agent.
3. Sense events and time notify events are both fixed agent events.
4. Fixed agent events are the only events which can cause the agent to start a thinking cycle, but they do not *necessarily* start a thinking cycle.

With the separation of fixed agent events from other events, the correctness guarantees that SPADES provides can now be stated:

1. All events which are not fixed agent events are realized in time order.
2. All events which send sensations to the agents are fixed agent events.
3. The set of fixed agent events for a particular agent are realized in time order.

Section 6.8 provides the technical details about the algorithms to achieve this.

4.2.2 Interface Description

Figure 4.1 shows the events provided by SPADES, and their inheritance relationships. All events created by the user of SPADES should subclass one of these basic event types.

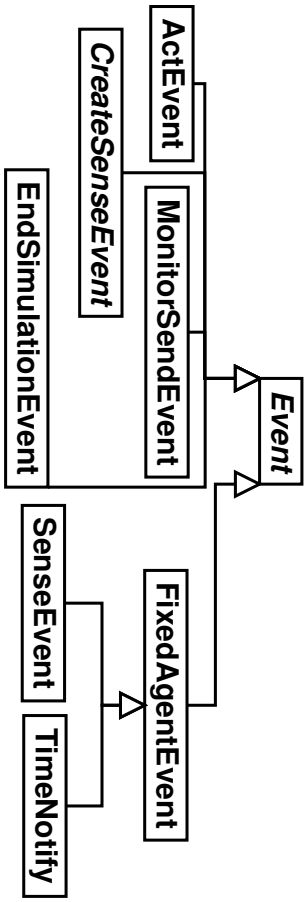

Figure 4.1: Event class hierarchy provided by SPADES. When two classes are connected, the lower one is a subclass of the upper.

The methods of these events and how those methods are used will now be described. Some details will be omitted in this discussion, so please see the source code for full information.

`Event` This is the abstract base type from which all events must inherit. `Event` maintains the time of event, some ordering properties, and provides virtual functions for subclasses to define.

The important methods are:

- The constructor takes two optional arguments, the time of the event and the event order constant. The time defaults to `SIMTIME_INVALID`, but an event with this time should never be enqueued in the simulation engine. The order constant is described under the method `getOrder`.
- `getTime` returns the time of the event.
- `setTime` sets the time of this event. Note that changing the time of an event in the queue of pending events can cause errors, and should never be done.
- `getOrder` gets the order constant of the event. All events have an ordering constant to order events with the same time. In general, all events of different types should have different ordering constants. The file `shared/sharedtypes.hpp` defines some constants for the various types.
- `getSecondaryOrder` This is a pure virtual function. If two events have the same time, and the same primary ordering constants (see `getOrder`), the return from `getSecondaryOrder` is used to order the events. If the same value is returned here, then SPADES can make no guarantees about the ordering of these events.
- `print` This is a pure virtual function. The text printing of events is used in various logging and error reporting facilities. The base class defines the `<<` operator on ostreams to call this.
- `realizeEvent` This is the function which is called to realize the event. While it is a virtual function, you should not override this method.[1] See the method `realizeEventWorldModel`. Returns whether the event can be deleted or was stored elsewhere.
- `realizeEventSimEngine` This protected pure virtual function does the work for this event to be realized by the simulation engine. It is called by `realizeEvent` before `realizeEventWorldModel` is called. You should not override this method unless you are changing SPADES as opposed to creating a simulation using SPADES.
- `realizeEventWorldModel` This protected pure virtual function should perform the work necessary for the world model to realize the event. This is the where all the work for an event to have its effect on the world model should be done. This function returns whether anything was done with this event. This return is mainly used for error checking to make sure that events are not being put in the queue which never do anything.

`ActEvent` An event of this type is queued when an act message is received from an agent. The `parseAct` method of the `WorldModel` object is responsible for creating these events (see Section 4.3). While this is not an abstract class, you will probably never want to use `ActEvent`, but rather subclass is so that you can override the `realizeEventWorldModel` method.

[1] The only reason that the method is virtual is so that `FixedAgentEvent` can override it. No other class should do so.

The only thing that `ActEvent` adds to `Event` is the storing of an agent which is associated with this event. This value is accessed via the `getAgent` and `setAgent` methods.

`CreateSenseEvent` This is an abstract base class useful for creating sensation events (see Section 4.6 for a description of the basic sense–think–act cycle and the associated events). While not all sensation events must be created through the realization of a `CreateSenseEvent`, it is strongly recommended that most are. Primarily, this allows the simulation engine to do additional error checking and eases the creation of sensations for the world model designer.

`CreateSenseEvent` adds two things to the basic `Event` class. The first is storage of an agent id, accessed via the `getAgent` and `setAgent` methods. Secondly, the pure virtual function `createSense` is called by `realizeEventSimEngine`. `createSense` should return an event of type `SenseEvent` which is the created sensation. That sensation will then be enqueued by the simulation engine.

Note also that `CreateSenseEvent` defines `getSecondaryOrder` to return the agent id, and you likely do not need to override this.

`EndSimulationEvent` The realization of this event will begin the shutdown process of the simulation. The recommended way to end the simulation is by enqueuing an `EndSimulationEvent`.

`MonitorSendEvent` This event is used internally by the simulation engine to create the monitor log file. There is probably no reason for the world model to create events of this type.

`FixedAgentEvent` Fixed agent events are the primary subtype of events. Section 4.2.1 describes exactly what fixed agent events mean and how they affect the ordering guarantees of SPADES. As a world model designer, you should be careful which defining subclasses of `FixedAgentEvent` because of their different ordering treatment. There are two subclasses of `FixedAgentEvent` which SPADES provides.

`SenseEvent` All sensations to be sent to the agent must arise from an event of this type. The realization of this event by the simulation engine causes a message to be sent to the agent.

A `SenseEvent` contains a value of type `ThinkingType` specifying the thinking disposition of this sensation. The possible values are:

`TT_Invalid` This should never be used here, it exists only to indicate an error.

`TT_Regular` A regular sensation with starts a thinking cycle. The timer module specified in the agent type is used to compute the thinking latency.

`TT_Untimed` This is like `TT_Regular`, except that the timing module output is ignored and the thinking latency is set to 0. This is known as an untimed sensation.

`TT_Not` This is a non-thinking sensation, also called an inform. An agent can not reply to a non-thinking sensation with actions, and any computation time used in processing an inform is added to the next thinking sensation.

The thinking status of a `SenseEvent` is controlled by the `getThinking` and `setThinking` methods.

`SenseEvent` also adds an additional time value to the event, the send time. Each sensation has a time at which it was generated and a time at which the agent can begin thinking about it; this interval is known as the sense latency. The time stored in the base `Event` class is the arrive time, and then `send_time` is a protected element of `SenseEvent`, accessed with the `getSendTime` method.

Lastly, `SenseEvent` maintains the data to be sent to the agent. This is stored in the protected member `data` which is of type `DataArray`, which is described fully in Section 4.9. The data to be sent can be arbitrary bytes.

`TimeNotifyEvent` This event is enqueued by the simulation engine for every time that the agent has requested a time notify with the request time notify message.

It is not recommended that the world model enqueue any of these events, as it could confuse the agents. Use sensations and `SenseEvent` instead.

4.3 World Model

The `WorldModel` class provides the primary interface for the simulation engine to interact with the user created part of the simulation. One of the primary tasks to be done to create a SPADES based simulation is to create an appropriate subclass of `WorldModel`. This section describes in detail what the functions should do. All methods here are pure virtual.

Constructor Most of the real initialization should be done in the `initialize` method (see below). Just setting values to essentially NULL is probably appropriate.

Destructor The `finalize` method will always be called before the destructor. SPADES will in fact never delete the `WorldModel` and the `SimEngine` object will already be deleted by the time the object is destroyed. Therefore, it is recommended that most cleanup be done in the `finalize` method.

`parseParameters` This should parse the parameters given in the command line. An object of type `EngineParam` must be returned. It is recommended that you inherit a parameter structure from `EngineParam` and use that to parse the options. Otherwise, you will have to do your own parsing and set the parameters in `EngineParam` yourself. See Section 5.2 for a description of how the parameter interface works. Note that this method is called *before* `initialize`.

`initialize` This is where the initialization should be done. It will be called after the `SimEngine` is initialized and the parameters have been parsed with `parseParameters`. Note that a pointer to the `SimEngine` class is passed in and you will probably want to store this value to call back to the simulation engine.

`finalize` This method will always be called before the destructor and before the `SimEngine` object is destroyed. After this method is called, any reference to the `SimEngine` object passed to `initialize` method is invalid.

`simToTime` This function asks the world model to advance the state of the simulation to the given time. Note that the world model can enqueue simulation events or perform other callbacks to the simulation engine while advancing the world time. Further, the world model is not required to advance the time all the way to the time requested. In particular, if events are inserted into the queue for a time before the requested time, you have to be careful *not* to advance the time past the earliest inserted event.

`getMonitorHeaderInfo` This method is called once for every monitor. It should return any header/setup information that is needed. Section 4.8 describes the monitor setup.

`getMonitorInfo` This method will be called periodically to get information about the current state of the world to send to the monitor. Section 4.8 describes the monitor setup.

`parseMonitorMessage` A monitor can send information in an arbitrary format to be decoded by the world model. This function is called when such information is received. It is recommended that you create events to actually perform the actions the monitor requests, rather than making the changes directly. This allows a better log and possibly reproduction of the simulation. Section 6.6 describes the monitor interface in detail.

`getMinActionLatency` This method should return the minimum action latency for any action by any agent. Section 4.10 discusses issues of how this latency affects the parallelism that is achieved. The return of this function should be a constant that does not change during the simulation. In future versions, this function may be only called once.

`getMinSenseLatency` This method should return the minimum sensation latency for any action by any agent. Section 4.10 discusses issues of how this latency affects the parallelism that is achieved. The return of this function should be a constant that does not change during the simulation. In future versions, this function may be only called once.

`parseAct` This method should parse the data given to it as an action request from the agent and return an event of type `ActEvent`. Note that this is a `const` method. Parsing of an action should not affect the world model in any way, or causality could be violated.

`pauseModeCallback` This method is called periodically while the simulation is in paused mode. Waiting for communication servers and starting up agents are typically done here. Also, at some point `changeSimulationMode` should be called to move the simulation to a run mode.

`agentConnect` This method is called for every agent once the agent has started up successfully. See Section 4.5 for a discussion of agent types.

agentDisappear This method is called after each agent exits. An argument which is of type `AgentLostReason` gives the reason for the agent leaving the simulation. The valid values are:

- **ALR_None** No reason; this value should never be passed to this method.
- **ALR_ProcessVanished** The agent process disappeared without sending an exit command.
- **ALR_BadFD** One of the file descriptors for communication with the agent was bad. This is usually caused by the agent process exiting.
- **ALR_InitError** An error was encountered in initialization, such as being unable to find or execute the program specified by the agent type.
- **ALR_WorldModel** The world model requested that the agent be killed.
- **ALR_CommServerDisconnect** The communication server which this agent was on disconnected.
- **ALR_AgentRequest** The agent sent an exit message.
- **ALR_Internal** Some sort of internal error occurrsed inside SPADES. This should not ever happen. The only current reason (as of 0.91) would be a failure of the timing mechanim.
- **ALR_ThinkTooLongWallClock** If an agent takes more than *max_secs_for_agent_think* seconds (measured by the wall clock time) to send a done thinking message in response to a sensation, the agent is shut down with this status.
- **ALR_ThinkTooLongSim** If an agent takes more than *max_simtime_for_agent_think* as measured by the current process timer (see Section 4.7.1), the agent is shut down with this status.

notifyCommserverConnect This method is called every time a communication server connects. In most cases, the world model does not need to know how when or how many communication servers connect, so this method can do nothing. However, there are situations where you may want to know. For example, you may want to start one agent on each communication server that connects.

notifyCommserverDisconnect This method is called for each communication server that disconnects. See the notes for `notifyCommserverConnect`.

4.4 Simulation Engine Interface

This section describes the methods of the simulation engine class in which the world model and associated events will be most interested. The class `SimEngine` is a bit of a monolithic object, so this section only describes the methods most relevant to the world model.

getSimulationTime Returns the current simulation time to which the engine has advanced. Note that this can disagree with the value perceived by the world model if it is in the middle of advancing the time in a `simToTime` call.

getNumAgents Returns the number of agents which have been started with `startNewAgent` calls. This is not necessarily equal to the number agents for which `agentConnect` has been called.

getNumCommServers Return the number of communication servers which have been connected.

startNewAgent Starts a new agent of the given type (see Section 4.5 for a discussion of agent types). Note that you can optionally specify which communication server to start the agent on. Otherwise, SPADES tries to distribute agents across all communication servers.

areAllAgentsInitialized Returns whether all agents are initialized. When an agent is initialized, the `agentConnect` is called, so the world model likely will not need this function.

killAgent This method will remove the agent from the simulation (notifying the agent first). It is possible that queued messages and events will still be received, so the world model needs to be able to ignore these.

sendExtraMonitorInfo This method sends extra data to all monitors. In general, the world model will not need to call this method, relying on the simulation engine to call the `getMonitorHeaderIn` and `getMonitorInfo` methods of `WorldModel`.

enqueueEvent This is one of the primary functions which the world model will use. It adds an event into the queue of pending events.

inPauseMode Returns whether the simulation is in any paused mode.

inShutdownMode Returns whether the simulation is in a shutdown mode.[2]

getSimulationMode Gets the current mode of simulation (see Section 4.1.2).

changeSimulationMode Changes the current mode of simulation (see Section 4.1.2).

initiateShutdown This begins the shutdown of the simulation. This is not the recommended way to initiate a shutdown except in exceptional circumstances. Rather, you should enqueue an event of type `EndSimulationEvent`. This is currently equivalent to `changeSimulationMode (SM_Shutdown)`

[2]Currently, the only shutdown mode is SM_Shutdown. However, in the future, other shutdown modes may be used to indicate why the shutdown is occurring.

`getWorldModel` Returns a pointer to the current world model.

`getAgentTypeDB` Returns a pointer to the agent database.

`getCurrWallClockTime` Returns a `timeval` structure for the current wall clock time (see '`man gettimeofday`').

4.5 Agent Types

All agents in the simulation have an associated type. This type easily allows heterogeneous agents to be mixed in the simulation. The agent database stores all information about the agent types. All agent types have an associated name to identify the agent. All names must have no white space and only include the alphanumeric characters plus - and _.

The details of the file format of the agent database are described in Section 6.2.

4.5.1 External Agents

An external agent is one that is started as a separate process and communicated with via pipes.

For an external agent, the agent database allows one to specify:

- The file descriptors the agent will use for input and output.
- The process timing module to use (see Section 4.7.1).
- The working directory for the agent.
- The executable to use, with arguments.

4.5.2 Integrated Agents

An integrated agent is loaded from a dynamic library and exists in the same process as the communication server. This allows the agent to corrupt the communication server and disallows the use of the some of the timing mechanisms. However, it has the benefit of being faster than running an external agent. Also, if running in an integrated communication server, the agent could access internal data of the world model, alleviating the need to serialize and deserialize the information.

For an integrated agent, the agent database allows one to specify:

- The path to the dynamic library which contains the agent.
- The process timing module to use (see Section 4.7.1).
- Whether the agent must execute under an integrated communication server or not.
- The arguments to give the agent when it starts up.

4.5.3 Placeholder Agents

A placeholder agent is an agent type with only a name. A placeholder agent can not be started by a communication server. It is useful only to give to the simulation engine so that it knows a particular agent types exists without having to specify how it would be started up. Note that the communication server which is responsible for starting the agent will have to have a fuller description of the type.

4.5.4 Working with the Agent Database

The writer of a world model must access the agent database in order to start up agents. This section describes how the agent database can be accessed.

First, the agent database is accessible through the `getAgentTypeDB` method of the `SimEngine`. The method returns a pointer to the agent type database, an object of type `AgentTypeDB`.

`AgentTypeDB` includes the types `AgentTypeIterator` and `AgentTypeConstIterator` which are the primary ways to access the agent types in the agent database.

The following methods of `AgentTypeDB` are useful for working with the agent types. They are come in two varieties, taking/returning a regular or a constant iterator.

`isIteratorValid` Returns whether the given iterator is valid (points to a real agent type).

`nullIterator` Returns the null (i.e. invalid) iterator.

`deref` Takes an iterator and returns the associated agent type. You should not dereference an invalid (i.e. null) iterator.

`getAgentType` Takes the name of an agent type, looks it up in the database, and returns an iterator to the agent type. Returns an invalid iterator if there is no agent type with that name in the database.

`getBeginIterator` Returns an iterator to the first agent type in the database.

The `AgentTypeIterator` supports the `++` operator to advance to the next type.

4.6 Agent Interface

This section describes in general terms the interface to the agents both from the agent and world model perspectives. Exact formating and other details are discussed in Section 6.4.

4.6.1 External Agent Perspective

An agent process is initiated by the communication server using the information in the agent database. The pipes for communicating with the communication server are initialized as discussed in Section 6.4.

The agent will then be given an initialization data message. If you are using agent migration (Section 5.3) and this is a migrated agent, this data will include information about the migration. If you are not using migration or this is the first execution of an agent, this data will be empty and can be ignored.

The agent should respond with a initialization done message once all initialization steps are completed. This allows the agent to do an arbitrary amount of untimed processing at startup.

Once the agent has completed its own initialization, it simply needs to wait for messages to be received from the communication server. A variety of messages can be received from the communication server but they can be broken into two classes, those that begin thinking cycles and those that do not.

Sensations and time notifies begin a thinking cycle where the agent can respond with actions. All thinking cycles are finished by a done thinking message to notify the communication server that the agent has finished.

All the other messages are non-thinking sensations and the agent can not respond with any actions. Note that any computation time used in processing these messages will be charged to the next thinking cycle.

The agent has two primary actions as far as SPADES is concerned. The first is an act message whose format is completely interpreted by the world model. The second is a request time notify which requests that a time notify is sent to the agent at the specified time. At the end of a thinking cycle, an agent must always send a done thinking message.

Two important parameters must be mentioned about the agent interface. With the tracking of computation time for the agents, it is possible for an agent to fall behind in the simulation. For example, the agent may be schedule to receive a sensation at time x, but because of the thinking done for the last sensation, the agent is already at time $x + 1$. If an agent is ever ahead of the arrive time of a sensation by more than *max_agentq_trailing_time*, then the sensation is turned in an inform, a non-thinking sensation. This is done by changing the `ThinkingType` element to `TT_Not`. *max_timenot_trailing_time* performs the same function, but for time notifies instead of sensations.

If the parameter *send_agent_think_times* is on, the agent will receive a think time message reporting how much simulation time was used for the last thinking cycle. This is a type of inform message in that it does not start a thinking cycle.

Agents will be sent error messages for various out of order and malformatted message errors. A full list can be found in Section 6.4.1.

Various time limits can be set up to control how long an agent has to think. This can notably be used to prevent a malfunctioning agent from interfering with the remainder of the simulation as well as providing detection facilities for this. Section 4.7.3 discusses this process.

SPADES provides logging of the standard out and standard error of every agent, though this can be turned off with the *create_agent_logfiles* parameter. The parameters *agent_stdout_log_fpat* and *agent_stderr_log_fpat* control the location and name of these files.

Agents can therefore use standard out for logging or other recording of information. It is always a good idea to check the standard error of all agents to insure that no errors were encountered in a simulation.

4.6.2 Integrated Agent Perspective

An integrated agent has much the same kind of interaction with the communication server as an external agents (as described in Section 4.6.1). The differences are:

- Communication is done with method calls, not via pipes.
- A string can be specified in the agent database to be passed to the integrated agent on initialization (through the `initialize` method). This allows something like the command line arguments that can be specified for external agents.
- An integrated agent provides an object which is a subclass of `IntegratedAgent`. The methods of `IntegratedAgent` are called to send messages to the agents.
- To send messages back to the communication server, the integrated agent calls methods of the `IntegratedAgentActions` object provided to the `IntegratedAgent` object representing the agent.
- Standard out and standard error of the agent are, naturally, the standard out and standard error of the communication server. Therefore, the agent should probably open its own files to log to.
- None of the time limits on thinking mentioned in Section 4.6.1 apply to integrated agents.

A full description of the classes involved is given in Section 6.5.

4.6.3 World Model Perspective

The world model interacts with agent through events and some method calls. Figure 4.2 shows the sense think act cycle of an agent from the world model's perspective. The cycle begins with an event of type `CreateSenseEvent` being realized. This enqueues an event of type `SenseEvent`. The time between the `CreateSenseEvent` and `SenseEvent` is the sensation latency. Upon realization of the `SenseEvent`, a message is sent to the agent and a thinking cycle is begun. The agent responds with some number of actions, and based upon the computation time used, the actions are assigned a time. The difference between the time of the `SenseEvent` and the simulation time assigned to the returned act messages is the thinking latency. The `parseAct` method of the `WorldModel` object is called to turn the act message into an event of type `ActEvent`. The

`ActEvent` is then realized to complete the cycle. The time between the time of the act messages and the `ActEvent` is known as the action latency.

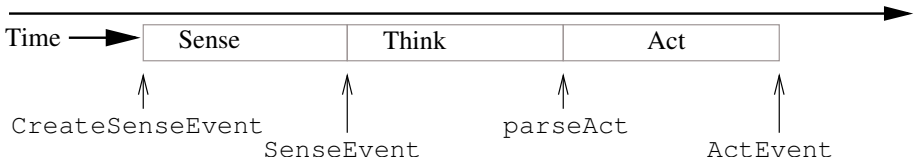

Figure 4.2: Time line showing the events and method calls relevant for the sense think act cycle of the agents

Note that a `CreateSenseEvent` is not the only way for a `SenseEvent` to be enqueued. However, it is recommended that whenever possible you do use it. SPADES does additional error checking with `CreateSenseEvent` to avoid causality problems which can be hard to detect otherwise.

`CreateSenseEvent` can easily be used to create regularly reoccuring sensations. The realization of `CreateSenseEvent` will cause the `createSense` method to be class to generate the `SenseEvent`. You can then define the `realizeEventWorldModel` method to enqueue a new `CreateSenseEvent` at the appropriate time in the future.

4.7 Agent Monitoring

This section covers the ways that a SPADES communication server monitors the agents for which it is responsible.

4.7.1 Agent Timers

SPADES provides a variety of ways in which thinking times of agents can be measured/computed. Note also that the world model can generate untimed sensations which, while they begin thinking cycles, always have a thinking latency of 0.

No matter what the method, SPADES can record the thinking latency used for the simulation. The parameter *record_think_times* controls whether this is done. The times are stored in a separate file for each agent, and the parameter *think_times_file_pattern* specifies the file name. The details of the file format (it is binary, not text) can be found in Section 6.7.3.

A perl script named `show_ttimes.pl` is provided which can read the binary format and output it in a more human readable format. It takes the name of a file to read and (optionally) how many think times to show per line. The values in the file are printed left to right then top to bottom.

Some timers can use be used for integrated agents (agents which are loaded from a dynamic library) and others can not. This is noted below.

A timer is specified either in the agent type database or the with the *default_process_timer* parameter. In both cases, a string is parsed to get the type of timer and its arguments. The valid values are:

- '`fixed` *`time`*'

  This is the fixed time timer. Every thinking request is given the same time *`time`*. This can be used for an integrated agent.

- '`replay` *`filepattern`*'

  This is the replay timer. The *`filepattern`* argument specifies the files to read. Just as in the *think_times_file_pattern*, the string '`%N`' is replaced with the agent number. Section 6.7.3 describes the file format, but just to replay the times SPADES previously recorded, it is not necessary to understand this.

  During replay, *replay_think_buffer_size* values from the file are read at one time. These is an efficiency enhancment, though changing this parameter will probably have at most minor effects on performance.

  This can be used for an integrated agent.

- '`jiffies` *`kinst`*'

  This is the jiffies timer. A "jiffy" is an amount of time scheduled on the processor as reported by the Linux kernel. The integer parameter *`kinst`* translates to the number of kilo-instructions per simulation step. See Section 6.7.1 for details. This can not be used for an integrated agent.

- '`perfctr_instr` *`kinst`*'

  This is the perfctr timer. By using features provided by the perfctr system[3], the number of processor instructions the agent uses are tracked. The integer parameter *`kinst`* translates to the number of kilo-instructions per simulation step. See Section 6.7.2 for details. This can not be used for an integrated agent.

- '`default`'

  This is the default timer. Whatever timer is specified with the parameter *default_process_timer* is used. If '`default`' is given as the value of *default_process_timer*, then '`fixed 0`' is used. This can be used with an integrated agent if the timer referred to by default is acceptable for an integrated agent.

[3] `http://sourceforge.net/projects/perfctr/`

4.7.2 Agent Process Tracking

SPADES also tries to track when external agent processes spawn other processes in order to properly time the agents as discussed in Section 4.7.1. This section describes how this is done and the relevant parameters. However, it should be noted that this scheme is neither perfect or foolproof. It is designed to make an agent designer's job easier and try to avoid accidental mistakes. It is still possible for an agent designer to fool the system. Also note that this only applies to external agents, not integrated ones.

The tracking works by SPADES uses the `LD_PRELOAD` mechanism of Linux to load a dynamic library which intercepts dynamic library calls. All dynamic library calls to `fork`, `vfork`, and `clone` are caught by the library to notify the commserver. Note that calls to `pthread_create` are implemented using `clone`, so it is also tracked. The parameter *agent_intercept_library* controls what library is preloaded. In general, SPADES will set this automatically, so you should not need to worry it.

Note that this means that any agent which does not use dynamic library calls to spawn new processes (e.g. an agent executable which is statically linked) will *not* have its forking tracked correctly. Currently, SPADES does nothing to try and verify that agents are dynamically linked, and it would be difficult to try and verify that all `fork` calls are done dynamically.

The notifications of forking are done via the SysV IPC Message Queue scheme. One implication is that only one communication server can be run per machine per user.[4] The other is that if a communication server exits abnormally, the next time you may get an error like this:

```
The IPC message queue seemed to already exist.
```

If you use the *ipc_force_remove* parameter, the message queue will be removed before the communication server starts.

You can disable the IPC message reception with the *enable_ipc_message_reception* parameter.

4.7.3 Checking on Agents

At the end of every message processing cycle, the communication server "checks up" on the agents. This is mainly to catch errors like crashed agents, but also allows some housekeeping to be done. Note that this only applies to external agents.

The steps are:

- Check that all processes which are part of the agent still exist. If they do not, it is not an error. The proceses are just removed from the tracked processes list for that agent. However, if all agents processes are gone, the agent is shutdown with status `ALR_ProcessVanished`.
- The wall clock time that the agent has been thinking is then computed. If it is greater than *max_secs_for_agent_think*, then the agent is shutdown with status `ALR_ThinkTooLongWallClock`.

[4] The user ID is part of the message queue identification, so multiple users can run communication servers, but the same user can run only one.

It is recommended that you set the parameter *max_secs_for_agent_think* quite high relative to the expected think time for the agent. This check should ideally ony catch agents in infinite loops or who fail to send a done thinking message. For example, it my current setup with the sample world model and agents, I expect the agent to think for at most 50ms, but I have *max_secs_for_agent_think* set to 10s.

- Based on the computed wall clock thinking time, a decision is made whether to do further checking. If *agent_check_use_randomness* is off, and the wall clock think time is greater than *agent_check_threshold_sec*, then further checking is done.

  If *agent_check_use_randomness* is on, then a threshold is computed from a Gumbel distribution with parameters *agent_check_gumbel_dist_A* and *agent_check_gumbel_dist_B* and the input as the wall clock time. With that probability, further checking is done.

- The process timer (see Section 6.7) is then consulted to compute a time in simulation time that this agent has been thinking. If this time is greater than *max_simtime_for_agent_think*, then the agent is shutdown with status `ALR_ThinkTooLongSim`

- If the simulation time the agent has been thinking has increased, a time update message is sent to the simulation engine. This should be completely transparent to the world model and agents. The time update should have no effect on the results of simulation, but can affect the ordering of non-causally related events for efficiency.

4.8 Monitor

The monitor interface allows external programs to connect to the simulation and get periodic updates about the state of the simulated world. Monitors can also send information back to the engine and world model to perform various functions like pausing and unpausing the simulation.

The simulation engine will only accept monitor connections if *accept_monitor_connections* is true. The monitor should connect over a TCP socket to the port *monitor_port*. After each connection, the engine calls the `getMonitorHeaderInfo` method of `WorldModel`. This initialization or header information is sent to the monitor. SPADES does not constrain the format of the information.

At time 0 and at every *monitor_interval* simulation steps thereafter, a `MonitorSendEvent` is enqueued. The realization of this event causes the method `getMonitorInfo` of `WorldModel` to be called and the information returned is sent to all monitors. SPADES does not constrain the format of the information.

The world model can also send extra information via the `sendExtraMonitorInfo` method of `SimEngine`. This should be used for any asynchronous updates.

SPADES can also create a monitor log file which records all information which would be sent to a connected monitor during the simulation. If *use_monitor_log*, all monitor information (including the header information) is written to *monitor_log_fn*.

A monitor can send commands to pause, run normal, run limited rate (see Section 5.6), disconnect from the simulation, shutdown the simulation, and send arbitrary bytes to the world model. Section 6.6 describes the details of the monitor interface.

4.9 DataArray

`DataArray` is a utility class which maintains a reference counted array of bytes, and stores the number of bytes. This means that a single copy of an array of bytes is maintained with arbitrarily many references encapsulated in a `DataArray` object. When the last of these references is deleted, the array of data is deleted automatically. In cases where the identical messages are sent to many agents, this should provide significant savings. This section will highlight and explain some of the important methods, but please see the source code for the exact details.

- `DataArray(const char* buffer, unsigned length)`

  This constructor copies the data in the buffer given. If you want `DataArray` to take over the memory, see `takeData`.

- `DataArray(std::stringstream& str)`, `DataArray(std::streambuf& strbuf)`, `DataArray(std::string str)`

  All of these constructors copy the data in the object given to them.

- `DataArray(std::ostrstream& ostr)`

  This constructor takes over the memory contained in the `ostrstream`. You should *not* call `freeze(0)` on the `ostrstream` after this call, or manipulate the `ostrstream` in any way.

- `void copyData(const char* data, unsigned length)`

  This method replaces any data currently stored with a copy of the data given.

- `void takeData(char* data, unsigned length)`

  This method does not copy the data, but takes over responsibility for the memory passed in. You should not manipulate the data in `data` after this call.

- `const char* getData() const`

  Used to access the actual array of bytes, but you should not store this value. If you need to keep a reference, simply use the copy constructor of `DataArray`..

- `unsigned getSize() const`

  This returns the number of bytes in the array of data controlled by this object.

4.10 Achieving Parallelism

Achieving good speedups when running a simulation on multiple machines is a tricky problem. The technical details of the algorithms used by SPADES are described in Section 6.8, but this section will describe some of the issues and gives advice to world model designers on how to achieve good parallel speedups.

A thorough experimental analysis of the effects of the various settings described here has not been done, so the advice here is based upon knowledge of the algorithms and experience in running SPADES based simulations.

SPADES achieves parallelism through two mechanisms. The first is identifying the times at which each agent can next enqueue an action. This allows SPADES to realize events with out having to wait for a reply from an agent. This revolves around the use of the the minimum action latency to determine the minimum agent time. The minimum agent time is the earliest time for which an agent can enqueue a new action. The method `getMinActionLatency` of `WorldModel` gives the minimum action latency.

In general of course, the longer the minimum action latency, the more parallelism can be achieved since more events can be realized at one time. This means that very short action latencies for any sort of special action should be avoided, since the minimum action latency is the minimum over all actions in the simulation.

The other mechanism involves identifying non-causally related events so that they can be realized out of order. The events of interest here are sensations and time notifies. Since the content of sensation and time notify messages is fixed, these events can be realized as soon as it is identified that no other message will be sent to this agent before this event. This decision revolves around the value of the minimum sensation latency, is the smallest time between when an event is realized and a new sensation for an agent can be enqueued. The minimum sensation time is the earliest time that a new sensation can be generated for any agent. Any sensations or time notifies for an agent before the minimum sensation time can be realized.

The longer the minimum sensation latency, the more parallelism can be achieved. Therefore, you should avoid any special case sensations with very short latencies, as the minimum sensation time as returned by the method `getMinSenseLatency` is the minimum over all types of sensation generating events.

If agents are aligned such that sensations (i.e. thinking cycles) are generated for the same simulation time, then it is easy to achieve parallelism. Therefore, in cases where you (as the world designer) can create situations like this, you should. Note however, that you should not sacrifice realism, as the ability to generate non-synchronous actions is one of the strengths of SPADES. For example, for a group of robots on a field, you would not expect synchronous vision updates, and it is not recommended that you do so. Note that SPADES does a good job our opportunistically taking advantage of whatever parallelism is available at a given time.

In the experiments conducted with SPADES thus far [Riley, 2003, Riley and Riley, 2003], the time that sensations were generated had a small random component. This helps insure that on average, there is parallelism, and bad cases of agent distribution and sensation times will not

persist.

Another interesting point concerning parallelism is load balancing among machines. In previous experiments [Riley and Riley, 2003], often no parallel speedups were achieved until the maximum number of agents on a machine dropped. For example, with 12 agents, moving from four machines to five machines sees no additional parallel speedup (since both cases have a machine with three agents). However, moving from five machines to six machines does see an increase in parallel speedup since the maximum number of agents per machine drops to two.

Agent migration could also be used to help perform load balancing. However, the current algorithm for making migration decisions is not very sophisticated. This will hopefully be improved in the future.

4.11 Randomness and Reproducibility

Another key feature desired in many simulations is reproducibility. Here, "reproducibility" will mean that the same simulation state can be achieved by rerunning the simulation in the same starting configuration.

First, the exact ordering guarantees provided by SPADES are discussed in Section 4.2.1. Since the realization of a fixed agent events should not affect the world state in any way, the primary guarantee is that all non-fixed agent events will be realized in time order.

For true reproducibility, you may want a stronger guarantee than just time order as there may be many events at the same time. In particular, you may want to guarantee a particular order of event realization. To do this, you should ensure that you define the ordering constants and `getSecondaryOrder` method of the `Event` class and its subclasses appropriately. That is, you should ensure that no two non-identical events have the some ordering constant and same value for `getSecondaryOrder`.

Further, the `parseAct` method of the `WorldModel` class should not affect the state of the WorldModel in any way. `parseAct` calls come in at uncontrolled times, so no state change should be allowed.

If you use pseudo-randomness in your simulation, things are slightly more complicated. There are some natural reasons to use randomness in the parsing of actions (to pick an action latency for example). However, to maintain reproducibility, you can not use pseudo-randomness in either the `parseAct` method or the realization of any fixed agent event. If you do want to have random act latencies, one possibility is for parseAct to put a place holder event for the current time plus the minimum action latency. The realization of that event can case the real action event to be queued at a random time forward.

Additionally, the `agentConnect` method can not use psuedo-randomness since the order of connection can not be controlled.

SPADES provides some functionality to help deal with pseudo-randomness. SPADES handles the seeding of the random number generator so that the world model does not need to do it. The parameter *random_seed* allows you to specify what random seed to use. Further, if the

parameter *print_random_seed_to_stdout* is true, then the random seed being used (either specified by *random_seed* or read from `/dev/urandom`) is printed to standard out. This allows you to recreate simulations if need be. Lastly, if you are using agent migration, you may need to use the *use_randomness* parameter to control whether the migration choice is random.

The most difficult part in providing completely reproducible results is in tracking of the agent thinking time. In the jiffies timer, the CPU time reported by the kernel varies somewhat based on other system activity in unpredictable ways. To alleviate these problems somewhat, you can use the fixed time timer (see Section 4.7.1) to turn off the tracking and always return a fixed value for the think latency. Alternatively, you can use the perfctr timer to get accurate instruction level accurate timings. Note, however, that this currently requires a patched kernel. Also, think times can be recorded and played back with the replay timer (see Section 4.7.1).

There is one other important note in achieving perfect reproducibility. It is natural to write the `simToTime` method to look something like this:

```
for (SimTime t = time_curr;
     t < time_desired;
     t++)
  {
    per step code
  }
finish stepping code
```

That is, the `simToTime` method may be able to amortize some of the cost of stepping forward by having code in the *`finish stepping code`* section. Especially when using floating point calculations, this will cause perfect reproducibility to be lost. The simToTime method is *not* guarunteed to be called in exactly the same way each run. Rounding at the lower order of the floating point numbers can eventually creep in to affect the overall results. Therefore, if you want perfect reproducibility, the *`finish stepping code`* should not contain anything which affects the state of the world model.

The technical paper Riley and Riley [2003] discusses these issues of reproducibility further and provides some experimental evidence of the reproducibility of SPADES.

Chapter 5

Miscellaneous Features

This chapter covers other features of SPADES that are not discussed in Chapter 4. This features are not essential for either creating a SPADES simulation or for understanding the basic functioning of the system. However, if you intend to use SPADES much, these features may be of interest to you.

5.1 Action and Error Logging

SPADES provides facilities for error, warning, and action logging. Errors are generally internal errors that should never occur. However, not all errors represent bugs in SPADES. Warnings are exceptional conditions that do not represent errors in system design, but may mean that this simulation is not running properly. Action logging is based upon the layered disclosure idea [Riley et al., 2001], and is used to track system actions for debugging.

You are encouraged to use these facilities for your own logging purposes.

5.1.1 Basic Usage

The classes are defined in the file `Logger.hpp`. The primary class is named `Logger` and follows the Singleton design pattern. This means that there should only be a single instance of the `Logger` and it is accessible through the static member function `instance`.

Several `#define` statements are used to provide easy access to the `Logger` facilities, and writing to the logs is done with standard C++ streams. The following `#define`'d statements convert into a expression which takes the `<<` operator to send information to the logger. In order to end a logging expression, you use the `ende` operator. Note however, that these are not in fact true stream types, so you can not do more complicated I/O on them (such as the STL uses).

- `errorlog` Use for critical errors and conditions that should never occur. All messages are printed to standard error and to the action log file at level 0 (if it is being used).

- `warninglog(x)` Use for exceptional conditions that do not indicate unrecoverable errors or bugs in the system. The variable `x` represents a logging level; I currently use 10 for all warnings, but feel free to assign your own meanings. All messages are printed to standard error and to the action log file at level 0 (if it is being used).
- `actionlog(x)` Use to log information about the execution of the program. Don't worry about how efficient it is to write data here because a compile flag can be used to remove all `actionlog` code. The variable `x` represents the logging level. The weakness of the layered disclosure idea is how to define the right logging level. In general, higher numbers mean more detailed information, and the logger provides a way to print only those action log messages below a certain level. All messages are printed to the action log file. By default, the logger prints the simulation time and a number of dashes equal to the level divided by 10. SPADES uses the following conventions:
  - Lower log levels are less detailed information.
  - Use levels in multiples of 10, in case you need to fill in levels in between.
  - All log levels above 200 may come up in inner loops and generate a great deal of output very quickly. This should only be used for targeted debugging.
  - The log levels below 100 give enough information for an understanding of the general control flow of a simulation run.
  - All levels are non-negative.

Here are a few examples of valid messages and how they will be printed in SPADES:

```
errorlog << "This is an error condition" << ende;
warninglog(10) << "I'm feeling a little sick today" << ende;
actionlog(150) << "I started running at time: " << time << ende;
actionlog(50) << "I just realized an event for you" << ende;
```

Note the use of `ende` manipulator to finish the logging expression. You can use newlines in your logging, but SPADES does not in any of its own logging.

These examples would write this to standard error:

```
EngineError(1234): This is an error condition
EngineWarning[10](1234): I'm feeling a little sick today.
```

Note that the warning message gives the warning level (10). The number is parentheses is the simulation time at the point of the error.

The example above will write this to the action log file:

```
1234 EngineError(1234): This is an error condition
1234 EngineWarning[10](1234): I'm feeling a little sick today.
1234 ---------------I started running at time 12/5/2002 12:12p
1234 -----I just realized an event for you
```

Note the use of the '`-`' character to indicate action log level.

5.1.2 Parameters

If the variable `NO_ACTION_LOG` is defined (e.g. by a `-DNO_ACTION_LOG` flag to the compiler), all `actionlog` statements are put inside of `if(0)` conditions, and will be removed by a reasonable compiler.

There are several parameters that affect how the logger works.

- *action_log_fn* This is the file name of the action log file.
- *action_log_level* Which action log commands to write to the file. All log messages at levels equal to or below this value will be written to the log file. All those levels above this value will be ignored. At a level of 0, only errors will be written to the file.

5.1.3 Advanced Usage

The sections above do not really give the whole story. The `Logger` is actually a little bit more general purpose than specified above. This section will not give a complete description, but give you some highlights of how this works. You should look at the code for details.

The logger uses a `TagFunction` which specifies the leader for every logging statement, including the error, warning, and action logs. The leader includes (in the above examples) all simulation times, '`-`' leaders for actions, and the "EngineError" and "EngineWarning" tags.

The error log and warning log use the same facilities; errors are simply warnings at level 0.

If you want to change where the logging output goes, you can use the `setLoggingStreams` method to set the output of the logger to C++ streams.

You have to be somewhat careful during the shutdown process. `Logger::removeInstance` is called to remove the singleton instance. Also, files are not opened until an explicit call, so action logs too early in the initialization process will go to standard out instead of to the action log file.

5.2 Parameter Reading

This section describes the parameter reading interface as seen by the code which makes up the processes. Section 6.1 describes how the parameters should be formatted in configuration files and on the command line.

All parameter classes are subclasses of `ParamReader`, which provides the basic processing functionality. The important methods are ParamReader are:

`ParamReader` The constructor takes a maximum version value. All configuration files are required to have a version line indicating the version of the file. If the version of a file being read is greater than the max version given here, then an error is given.

`getOptions` This is the function which actually does the argument and file parsing.

addAll2Maps This pure virtual method is called to add all parameters to the string to data type maps. For every subclass, you will need to define your own version of `addAll2Maps` to call `add2Maps` for all the parameters in your parameter class. It is important that every child parameter class calls its parents `addAll2Maps` in this function, like `ParentParam::addAll2Maps()`.

setDefaultValues This pure virtual method is called to set all parameters to default values. This is used for initialization and in case a given parameter is never set. It is important that every child parameter class calls its parents `setDefaultValues` in this function, like `ParentParam::setDefaultValues()`.

postReadProcessing This method will be called after the values have been read, though it may be called several times. It is important that every child parameter class calls its parents `postReadProcessing` in this function, like `ParentParam::postReadProcessing()`.

add2Maps This overloaded method is used to add parameters of all types into the map of parameter names to value. It should probably only be called from inside `addAll2Maps`.

addParamStorer Rather than using the map based mechanism, you can define your own `ParamStorer`. Please see the code for details, but basically, your class will be notified every time a parameter is read and it is not found in the maps.

In order to create your own parameter class, you will need to subclass `ParamReader` or one of it's descendants and redefine the virtual functions listed above. If you subclass anything except `ParamReader`, make sure you follow the instructions above for calling the parent methods in `addAll2Maps`, `setDefaultValues`, and `postReadProcessing`.

You may also want your parameter class to follow the Singleton design pattern, where only one object of this type can exist. See `EngineParam` for an example.

In order to add a parameter to a parameter class, you have to do four or five things:

1. Add a member to your parameter class to store the value. The convention used thus far in SPADES is that members are all lowercase with words separated by underscores.

2. Add an access method for the member. The convention is to preface the member name with `get`, capitalize (at least) the first letter of each word, and remove the underscores.

3. Add the appropriate call to `add2Maps` in the method `addAll2Maps`. The convention is that the external name of the parameter is the same as the member name.

4. Add a line with a default value to `setDefaultValues`.

5. Optionally, you may need to add code to `postReadProcessing` in order to deal with or further process the parameter value given.

5.3 Agent Migration

Agent migration refers to moving agents between communication servers in order to try and load balance better and thereby improve the overall efficiency of the simulation. Often the speed of the simulation is held back by the speed of the slowest agent, so load balancing has the potential to make an important difference in the efficiency of the simulation.

Agent migration is supported by SPADES, but the sophistication of the mechanism is not very high. Supporting migration puts a significant burden on the agents themselves, so in many cases it may not be worth supporting this feature.

Migration is turned on by the *use_migration* parameter. If migration is turned on, the simulation engine tracks the amount of wall clock time it takes an agent to respond to a sensation. Once at least *min_responses_before_migration* responses have been received from the agent, the agent is a candidate for migration. The agent will be migrated if its response time is greater than twice the mean response time over all agents.

When an agent is migrated, first a migration request message is sent to the agent. The agent should respond with a migration data message which includes the all the data representing the current state of the agent. That agent process is shut down and a new agent process is started on another machine. The initialization data that the agent is given is the migration data from the agent process that was shut down. The simulation then continues as normal.

5.4 Integrated Communication Server

The simulation engine provides a facility to run an "integrated communication server." This simply means that communication server runs in the same process as the simulation engine and world model. Therefore, messages do not have to be serialized and sent to a socket.

The provides a small reduction in communication costs for a communication server running on the same machine as the simulation engine. In a small set of possibly non-representative experiments, this speedup is about 5%.

The use of the integrated communication server is controlled by the parameter *run_integrated_commserver*. All agents and the world model should find it indistinguishable whether an integrated or regular communication server is being used.

5.5 Agent Shutdown Management

When a communication server wants an agent process to exit, it goes through this process:

1. If the exit is initiated by the communication server, and not by agent request or some sort of failure on the agent's part, an exit message is sent.

2. `SIGTERM` is sent to the agent processes.

3. Periodically, the agent processes are checked to see if they still exist.

4. If an agent process does exit, the child process should be reaped immediately, so no zombie processes should exist for an appreciable amount of time.

5. If the agent process does not exit in *secs_for_agent_shutdown* seconds, then a `SIGKILL` is sent to forcibly kill the process. If this must be done, a warning is will be written to the warning log (see Section 5.1).

Note that if your agents write data or perform other time intensive tasks at the end of the simulation time, you may exceed the default time of 2.0 seconds for *secs_for_agent_shutdown*.

5.6 Limited Rate Run Mode

SPADES supports a run mode where the speed of the simulation is limited. This is useful, for example, in order to maintain a "real-time" performance for human interaction or monitoring. Note that SPADES does not guarantee that this rate will be maintained, it only guarantees that the simulation rate will not *exceed* the given value.

The simulation is put into the limited rate run mode by changing the simulation mode to `SM_RunLimitedRate`. This can be done by the world model or by a command from the monitor (see Section 6.6).

In limited rate mode, the parameter *limited_rate_default_st_per_sec* controls the speed of the simulation.[1] SPADES aims to have *limited_rate_default_st_per_sec* simulation steps advanced with all events realized every second. Note that the simulation engine prints out the average amount of simulation time advanced per second, so you can track if the desired performance is being achieved.

The algorithm for doing this is straightforward. Call the desired number of simulation time steps per seconds R. A base wall clock (B_w) and simulation (B_s) is taken. Before realizing an event, the desired wall clock time for that event t to occur is calculated by:

$$\frac{t - B_s}{R} + B_w \tag{5.1}$$

If that time has not yet arrived, the simulation engine sleeps (with the `select` system call) until the desired time for the event.

There are several points to note about this algorithm. First, only the wall clock time of event realization is controlled. The simulation time as advanced by the world model is not checked explicitly. Secondly, if the process of event realization is computationally intensive, it is possible that SPADES will fall behind, even when there is enough processing power available. Lastly, fixed agent events are not controlled in the same way; the same lookahead algorithm (described in Section 6.8) is applied even if the wall clock time for the next regular event to be realized has not yet arrived.

[1] Currently, the rate is fixed by this parameter. In the future, the world model and the monitor interface will allow this rate to be adjusted.

Every time the simulation mode changes to the limited rate run mode, “rebasing” is done for the base simulation and wall clock times. Additionally, every *limited_rate_rebase_interval* seconds, rebasing is done. Notably, if there is a machine or network slowdown, the simulation will not indefinitely try to to catch up.

Chapter 6

Technical Details

This chapter provides a myriad of technical details about various aspects of SPADES. The one thing that holds everything in this chapter together is that there were too many details to go over in other sections of the manual. If you just want to get a general understanding of how SPADES works, you can probably skip this chapter until you need it.

6.1 Parameters

This section describes all the parameters that SPADES understands. In general, only brief descriptions of the parameters are given. All parameters that need further explanation are explained elsewhere in the manual. See the index under the parameter name to find where the information is.

All parameters can be given in one of two ways (where *`param`* is the name of a parameter and *`value`* is the value):

- On the command line as '`--`*`param value`*'
- In a configuration file, with a line of '*`param`*`:  `*`value`*'

Configuration files to read can be specified on the command line as '`--file` *`filename`*' or configuration files can be nested by putting '`file:  `*`filename`*' in a configuration file. There is a maximum nesting of configuration files of 16.[1] A relative path from a `file` command inside of a configuration file is resolved relative to the location of the configuration file (*not* the current working directory).

Further, every configuration file must specify a version number. The version is used during parameter reading to insure that the version of the software reading the configuration file is at least as new as the configuration file (see Section 5.2 for details on the programmer's interface). The version line must be the first non-comment and non-whitespace line in the configuration file, and looks like this: '`version:  `*`num`*' where *`num`* is the version of the file.

[1] You can change this in the file `shared/FileReader.hpp` if need be

As far as order of parameter processing, all configuration files specified by `--file` on the command line are read before the rest of the command line is processed. For nested configuration files, the entire sub-file is processed at the point of the `file` command, then the rest of the original file is processed.

There are eight types of parameters:

String Arbitrary length character string. If a value is not specified, the empty string is set for the parameter values.

File Path This is also a character string, but unlike 'String' the value is resolved to an absolute path name. On the command line, this is done relative to the current working directory. In configuration files, this is done relative to the location of the conf file. There are two exceptions to this resolution: If the value begins with a '`%`', the value is left untouched (this is done because many parameters use a '`%X`' format to expand values). The other is if the value begins with '`&`', the '`&`' is removed and the rest of the value remains untouched. This could allow, for example, for a file path to be specified in a conf file that is relative to the current working directory.

Integer An integer value. This always requires a value. In general this can be positive or negative, but for some parameters, only certain values make sense, as noted in the tables below.

Real A real value, stored as a double precision floating point number. This always requires a value.

Boolean An on/off value. This does *not* require a value. Without a value, the parameter is turned on. The only valid values are "on" and "off".

Vector(Integer) A vector of integers. A minimum and (optionally) maximum number of entries can be set. A warning will be printed if these constraints are not honored. For this manual, the bounds will be enclosed in `[]`. The values are white space separated in conf files, e.g. '`2 12 47`'.

Vector(Real) A vector of real values, with the same properties as Vector(Integer).

Vector(String) A vector of string values. The entries are in the conf file. On the command line, an entry for a vector of strings can not begin with `--` (because they looks like the next parameter is starting).

Note that Vector(Integer), Vector(Real), and Vector(String) are not used for any of the parameters provided by SPADES (though the sample world model uses some of them), but are provided for the convenience of world model designers.

6.1.1 Shared Parameters

These are parameters which both the communication server and simulation engine understand.

Name	**Type** **Default** **Description**
version	Boolean off If this is specified, the version is printed and the communication server/simulation engine will exit.
logfile_dir	File Path Logfiles This directory is used in many other parameters as a replacement for '`%D`'. This shoudl allow easy redirection of all logfiles to a given directory.
create_logfile_dir	Boolean on If *logfile_dir* does not exist and this parameter is specified, then the logfile directory is created. Only one level of directory will be created (like `mkdir`, not `mkdir -p`).
action_log_fn	File Path %D/actions.log The name of the action log file. The string '`%D`' is replaced by the *logfile_dir*.
action_log_level	Integer(≥ 0) 0 Which actions to log; all levels less than or equal to this value are written to the logfile.
engine_port	Integer 12000 The port on which the simulation engine listens for communication server connections.
use_randomness	Boolean on Whether to use any randomness to break ties. The only place this is currently used is in migration decisions for agents.
random_seed	Integer -1 Any negative value means to read a new random seed. Any positive value specifies the random seed to use.
print_random_seed_to_stdout	Integer off Whether to print the random seed used to standard output. This is useful for reproducing a simulation.
internal_tcp_packet_size	Integer 1024 SPADES manually gathers data to avoid doing many small writes to a socket. This parameter specifies how many bytes should be collected before sending. Any non-positive value turns off this collection.

Continued on next page

Name	**Type** **Default** **Description**
agent_db_fn	File Path agentdb.list The filename for the agent database.
status_update_interval	Integer 5 How often (in seconds) to print simulation status updates to standard out. Any non-positive value turns off status updates.
trace_on_error	Boolean on On various errors like timeouts and unexpected failures, some useful logging information is printed. It is strongly recommended that you leave this on, as this information is extremely useful to track down bugs.
create_agent_logfiles	Boolean on Whether to create logfiles for the standard error and standard output of the agents.
secs_for_agent_shutdown	Real(≥ 0) 2.0 How many seconds to give the agent processes to exit from the time the termination signal is sent. If the agent processes do not exit in this time, they are killed forcibly.
agent_packet_size	Integer 1024 SPADES manually gathers data to avoid doing many small writes to the agent input. This parameter specifies how much data should be collected before sending. Any non-positive value turns off this collection.
default_agent_input_fd	Integer(≥ 3) 3 This is the default file descriptor for the pipe from which the agent will accept input (the agent type can override this). The value must be greater than or equal to 3 since file descriptors 0,1,and 2 are standard in, standard out, and standard error respectively.
default_agent_output_fd	Integer(≥ 3) 4 This is the default file descriptor for the pipe to which the agent will write its actions (the agent type can override this). The value must be greater than or equal to 3 since file descriptors 0,1,and 2 are standard in, standard out, and standard error respectively.

Continued on next page

Name	**Type** **Default** **Description**
load_send_interval	Real(≥ 0) 2.0 How often (in seconds) to send the load values of the machine to the simulation engine. This is in the shared parameters because of the possible use of an integrated communication server.
max_unnatural_lost_agents	Integer 10 If this many agents exit unnaturally (that is, without being given an explicit exit or explicitly requesting an exit), the communication server exits. Any negative value disables this. This is in the shared parameters for the integrated communication server.
send_agent_think_times	Boolean on Whether to send think time messages to the agents.
send_agent_send_time	Boolean on Whether to send the send time for a sensation or inform message. The send time is the time at which the sensation was generated.
send_agent_arrive_time	Boolean on Whether to send the arrive time for a sensation or inform message. The arrive time is the time at which the sensation is received by the agent.
default_process_timer	String fixed 0 Specifies the timer to use when the default timer is specified for an agent. If this value is the default timer, then "fixed 0" is used.
record_think_times	Boolean off Whether to record the think latencies of the agents to the files given by *think_times_file_pattern*.
think_times_file_pattern	File Path %D/agent%A-ttimes.log Specifies the files to which to record the agents' thinking latencies. The string '%A' is replaced by the agent number and '%D' by the *logfile_dir* to get the file name for an agent.
replay_think_buffer_size	Integer 64 Gives the number of values to read at once when using the replay timer. There is currently no reason to change this parameter.

Continued on next page

Name	**Type** **Default** **Description**
agent_intercept_library	File Path %ID/libspadesint.so Specifies the library to be preloaded into the agents (see Section 4.7.2). The string '`%ID`' is replaced with the package library dir specified at configure time (`/usr/local/lib/spades` by default).
enable_ipc_message_reception	Boolean on Whether to listen for SysV IPC messages, which are used to notify the communication server of process forking.
ipc_force_remove	Boolean off If this is specified, the IPC message queue is removed before the communication server is started. This should only be needed if the communication server crashes or exits abnormally.
max_secs_for_agent_think	Real 10 Gives the maximum time (as measured by the wall clock) between the time an agent is sent a message and it responds with a done thinking message. If this is less than 0, the wall clock time is not checked.
max_simtime_for_agent_think	Integer 1000 Gives the maximum time (as measured by the current agent timer) between the time an agent is sent a message and it responds with a done thinking message. If this is less than 0, there is no maximum.
agent_check_use_randomness	Boolean on Specifies whether to use randomness to decide whether to check up on an agent (see the *agent_check_gumbel_dist_A* and *agent_check_gumbel_dist_B* parameters) or not (see the *agent_check_threshold_sec* parameter).
agent_check_threshold_sec	Integer(≥ 0) 1 If *agent_check_use_randomness* is off, once the agent has been thinking this long (measured by the wall clock), a deep checkup has been done.
agent_check_gumbel_dist_A	Real 1.5 If *agent_check_use_randomness* is on, then a deep checkup is done randomly. This parameter and *agent_check_gumbel_dist_B* are parameters to a gumbel distribution which gives the probability of a deep checkup as a function of the wall clock time the agent has been thinking.

Continued on next page

Name	Type Description	Default
agent_check_gumbel_dist_B	Real(> 0) See the parameter *agent_check_gumbel_dist_A*.	1
agent_stdout_log_fpat	File Path This parameter describes what file to direct agent standard out to. The string '%A' is replaced by the agent number and '%D' by the *logfile_dir*.	%D/agent%A-stdout.log
agent_stderr_log_fpat	File Path This parameter describes what file to direct agent standard out to. The string '%A' is replaced by the agent number and '%D' by the *logfile_dir*.	%D/agent%A-stderr.log

6.1.2 Communication Server

These are the parameters which the communication server understands (in addition to the shared parameters of Section 6.1.1).

Name	Type Description	Default
engine_host	String The name of the machine which is running the simulation engine.	localhost
max_connect_reply_wait	Integer(≥ 0) How long (in seconds) to wait for a correct reply from the simulation engine in making the connection.	5
wait_sec	Integer(≥ 0) The time in seconds to pass into the `select` system call (combined with *wait_usec*).	5
wait_usec	Integer(≥ 0) The time in microseconds to pass into the `select` system call (combined with *wait_sec*).	0
no_message_timeout	Integer(≥ 0) If the communication server does not receive a message from the simulation engine in this number of seconds, the communication server exits.	20

6.1.3 Simulation Engine

Name	**Type** **Default** **Description**
port_bind_retries	Integer(≥ 0) 3 The number of times to retry binding the port to listen for communication server connections. This is especially useful if you are running multiple simulations in a row, as the TCP socket is not always immediately completely closed at the system level.
port_bind_sleep_sec	Integer(≥ 0) 2 The time (in seconds) to sleep between tries to bind the port to listen for communication server connections.
monitor_port	Integer(≥ 0) 12001 This is the port to which monitors can connect to the server.
accept_monitor_connections	Boolean on Whether to listen on the *monitor_port* for monitor connections.
monitor_interval	Integer (≥ 0) 33 The is the interval for when information is sent to the monitors.
use_monitor_log	Boolean on Controls whether *monitor_log_fn* is created with a log of all data that would be sent to a monitor.
monitor_log_fn	File Path %D/monitor.log The file to write out the data that would be sent to a monitor. The string '`%D`' is replaced by the *logfile_dir*.
text_event_log_fn	File Path %D/event_text.log The file to which to write a text (i.e. human readable) version of every event processed. The string '`%D`' is replaced by the *logfile_dir*. See *use_text_event_log*.
use_text_event_log	Boolean off Whether to write a text (i.e. human readable) log file of the events processed. This is not recommended for general use as the file can get quite large quickly, but this is useful for debugging. See also *text_event_log_fn*.
wait_sec	Integer(≥ 0) 3 The time in seconds to pass into the `select` system call (combined with *wait_usec*).

Continued on next page

Name	**Type** **Default** **Description**
wait_usec	Integer(≥ 0) 0 The time in microseconds to pass into the `select` system call (combined with *wait_sec*).
pause_mode_wait_sec	Integer(≥ 0) 1 Like *wait_sec*, but used when the simulation is paused.
pause_mode_wait_usec	Integer(≥ 0) 0 Like *wait_usec*, but used when the simulation is paused.
max_pause_mode_seconds	Real 90 The maximum number of seconds in pause mode before the simulation engine exits. Any non-positive value disables this, though it is not recommended that you disable this as the simulation engine process could then wait indefinitely.
timeout_for_event	Integer 60 The maximum amount of time (in seconds) to wait for an event to be processed. That is, if not event is processed in this amount of time (in non-pause mode), the simulation engine exits. Any non-positive value disables this, though it is not recommended that you disable this as the simulation engine process could then wait indefinitely.
use_migration	Boolean off Whether to use migration of agents during this simulation.
min_responses_before_migration	Integer(> 0) 10 Minimum number of responses from an agent on an machine before it is considered for migration.
max_agentq_trailing_time	Integer 50 Because of an agent's thinking time, it can be late in receiving a sensation. For example, if a sensation is to be delivered time x and the agent is already at time $x + 10$, it is 10 simulation time units behind. If an agent is more than this parameter value simulation time units behind, the sensation is converted into an inform (a non-thinking sensation). A negative value disables this feature.

Continued on next page

Name	Type Description	Default
max_timenot_trailing_time	Integer Like *max_agentq_trailing_time*, but for time notifies.	50
run_integrated_commserver	Boolean Whether to run a communication server which is integrated in the same process as the simulation engine. This improves efficiency slightly for a communication server on the same machine.	off
secs_for_socket_shutdown	Integer(≥ 0) When the simulation engine is shutting down, up to this amount of time is waited for all the write buffers of the sockets to be flushed.	5
limited_rate_default_st_per_sec	Real(≥ 0) In the limited rate run mode, the simulation engine tries to maintain a correspondence between wall clock time and simulation time advancement. The target is to run this many simulation steps every second.	1000.0
limited_rate_rebase_interval	Real When running in limited rate mode, the engine periodically resets the correspondence point between wall clock time and simulation time. This parameter determined how often (in seconds) this is done. A value less than or equal to 0 turns this feature off.	2.0

6.1.4 Sample World Model

These are the parameters which the sample world model understands (in addition to the shared and engine parameters of Sections 6.1.1 and 6.1.3).

Name	Type Description	Default
size	Vector(Real)[2](> 0) Specifies the X and Y size of the world (in m).	200 200
min_sense_latency	Integer(≥ 0) See *max_sense_latency*.	10
max_sense_latency	Integer(≥ 0) Along with *min_sense_latency*, provides a range for the sensation latency for each sensation.	20
min_sense_interval	Integer(≥ 0) See *max_sense_interval*.	50

Continued on next page

Name	**Type** **Default** **Description**
max_sense_interval	Integer(≥ 0) 70 Along with *min_sense_interval*, provides a range for the simulation time between sensations.
min_action_latency	Integer(≥ 0) 90 See *max_action_latency*.
max_action_latency	Integer(≥ 0) 100 Along with *min_action_latency*, provides a range for the action latency a each action. See also *random_act_latency*.
num_agents	Integer(≥ 1) 1 The number of agents to start.
sim_time_per_second	Real(> 0) 100 How many simulation time steps translate to one second. This is used to interpret other parameters.
stiction	Real(> 0) 3 This paramater affects the physics of the balls. It provides a maximum on the acceleration (simulating the maximum friction the wheels can have on the ground). Units are $\frac{m^2}{s}$.
speed_max	Real(> 0) 3 Gives the maximum speed of the balls (in $\frac{m}{s}$).
accel_rest	Real(> 0) 5 Gives the maximum acceleration of the balls when at rest (in $\frac{m^2}{2}$). The acceleration decreases as speed increases.
simulation_length	Integer 10000 The number of simulation time steps to run the simulation.
num_comm_servers_wanted	Integer 1 The number of communication servers to wait for before unpausing the simulation.
agent_speed	String fast This is passed on to the agents to specify how much of a busy wait delay they should have. The valid values are 'fast', 'medium', 'slow', and 'soccer.' The last speed has to do with experiments comparing SPADES to the SoccerServer (`http://sserver.sf.net`). It is in between medium and slow.

Continued on next page

Name	Type Description	Default
agent_types	Vector(String)[1–∞] These strings give the names of the agent types to start. They are used in order, repeating as necessary to get to *num_agents*.	type0 type1
random_act_latency	Boolean Whether or not to use randomness is the action latency. See Section 4.11 for important information about using randomness.	true
random_agent_placement	Boolean Whether or not to use randomness in the initial location of the agents. See Section 4.11 for important information about using randomness.	true
use_random_untimed_sensations	Boolean If this is on, some sensations are marked as untimed sensations. This is to simply to allow for testing of the untimed sensation mechanism.	false

6.2 Agent Database

The agent database describes the agent types (as discussed in Section 4.5).

The database is read in from an XML formatted file (the *agent_db_fn* parameter specifies the location of this file). A schema `agentdb.xsd` is provided in the SPADES distribution and installation. This section descibes the format in detail.

All elements should be in the namespace `http://spades-sim.sourceforge.net/agentdbxml.html`. In the agent database file provided in `sample_agent`, this is given the local name `adb`.

The top level element should be `agentdb`. It must have the attribute `version`. The contents of this attribute should be a decimal number giving the version of SPADES for which this file was written. This will hopefully allow some backward compatibility.

All `name` attributes must include only alphanumeric characters and `-` and _ with no white space.

The following child elements can occur any number of times and in any order:

`include` No attributes, no child elements, text content. Gives a (possibly relative) path to a fully formed agent database file. That file is read in before the rest of this file is processed.

`agent_type_external` Describes an agent that is to be started as an external process. The element has one required attribute `name` which gives the name to the type.

The child elements are:

`inputfd` (Optional) Gives the file descriptor which the agent will use to read information from the communication server. If this is omitted or a -1 is given, the value of *default_agent_input_fd* is used.

`outputfd` (Optional) Gives the file descriptor which the agent will use to send information to the communication server. If this is omitted or a -1 is given, the value of *default_agent_output_fd* is used.

`timer` (Optional) Gives is a timer type description string as discussed in Section 4.7.1. If ommitted, `default` is used.

`workingdir` (Optional) Gives the working directory for the agent. Before the executable is run, the current directory is changed to this. All relative directories are resolved from the location of the agent database file which is being processed.

`commandline` (Required) Gives the full command line for the agent. Note that the the format and parsing rules for this element are likely to change in future releases. The following parsing rules are used:

- The value must be brace ('{}') balanced.
- Whitespace separates arguments in the command line.
- If you need whitespace in an argument, use double-quotes around the argument. The double quote character will be removed.
- If you need a double quote character in your argument, put two double quotes next to each other. They will be reduced to a single double quote character.
- No other characters are special. Note in particular that this means that backslash ('\') has no special meaning and shell redirection can *not* be done (with the '>' and '<' characters).

Here are some examples of how command lines are parsed. First, the command line is given, then the arguments are given as they will be given to the agent, separated by commas and enclosed by single quotes.

```
Given command line      Parsed and separated arguments
foo -a b --code \\      'foo', '-a', 'b', '--code', '\\'
bar -a "Some Name"      'bar', '-a', 'Some Name'
baz -a "Some""Name"     'bar', '-a', 'Some"Name'
bif -a """SomeName"""   'bif', '-a', '"Some Name"'
```

`agent_type_integrated` Describes an agent that is to be loaded from a dynamic library and exist in the same process as the communication server. The element has one required attribute `name` which gives the name to the type.

The child elements are:

`lib_path` (Required) Gives the (possibly relative) path to the dynamic library that defines the agent.

`init_arg` (Optional) If specified, gives a string to be passed to the `initialize` method of the `IntegratedAgent` class for the agent.

`timer` (Optional) Gives is a timer type description string as discussed in Section 4.7.1. If ommitted, `default` is used. Note that some timer types are not acceptable for integrated agents (see Section 4.7.1).

`require_integrated_commserver` (Optional) A boolean which indicates whether the agent is required to run only in an integrated commserver, not an external commserver. The default is false.

`agent_type_placeholder` The element has one required attribute `name` which gives the name to the type. There are no child elements and no text. An agent of this type can not actually be started. It can be useful to give to the simulation engine (which does not actually start agents), and give a full type description to the communication server.

6.3 Length Prefixed I/O Format

In order to avoid any arbitrary restrictions of message length and format, SPADES uses a length prefixed input/output format for various messages. The length of the message is sent first, then the data. Note that SPADES may be sending data incrementally, so the entire message may not be available at once. Therefore, the reader must do some read buffering. The class `ReadBuffFD` provides this functionality, and, for the agents, it it further incorporated into the agent library which is provided as part of SPADES.

For every message, four bytes are sent (in network byte order) for the length of the message (which does not include the four bytes for the length). Then the data itself is sent.

6.4 External Agent Input/Output

This section gives the details of the agent input and output to and from and external agent. Here input refers to input that the agent receives from the communication server and output refers to information that the agents sends to the communication server.

First, all communication is done via pipes. When the agent starts up, the file descriptors given by the agent type (or the defaults *default_agent_input_fd* and *default_agent_output_fd*) should be used for reading/sending data from/to the communication server.

The length prefixed I/O format (see Section 6.3) is used for all communication to and from the agent.

6.4.1 Agent Input Format

This section describes the format of messages sent to the agent from the communication server. The first byte gives the type of message, and the remaining bytes are arguments and data for the

message. There is no space immediately after the first byte.

Throughout this section, the following variables are used:

- *time* means the simulation time (an integer). This is always followed by a single space.
- *data* means an arbitrary data string. Note that because of the use of the length prefixed format (Section 6.3), there are *no* restrictions on the format or length of the data. In particular, arbitrary binary data can be sent.
- *token* is a text string with no spaces.

The messages sent to the agent are:

- 'S*time time data*'

  This is a sensation to be given to the agent. It begins a thinking cycle.

  The first time is the simulation time the sensation was generated (also known as the send time) and the second is the time that the sensation is delivered to the agent (also known as the arrive time). If the parameter *send_agent_send_time* is off, -1 is always sent as the send time, and if the parameter *send_agent_arrive_time* is off, -1 is always sent as the arrive time. The data is an arbitrary data string generated by the world model. The agent can reply with act messages, and must finish with a done thinking message.

- 'I*time time data*'

  This is an inform message. The meaning this is the same as the `sense` above, except that it does not start a thinking cycle. The agent should not reply to this message.

  Note that informs can be explicitly generated by the world model, or "old" senses can be converted to informs. See Section 4.6 for details.

- 'T*time*'

  This is a time notify, an empty sensation. It does start a thinking cycle. The time is the time for which the time notify was requested, regardless of when the time notify is actually delivered. The agent can reply with act messages, and must finish with a done thinking message.

- 'O*time*'

  This is a time notify that does not start a thinking cycle. The agent should not reply to this message.

- 'X'

  This is an exit message that tells the agent to shutdown. No more messages will be sent to the agent and no reply should be sent. Note the shutdown process discussed in Section 5.5.

- '`D`*`data`*'

  After startup, this initialization data message is sent to the agent. On a normal startup, *`data`* will be empty. On an agent migration (see Section 5.3), the data returned by the old agent processes will be given to the new agent. An initialization done message should be sent once the initialization data has been processed and all other startup is complete.

- '`M`'

  This migration request message is used to tell the agent that it will be migrated. The agent must reply with a migration data message. See Section 5.3 for information about agent migration.

- '`K`*`time`*'

  This is a think time message that notifies the agent how much thinking time was used for the last thinking cycle. These are only sent if *send_agent_think_times* is on.

- '`E`*`token`*'

  This is an error message to the agent. The value of *`token`* gives the type of error.

  `no_token_on_line` The agent sent an empty message to the communication server.

  `act_when_not_thinking` The agent sent in act message while not in a thinking cycle.

  `rtn_when_not_thinking` The agent sent a request time notify while not in a thinking cycle.

  `bad_time_in_rtn` The time for a request time notify was not a properly formatted number.

  `done_thinking_when_not_thinking` The agent sent a done thinking message while not in a thinking cycle.

  `mig_data_when_not_mig` The agent sent a migration data message, but the communication server had not sent a migration request message.

  `bad_token` The first character identifying the message was not a valid message indicator.

  `init_done_when_not_init` The agent sent an initialization done message when it was not in initialization mode.

6.4.2 Agent Output Format

This section describes the format of messages sent to the communication server from the agent.

Throughout this section, the following variables are used:

- *`time`* means the simulation time (an integer). This should be followed by a single space.

- *data* means an arbitrary data string. Note that because of the use of the length prefixed format (Section 6.3), there are *no* restrictions on the format or length of the data. In particular, arbitrary binary data can be sent.

The format is primarily text based. The first byte gives the type of message, and the remaining bytes are arguments and data for the message. There should be no space added immediately after the first byte.

- 'A*data*'

  This is an action to be taken by the agent. The *data* is in arbitrary format, determined by the world model. Note that because of the length prefixed I/O (Section 6.3), their are no restrictions on the format or length of the data.

- 'R*time*'

  This is a request time notify. The *time* is the time for which the notification is requested. See Section 4.6 for details.

- 'D'

  This is the done thinking message which is used to conclude every thinking cycle. All actions and request time notifies need to be sent before this message.

- 'M*data*'

  This gives migration data for the agent. *data* is data to be passed to the agent on startup with an initialization data message. See Section 5.3 for information on agent migration.

- 'X'

  This is the exit message. The agent is notifying the communication server that it is exiting. This message can be sent while the agent is thinking or not. If the agent is in a thinking cycle, the exit message also functions as a done thinking message. No more information will be sent to or read from this agent.

- 'I'

  This is the initialization done message which the agent must send after starting up.

6.5 Integrated Agent Input/Output

This section describes the process of creating an integrated agent.

An integrated agent is a dynamic library that is loaded by SPADES. You create the library just as you would create any other dynamic library, so please see your system documentation for that

process. Note that SPADES using `libltdl` [2], and using `libtool` should integrate well with SPADES.

Your library must provide two things. First, a subclass of `IntegratedAgent` (whose interface will be described below) and a known entry point for creating an agent.

`IntegratedAgent` is a virtual base class. The methods generally correspond to messages that can be received by an external agent, so please see Section 6.4.1 for details on what the messages mean.

The methods are:

`initialize` This method will be called when the agent is started up. It takes a string which is the value of the initialization parameter specified by the agent type in the agent database.

`receiveTimeNotify` This method is called for the agent to receive a time notify message. The `ThinkingType` parameter specified whether this is a thinking or a non-thinking sensations (i.e. whether actions can be sent in response).

`receiveSense` This method is called for the agent to receive a sensation. The `ThinkingType` parameter determines whether this is a sense of inform message.

`receiveMigrationRequest` This method is called to request that the agent begin the migration process.

`receiveExit` This method is called to send the agent an exit message. The communication server will delete this memory after this message.

`receiveInitData` This is called on startup. If this is a migration, then this will be non-empty.

`receiveError` This is called to notify the agent of an error.

`receiveThinkTime` This is called for the agent to receive a think time message.

`getActions` This method give you an object of type `IntegrateAgentActions` which you can use to communicate with the communication server.

`setActionCallbacks` You should never need to call this method. It is used by SPADES to set the `IntegratedAgentActions` callback object (see below).

The class `IntegratedAgentActions` is what the agent uses to communicate with the communication server. The `getActions` method of `IntegratedAgent` give the agent an `IntegratedAgentActions` object on which to call methods. These calls parallel the messages that an external agent can send, so see Section 6.4.2 for more detail on the meanings of the messages.

The methods of `IntegratedAgentActions` are:

[2] `http://www.gnu.org/software/libtool/libtool.html`

act This is an act message with associated data.

requestTimeNotify This is a request time notify message. The parameter is the time for which the notification is requested.

doneThinking This is the done thinking message which ends every thinking cycle.

exit This method should be called when the agent requests an exit.

initDone This is the initialization done message which should be called in response to the initialization data message to the agent.

migrationData This is the migration data message. This method should be called in response to the migration request message.

The entry point for creating an agent should be a function with one of these names and signatures:

```
spades::IntegratedAgent* createAgent(void);
spades::IntegratedAgent* libname_LTX_createAgent(void);
```

where *libname* is the name of your library without the extension. For example, if you create a library called `myagent.so`, then *libname* should be `myagent`. In either case, this function should return a new dynamically allocated object which is a subtype of `IntegratedAgent`. Note that if you compile with a C++ compilier, you should but this function declaration inside of an

```
extern "C"
{
  your declaration here
}
```

to avoid name mangling on the function name.

6.6 Monitor Interface

Section 4.8 provides a general discussion of how monitors function. This section provides details of the interface.

The format of input received by the monitors is completely determined by the world model; SPADES adds nothing to this.

For receiving data from the monitors, SPADES uses the length prefixed data format (Section 6.3). The first character determines the type of message.

P The simulation is paused (i.e. the mode is changed to `SM_PausedMonitor`).

R The simulation is put into normal run mode ((i.e. the mode is changed to `SM_RunNormal`).

L The simulation is put into limited rate run mode ((i.e. the mode is changed to `SM_RunLimitedRate`). See Section 5.6 for details.

D This notifies the engine that this monitor is disconnecting. No further information will be sent to this monitor.

X This initiates a shutdown of the simulation.

W The indicates that this data should be passed onto the world model for processing. The initial 'W' is removed, and the remainder of the data is sent to the `parseMonitorMessage` method of `WorldModel`.

If an invalid character is sent, an error is printed, but it is otherwise ignored.

6.7 Agent Thinking Time

In order to determine the amount of computation used by an agent process, SPADES relies upon facilities provided by the Linux kernel. Overall, there is a two step process.

1. Askt the kernel how much computation was used.
2. Convert that computation time into simulation time and apply the actions.

6.7.1 Tracking for Jiffies Timer

The jiffies timer uses the facilities provided by the standard Linux kernel.

In the file '`/proc/`*`pid`*`/stat`', the Linux kernel provides a count of the number of "jiffies" which the process has used on the CPU. A jiffy is a fairly coarse measure of time, usually 10ms. The kernel actually reports four different jiffy counts:

`utime` The number of jiffies spent in user mode.

`stime` The number of jiffies spent in system mode.

`cutime` The number of jiffies spent in user mode by the process and its children.

`cstime` The number of jiffies spent in system mode by the process and its children.

The differences in `utime` and `stime` are added together to get a total jiffy count for the thinking by the agent. I have not been able to determine the circumstances under which `cutime` and `cstime` get updated, so SPADES does not currently use them.

Once this number of jiffies has been determined, it must be converted to simulation time. The following calculation is performed

- Let J be the number of jiffies.
- Let B be the bogoMIPS of the machine, as read from `'/proc/cpuinfo'`
- Let K be the value of the parameter supplied to the jiffies timer (see Section 4.7.1), which represents the number of kilo-instructions which translates to 1 simulation step.

The number of simulation steps is then (rounded to the nearest integer).

$$\frac{10JB}{K} \tag{6.1}$$

As you can see, the parameter of this linear transformation is a user controllable parameter. However, if different types of machines are to be used in the same simulation, some benchmarking would need to be done to determine what an equitable setting of these CPU time to simulation time parameters would be.

6.7.2 Tracking for Perfctr Timer

The Linux Performance Counters Driver[3] allow SPADES to track the number of instructions used by agent processes. This allows for much more accurate thinking time tracking and reproducible results.

However, currently, this requires the use of a patched kernel. SPADES currently works with version 2.5.1. The 2.6 series has changes at the user level, and SPADES will not currently work with any of these.

Also, the `ptrace` mechanism is used for SPADES to track the process correctly. Using `ptrace` can have subtle effects on the performance of the agent (see the `ptrace` documentation for details).

This timer takes a parameter giving the number of kilo-instructions which correspond to one simulation step.

6.7.3 Recorded File Format

As described in Section 4.7.1, SPADES provides the ability to record the thinking time used by in agent. This section describes the file format for that recording.

The file is in a binary format to save space and parsing time. All numbers are stored in network byte order.

The first 4 bytes of the file say how many bytes are used for each thinking time recording. Currently, SPADES only supports 2 bytes for each thinking time, so this is mainly for upward compatibility. After that, every 2 bytes represent a thinking latency for the agent.

Note that a perl script named `show_ttimes.pl` is installed by SPADES to show the contents of a thinking time file in a more human readable format. See Section 4.7.1 for details.

[3] `http://sourceforge.net/projects/perfctr/`

6.8 Algorithms

The event ordering and realization algorithms are also described in technical papers [Riley, 2003, Riley and Riley, 2003]. The text here is largely copied from those papers.

This section describes the simulation algorithm used by SPADES. This algorithm is a modification of a basic discrete event simulator.

```
repeat forever
    receive messages
    next_event = pending_event_queue.head
    while (no event will be received for time less than next_event.time)
        advanceWorldTime (next_event.time)
        pending_event_queue.remove(next_event)
        realize (next_event)
        next_event = pending_event_queue.head
```

Table 6.1: Inner loop for basic serial discrete event simulator

The inner loop of a basic serial discrete event simulator is shown in Table 6.1. The one primary difference is the *advanceWorldTime* call. This supports the one continuous aspect of the simulation, the world model. This function advances the simulated world to the time of the discrete event. The realization of the event causes changes in the world, and notably can cause other events to be enqueued into the `pending_events_queue`.

In order to insure that all events will be executed in causal order, the simulation environment must determine whether or not it is possible to receive a future event with a timestamp less than the next pending event. This so–called *time–management* function of parallel simulators is well studied, and there are a number of approaches that can be used [Chandy and Misra, 1981, Bryant, 1977, Mattern, 1993, Chandy and Misra, 1979, Chandy and Sherman, 1989, Lubachevsky, 1989, Steinmann, 1991, Nicol, 1993, Riley et al., 2000]. Much of the complexity of these approaches is due to the fact that typically a distributed simulation will manage private event lists for each process in the distributed environment. In other words, each process manages its own event list, and schedules events to and from this list independently from other processes (within the constraints imposed by the time management algorithms). For ease of implementation, we chose another well–known approach known as a *centralized event list*. In this approach, a single composite event list is managed by a *master* process, which is responsible for scheduling events and managing the event list for all other processors. Any process that needs to schedule a future event must notify the master process (the manager of the central event list) to get the event scheduled. This method is quite simplistic and easy to implement. This master process has complete knowledge at all times

of pending events, and can independently determine which pending events can be safely processed. A drawback of the central event list approach is that each process must notify the central scheduler that it has finished processing a prior event and is ready to process more events. The design of the agents using the sense–think–act paradigm mitigates this drawback, since all agents produce an action in response to sensed events, which serves as notification to the scheduler that the processing has completed. An obvious major drawback of this approach is efficiency and scalability, since a single process coordinates activities for all agents. This single coordination point could become a bottleneck and slow down the entire simulation. For our purposes, the total number of agents is reasonably small, and we haven't observed the centralized event list to be the primary performance bottleneck.

It is well understood that any conservative parallel discrete event simulator requires a non–zero *lookahead* property in order to achieve good parallel performance [Ferscha and Tripathi, 1996]. Simply stated, the *lookahead* value is a lower bound on the simulation time difference between the generation of an event on any processor A and the realization of that event on some other processor B. Larger lookahead values are known to give rise to better parallel performance. We now discuss the the lookahead algorithm of SPADES. We will first cover a simple version which covers some of the fundamental ideas and then describe the SPADES algorithm in full.

An explanation of the events that occur in the normal think-sense-act cycle of the agents must first be given. The nature of this cycle illustrated in Figure 6.2. First, an event is put into the queue to create a sensation. Typically, the realization of this event reads the state of the world and converts this to some set of information to be sent to the agent. This set is encapsulated in a sense event and put into the event queue. SPADES requires that the time between the create sense event and the sense event is at least some minimum sense latency, which is specified by the world model. When the sense event is realized, this set of information will be sent to the agent to begin the thinking process. Notice that the realization of a sense event does not require the reading of any of the current world state since the set of information is fixed at the time of the realization of the create sense event. Upon the receipt of the sensation, the communication server begins timing the agent's computation. When all of the agent's actions have been received by the communication server, the computation time taken by the agent to produce those actions is converted to simulation time. All the actions and the think latency are sent to the simulation engine (shown as "Act Sent" in Figure 6.2). Upon receipt, the simulation engine adds the action latency (determined by querying the world model) and puts an act event in the pending events queue. Similar to the minimum sense latency, there is a minimum action latency which SPADES requires between the sending time of an action and the act event time. The realization of that act event is what actually causes that agent's actions to affect the world.

Note that a single agent can be have multiple sense–think–act cycles in progress at once, as illustrated in Figure 6.1. For example, once an agent has sent its actions (the "Act Sent" point in Figure 6.2), it can receive its next sensation even though the time which the actions actually affect the world (the "Act Event" point in Figure 6.2) has not yet occurred. The only overlap SPADES forbids is the overlapping of two think phases.

Note also that all actions have an effect at a discrete time. Therefore there is no explicit support

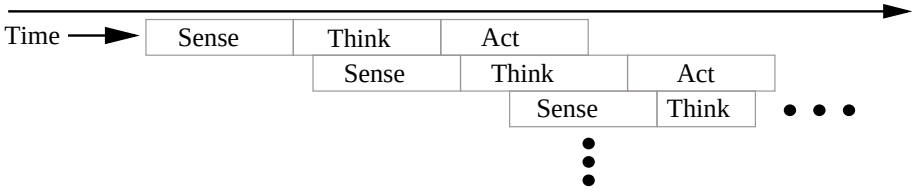

Figure 6.1: Example timeline for the sense-think-act loop of an agent to illustrate overlapping cycles

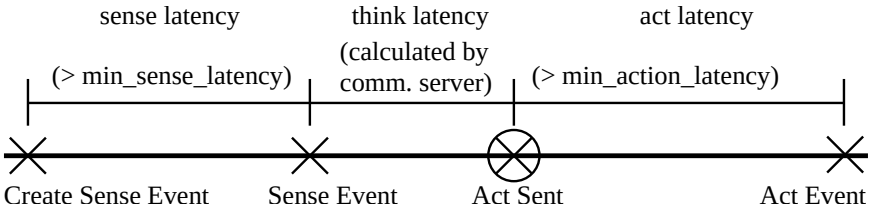

Figure 6.2: The events in the sense-think-act cycle of an agent. The "Act Sent" time is circled because unlike the other marks that represent events in the queue, "Act Sent" is just a message from the communication server to the engine and not an event in the event queue.

by SPADES for supporting the modeling of the interaction of parallel actions. For example, the actions of two simulated robots may be to start driving forward. It is the world model's job to recognize when these actions interact (such as in a collision) and respond appropriately. Similarly, communication among agents is handled as any other action. The world model is responsible for providing whatever restrictions on communication desired.

The sensation and action latencies provide a lookahead value for that agents and allows the agents to think in parallel. When a sense event is realized for agent 1, it cannot cause any event to be enqueued before the current time plus the minimum action latency. Therefore it is safe (at least when only considering agent 1) to realize all events up till that time without violating event ordering.

The quantity we call the "minimum agent time" determines the maximum safe time over all agents. The minimum agent time is the earliest time which an agent can cause an event which affects other agents or the world to be put into the queue. This is similar to the Lower Bound on Timestamp (LBTS) concept used in the simulation literature. The calculation of the minimum agent time is shown in Table 6.2. The agent status is either "thinking," which means that a sensation has been sent to the agent and a reply has not yet been received, or "waiting," which means that the agent is waiting to hear from the simulation engine. Besides initialization, the agent status will always be thinking or waiting. The current time of an agent is the time of the last communication with the agent (sensation sent or action received). The receipt of a message from a communication

```
calculateMinAgentTime()
    ∀i ∈ set_of_all_agents
        if (agent_i.status = Waiting) agent_time_i = ∞
        else agent_time_i = agent_i.currenttime + min_action_latency
    return min_i agent_time_i
```

Table 6.2: Code to determine the minimum time that an agent can affect the simulation.

```
repeat forever
    receive messages
    next_event = pending_event_queue.head
    min_agent_time = calculateMinAgentTime()
    while (next_event.time < min_agent_time)
        advanceWorldTime (next_event.time)
        pending_event_queue.remove(next_event)
        realizeEvent (next_event)
        next_event = pending_event_queue.head
        min_agent_time = calculateMinAgentTime()
```

Table 6.3: Code for parallel agent discrete event simulator for strict timestamp order.

server cannot cause the minimum agent time to decrease. However, the realization of an event can cause an increase or a decrease. Therefore, the minimum agent time must be recalculated after each event realization. However, this algorithm could be modified to be incremental so that the entire agent set does not have to be scanned each time.

Based on the calculation of the minimum agent time, we can now describe a simple version of parallel agent discrete event simulator, which is shown in Table 6.3. The value `min_agent_time` is used to determine whether any further events can appear before the time of the next event in the queue.

While this algorithm produces correct results (all events are realized in time stamp order) and achieves some parallelism, it does not achieve the maximum amount of possible parallelism. Figure 6.3 illustrates an example with two agents. When the sense event for agent 1 is realized, the minimum agent time becomes A. This allows the create sense event for agent 2 to be realized and the sense event for agent 2 to be enqueued. However, the sense event for agent 2 will not be realized until the response from agent 1 is received. However, as discussed above, the effect of the

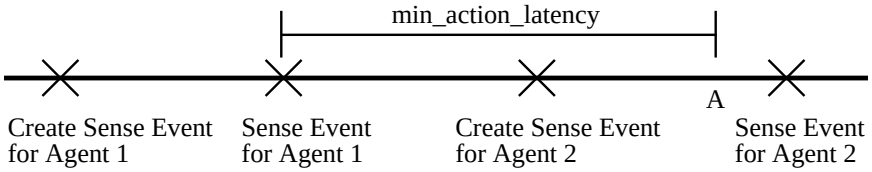

Figure 6.3: An example illustrating possible parallelism that the simple parallel agent algorithm fails to exploit.

realization of a sense event does not depend on the current state of the world. If agent 2 is currently waiting, there is no reason not to realize the sense event and allow both agents to be thinking simultaneously.

However, this allows the realization of events out of order; agent 1 can send an event which has a time less the time of the sense event for agent 2. Certain kinds of out of order realizations are acceptable (as the example illustrates). In particular, we need to verify that out of order events are not causally related. The key insight is that sensations received by agents are casually independent of sensations received by other agents. In order to state our correctness guarantees, we will define a new sub-class of events "fixed agent events" which have the following properties:

1. They do not depend on the current state of the world.
2. They affect only a single agent, possibly by sending a message to the agent.
3. Sense events and time notify events are both fixed agent events.
4. Fixed agent events are the only events which can cause the agent to start a thinking cycle, but they do *not* necessarily start a thinking cycle.

The correctness guarantees that SPADES provides are:

1. All events which are not fixed agent events are realized in time order.
2. All events which send sensations to the agents are fixed agent events.
3. The set of fixed agent events for a particular agent are realized in time order.

In order to achieve this, several new concepts are introduced. The first is the notion of the "minimum sensation time." This is the earliest time that a *new* sensation (i.e. fixed agent event) *other than a time notify* can be generated and enqueued. The current implementation of SPADES requires that the world model provide a minimum time between the create sense event and the sense event (see Figure 6.2), so the minimum sensation time is the current simulation time plus that time.

The time notifies are privileged events. They are handled specially because they affect no agent other than the one requesting the time notification. SPADES also allows time notifies to be

```
checkForReadyEvents(a: Agent)
    while (true)
        if (agent_a.status = thinking)
            return
        if (agent_a.pending_agent_events.empty())
            return
        next_event = agent_a.pending_agent_events.pop()
        realizeEvent(next_event)
```

```
enqueueAgentEvent(e:Event)
    a = e.agent
    agent_a.pending_agent_events.insert(e)
    checkForReadyEvents(a)
```

```
doneThinking(a: Agent, t:time)
    agent_a.currenttime = t
    checkForReadyEvents(a)
```

Table 6.4: Code for maintaining the per agent fixed agent event queues

requested an arbitrarily small time in the future, before even the minimum sensation time. This means that while an agent is thinking, the simulation engine cannot send any more fixed agent events to that agent without possibly causing a violation of correctness condition 3. However, if an agent is waiting (i.e. not thinking), then the first fixed agent event in the pending event queue can be sent as long as its time is before the minimum sensation time.

To insure proper event ordering, one queue of fixed agent events per agent is maintained. All fixed agent events enter this queue before being sent to the agent, and an event is put into the agent's queue only when the event's time is less than the minimum sensation time.

There are two primary functions dealing with the agent queue. First, enqueueAgentEvent puts a fixed agent event into the queue. The doneThinking function is called when an agent finishes its think cycle. Both functions call a third function checkForReadyEvents. Psuedo-code for these functions is shown in Table 6.4. Note that in checkForReadyEvents, the realization of an event can cause the agent status to change from waiting to thinking.

Using these functions, we describe in Table 6.5 the main loop that SPADES uses. This is a modification of the algorithm given in Table 6.3. The two key changes are that in the first while loop, fixed agent events are not realized, but are put in the agent queue instead. The second loop (the "foreach" loop) scans ahead in the event queue and moves all fixed agent events less that the minimum sensation time into the agent queues. Note that in both cases, moving events to the agent queue can cause the events to be realized (see Table 6.4).

```
repeat forever
    receive messages
    next_event = pending_event_queue.head
    min_agent_time = calculateMinAgentTime()
    while (next_event.time < min_agent_time)
        advanceWorldTime (next_event.time)
        pending_event_queue.remove(next_event)
        if (next_event is a fixed agent event)
            enqueueAgentEvent(next_event)
        else
            realizeEvent (next_event)
        next_event = pending_event_queue.head
        min_agent_time = calculateMinAgentTime()
    min_sense_time = current_time + min_sense_latency
    foreach e (pending_event_queue) /* in time order */
        if (e.time > min_sense_time)
            break
        if (e is a fixed agent event)
            pending_event_queue.remove(e)
            enqueueAgentEvent(e)
```

Table 6.5: Code for efficient parallel agent discrete event simulator as used by SPADES

Appendix A

GNU Free Documentation License

Version 1.1, March 2000

Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

A.1 Applicability and Definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers

to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A.2 Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation

in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

A.3 Copying in Quantity

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

A.4 Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- Use in the Title Page (and on the covers, if any) a title distinct from that of the Document,

and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

- List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- State on the Title page the name of the publisher of the Modified Version, as the publisher.
- Preserve all the copyright notices of the Document.
- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- Include an unaltered copy of this License.
- Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

- Do not retitle any existing section as “Endorsements” or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties – for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

A.5 Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled “History” in the various original documents, forming one section entitled “History”; likewise combine any sections entitled “Acknowledgements”, and any sections entitled “Dedications”. You must delete all sections entitled “Endorsements.”

A.6 Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

A.7 Aggregation With Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an “aggregate”, and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document’s Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

A.8 Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

A.9 Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

A.10 Future Revisions of This License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License ”or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled “GNU Free Documentation License”.

If you have no Invariant Sections, write “with no Invariant Sections” instead of saying which ones are invariant. If you have no Front-Cover Texts, write “no Front-Cover Texts” instead of “Front-Cover Texts being LIST”; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Bibliography

Scott D. Anderson. A simulation substrate for real-time planning. Technical Report 95-80, University of Massachusetts at Amherst Computer Science Department, 1995. (Ph.D. thesis). 2.3

Scott D. Anderson. Simulation of multiple time-pressured agents. In S. Andradóttir, K. J. Healy, D. H. Withers, and B. L. Nelson, editors, *Proceedings of the 1997 Winter Simulation Conference*, pages 397–404, 1997. 2.3

A. Birk, S. Coradeschi, and S. Tadokoro, editors. *RoboCup-2001: The Fifth RoboCup Competitions and Conferences*. Springer Verlag, Berlin, 2002. (to appear). 3.2.1

R. E. Bryant. Simulation of packet communications architecture computer systems. In *MIT-LCS-TR-188*, 1977. 6.8

K. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. In *IEEE Transactions on Software Engineering*, September 1979. 6.8

K. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. In *Communications of the ACM*, volume 24, November 1981. 6.8

K. M. Chandy and R. Sherman. The conditional event approach to distributed simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, March 1989. 6.8

Alois Ferscha and Satish Tripathi. Parallel and distributed simulation of discrete event systems. In A. Y. Zomaya, editor, *Parallel and Distributed Computing Handbook*, pages 1003 – 1041. McGraw-Hill, 1996. 6.8

Boris D Lubachevsky. Efficient distributed event-driven simulations of multiple-loop networks. *Communications of the Association for Computing Machinery*, 32(1):111–123, 1989. 6.8

F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. In *Journal of Parallel and Distributed Computing*, 1993. 6.8

David M Nicol. The cost of conservative synchronization in parallel discrete event simulations. *Journal of the Association for Computing Machinery*, 40(2):304–333, 1993. 6.8

George F. Riley, Richard M. Fujimoto, and Mostafa H. Ammar. Network aware time management and event distribution. In *Proceedings of the 14th Workshop on Parallel and Distributed Simulation*, May 2000. 6.8

Patrick Riley. MPADES: Middleware for parallel agent discrete event simulation. In Gal A. Kaminka, Pedro U. Lima, and Raul Rojas, editors, *RoboCup-2002: The Fifth RoboCup Competitions and Conferences*. Springer Verlag, Berlin, 2003. (to appear). 1.3, 4.10, 6.8

Patrick Riley and George Riley. SPADES — a distributed agent simulation environment with software-in-the-loop execution. In S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, editors, *Winter Simulation Conference Proceedings*, 2003. (to appear). 1.3, 2.3, 4.10, 4.11, 6.8

Patrick Riley, Peter Stone, and Manuela Veloso. Layered disclosure: Revealing agents' internals. In C. Castelfranchi and Y. Lespérance, editors, *Intelligent Agents VII. Agent Theories, Architectures, and Languages — 7th. International Workshop, ATAL-2000, Boston, MA, USA, July 7–9, 2000, Proceedings*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, Berlin, 2001. 5.1

Jeff Steinmann. Speedes: Synchronous parallel enviornment for emulation and discrete event simulation. *Advances in Parallel and Distributed Simulation, SCS Simulation Series*, 23:95–103, 1991. 6.8

Index