JACK Intelligent Agents®
JACK Sim Manual

Copyright

Copyright © 2012, Agent Oriented Software Pty. Ltd.

All rights reserved.

No part of this document may be reproduced, transferred, sold, or otherwise disposed of, without the written permission of the owner.

US Government Restricted Rights

The JACK™ Modules and relevant Software Material have been developed entirely at private expense and are accordingly provided with RESTRICTED RIGHTS. Use, duplication, or disclosure by Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of DFARS 252.227-7013 or subparagraph (c)(1) and (2) of the Commercial Computer Software Restricted Rights and 48 CFR 52.2270-19, as applicable.

Trademarks

All the trademarks mentioned in this document are the property of their respective owners.

Publisher Information

Agent Oriented Software Pty. Ltd.

P.O. Box 639,

Carlton South, Victoria, 3053

AUSTRALIA

Phone:	+61 3 9349 5055
Fax:	+61 3 9349 5088
Web:	http://www.agent-software.com

If you find any errors in this document or would like to suggest improvements, please let us know.

The JACK™ documentation set includes the following manuals and practicals:

Document	Description
Agent Manual	Describes the JACK programming language and infrastructure. JACK can be used to develop applications involving BDI agents.
Teams Manual	Describes the JACK Teams programming language extensions. JACK Teams can be used to develop applications that involve coordinated activity among teams of agents.
Development Environment Manual	Describes how to use the JACK Development Environment (JDE). The JDE is a graphical development environment that can be used to develop JACK agent and team-based applications.
JACOB Manual	Describes how to use JACOB. JACOB is an object modelling language that can be used for inter-process transport and object initialisation.
WebBot Manual	Describes how to use the JACK WebBot to develop JACK enabled web applications.
Design Tool Manual	Describes how to use the Design Tool to design and build an application within the JACK Development Environment.
Graphical Plan Editor Manual	Describes how to use the Graphical Plan Editor to develop graphical plans within the JACK Development Environment.
JACK Sim Manual	Describes how to use the JACK Sim framework for building and running repeatable agent simulations.
Tracing and Logging Manual	Describes the tracing and logging tools available with JACK.
Agent Practicals	A set of practicals designed to introduce the basic concepts involved in JACK programming.
Teams Practicals	A set of practicals designed to introduce the basic concepts involved in Teams programming.

Table of Contents

1 Introduction

Discrete event simulation is concerned with the modelling of behaviour in terms of entities which undergo discrete state transitions over time. There are various ways in which entity behaviours can be partitioned – these partitionings are known as *simulation world views*. Traditionally, three major world views have been distinguished, namely: activity, event and process (Kreutzer, 1986). JACK Intelligent Agents® (JACK) supports a new world view that we have called the *BDI world view*. In this world view, entity behaviours are encapsulated within agents and the JACK execution model is used to drive the simulation. The BDI world view provides a much richer and more intuitive interaction model than is afforded by the traditional world views and has proven to be especially useful for the simulation of distributed systems whose component entities exhibit complex internal behaviours and rich interaction models, both with each other and with their environment.

JACK and JACK Teams™ (Teams) provide concepts, programming constructs and run-time support to directly support the BDI world view, thereby making simulation model development using the BDI world view significantly easier. JACK also has constructs which facilitate the interfacing of JACK agents with existing applications and, since JACK is a superset of Java, the JACK programmer has access to all of the Java language and to existing Java classes and frameworks. JACK is neutral with respect to time management – three types of clock (real time, dilated and simulation) are supported. Every agent has a timer member which is by default set to the real time clock. Clocks can be shared between agents on either a machine (real time) or process (dilated or simulation) basis. Inter-machine sharing of real time clocks and inter-machine/inter-process sharing of dilated and simulation clocks is the responsibility of the application developer. In summary, using JACK for simulation model development offers the following advantages:

* support for a BDI world view

* support for an extensible time management infrastructure

* ability to code in Java when appropriate

* support for the interfacing to and encapsulation of existing applications in an agent-friendly manner.

JACK and Teams have been used to develop simulations in areas such as

* air traffic control

* manufacturing control

* virtual enterprise management

* mission management for teamed UAVs

* military command and control.

In addition, they have been used to augment the behaviour of entities in existing simulation environments, such as CAEN, OTB and STAGE.

These applications have been concerned with assessing the feasibility of particular strategies and tactics in the domains of interest – performance and sensitivity analyses were not conducted. Such analyses require that simulation runs are repeatable. This is not an issue with conventional simulation languages, as the simulation executes within a single thread of control within a single process and repeatability is guaranteed. However, when the simulation executes over multiple threads or multiple processes, repeatability needs to be explicitly addressed. JACK was designed so that within a single JACK process, agent execution is repeatable so long as there is no inter-agent communication. In practice this means that repeatability is constrained to applications consisting of a single agent.

In addition, while the BDI paradigm dictates how a simulation operates, it does not provide any indication as to how the underlying software architecture should be structured, other than that the particular application will be modelled using agents. Furthermore, our experience has shown that the creation and initialisation of agents and teams of agents can become a significant component of a simulation project as the developer is totally responsible for agent creation and initialisation. Typically this means that each application requires a bespoke main program that creates and initialises the required agents. JACK provides the JACOB™ Object Modelling Language (JACOB) for efficient object transport and initialisation. JACK also provides extensive support for the graphical display of application execution (graphical plan tracing, design diagram tracing, agent interaction diagrams).

JACK Sim™ (JACK Sim) consists of three major components:

- a model management infrastructure that provides a clear separation between scenario definition and scenario execution

- a time management infrastructure that provides guaranteed repeatability for multi-agent applications, regardless of whether the agents reside in multiple processes or on multiple machines.

- a visualisation infrastructure that facilitates the development of 2D visualisations of model execution.

Also JACK Sim presupposes that an actual application will conform to a reference model in which agent behaviour, embodiment and visualisation are explicitly represented.

2 Overview

JACK Sim is a framework for building and running repeatable agent based simulations. Simulations built with JACK Sim require agent behaviours to be implemented using either JACK or Teams. Also note that the JACOB™ Object Modeller (JACOB) is used for initialisation of data. It is assumed that the reader is familiar with JACK, Teams and JACOB.

In the BDI world view, agents respond to events issued by the environment, other agents or by themselves. The behaviours that are triggered by these events result in computations (that take no time) and delays(that consume time). Thus an action will in general be modelled as a computation and a delay, and will be triggered by an event. JACK provides programming constructs (events, plans, beliefsets, views and agents) to support this world view.

A JACK Sim application has a single clock that is maintained by a single agent of type `TimeSource`. The time source agent issues execution requests to one or more agents of type `TimeDispatcher` – there is one time dispatcher agent per process. Each time dispatcher agent allows the agents that are under its management to execute in a repeatable manner until all the agents are blocked. At this point, the time dispatcher agent sends a response to the time source agent indicating the earliest time at which one of its agents will become unblocked. When the responses from all time dispatcher agents have been received, the time source agent advances the clock to the earliest time that would unblock an agent. It then issues another round of execution requests and the cycle is repeated. The simulation stops when the clock can no longer be advanced. Clock management is transparent to developers – they are responsible only for the definition of the time management agents.

Agents within an application do not have to be managed by a time dispatcher agent, but if they are not, there may be implications with respect to repeatability. In order to be fully managed, an agent must

- be registered with the time management infrastructure
- implement either the `TimeManaged` or `TimeSyncManaged` marker interface
- have the `SimulationTiming` capability
- provide plans (and capabilities) to handle `RuntimeControl` events.

If these conditions are satisfied, then the infrastructure will manage the agent's life cycle. The life cycle consists of the following phases:

Phase	Description
Creation	Agent creation is managed on a per-process basis by `aos.jack.sim.run.Loader`. The initialisation for the agent is specified in a scenario definition file. The loader also registers the agent with the time management infrastructure.
Setup	Setup refers to initialisation involving other agents, such as role establishment.
Execution	Execution refers to the actual playout of agent behaviours for a particular scenario.

Table 2-1: Phases of the agent life cycle

If an agent implements one of the two marker interfaces and it has the `aos.jack.sim.time.SimulationTiming` capability, the agent's progress through the setup and execution phases will be controlled by events issued by its time dispatcher agent. In particular, the agent will receive `aos.jack.sim.time.RuntimeControl` events at the following times:

- SETUP - the commencement of the setup phase

- BEGIN - the beginning of the execution phase

- END - the end of the execution phase.

These points are distinguished by the event's `mode` member – it assumes a value of `SETUP`, `BEGIN` or `END` respectively.

As of JACK 5.5 a simulation can be automatically repeated a given number of times. An event containing the `BEGIN` or `END` mode field will also have the `data` field set. The 'begin' event's data will contain the number of the next simulation iteration. The 'end' event's data will contain the number of simulation iterations left to process.

Normally `SETUP` and `END` events can be ignored by the agent. However, if running multiple iterations of the simulation, they should be used to initialise and take down, respectively, the simulation entities so that each iteration can begin in a clean and reset state.

`BEGIN` events **must** always be handled or no execution will occur. Note that the protocol is synchronous – the sending of a `BEGIN` event to the next agent will **not** proceed until processing of the preceding `BEGIN` event has completed. Therefore, the plan that handles this event should eventually post an event (that will actually initiate execution within the agent) and simply exit.

In the case of agents that implement the `TimeSyncManaged` interface (as opposed to the `TimeManaged` interface), additional events with a mode of `STEP` will be received whenever the simulation clock is advanced. Time dispatcher agents implement the `TimeSyncManaged` interface; the intended use for user defined agents that implement this interface is to integrate computations performed by external processes into the JACK Sim repeatability framework.

If a simulation agent is registered with the infrastructure but does not implement one of the marker interfaces, it is loaded, but its execution is not managed by the infrastructure. If an agent is not registered with the infrastructure, then the developer is responsible for both its instantiation and execution. Note that in both situations, there may be implications in terms of repeatability.

A simulation is started by running the `Loader` class with the name of a scenario definition file as an argument. To begin a simulation that runs in a single process, type

```
aos.jack.sim.run.Loader <file-name>
```

where `<file-name>` is the name of the scenario definition file.

Note that if the simulation involves multiple processes, each process will have its own scenario definition file and will require a separate invocation of `aos.jack.sim.run.Loader`. Also, the process that contains the time source agent must be the last process to be started, as it initiates agent execution.

2.1 Example 1

As a first example, consider a variation on 'hello world' to illustrate what is involved in generating a minimal JACK Sim application. We assume that a normal JACK application has been developed that consists of three agents, `ralph1`, `ralph2` and `world`. `ralph1` and `ralph2` are of type `Speaker1` and `world` is of type `Speaker2`. `Speaker1` has a `Speak` plan which is triggered by a `Start1` event. The `Speak` plan consists of a continuous loop that sends an `Utterance` event to `world`. The plan waits for a response from world, waits a further 5 seconds and then repeats the process continuously. `Speaker2` has a `Respond` plan which is triggered by an `Utterance` event; it composes a response and sends it using `@reply`. Code for the example is presented in Appendix A. Design diagrams are shown below:
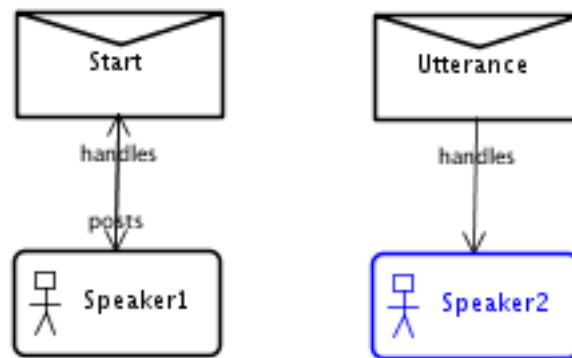
Figure 2-1: Agent/event diagrams for `Speaker1` and `Speaker2`
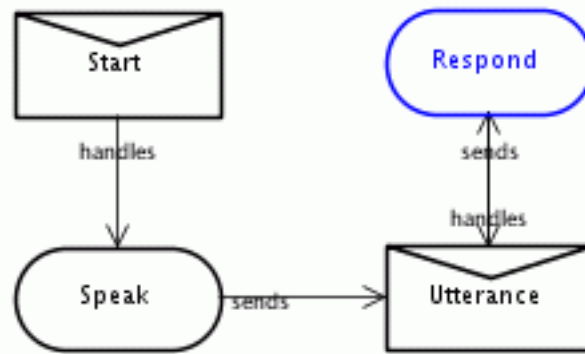
Figure 2-2: Plan/event diagrams for `Speaker1` and `Speaker2`

Conversion of this application into a minimal JACK Sim application involves the following steps:

1. In `Speaker1.agent`

    a) add the following statements

    ```
    import aos.jack.sim.time.RuntimeControl;
    import aos.jack.sim.time.SimulationTiming;

    #handles event RuntimeControl;
    #has capability SimulationTiming cap;
    #uses plan Prepare1;
    ```

    b) declare `Speaker1` as follows:

    ```
    public agent Speaker1 extends Agent
        implements aos.jack.sim.time.TimeManaged
    ```

2. In `Speaker2.agent`

    a) add the following statements

```
import aos.jack.sim.time.RuntimeControl;
import aos.jack.sim.time.SimulationTiming;

#handles event RuntimeControl;
#has capability SimulationTiming cap;
#uses plan Prepare2;
```

b) declare Speaker2 as follows:

```
public agent Speaker2 extends Agent
    implements aos.jack.sim.time.TimeManaged
```

3. Write Prepare1.plan:

```
package hello;
import aos.jack.sim.time.RuntimeControl;

public plan Prepare1 extends Plan {
  #handles event RuntimeControl rc;
  #posts event Start1 s1;

  static boolean relevant(RuntimeControl ev)
  {
    return  ev.mode == RuntimeControl.BEGIN;
  }

  #reasoning method body()
  {
    @post(s1.start());
  }
}
```

4. Write Prepare2.plan:

```
package hello;
import aos.jack.sim.time.RuntimeControl;

public plan Prepare2 extends Plan {
  #handles event RuntimeControl rc;

  static boolean relevant(RuntimeControl ev)
  {
    return  ev.mode == RuntimeControl.BEGIN;
  }

  #reasoning method body()
  {
     // do nothing
  }
}
```

5. Create a scenario definition file called scenario.def:

```
// We use the JACK Sim time management
<Include :dict "aos.jack.sim.time.Init__base" >

<TimeInit :date "Mon, Sep 29, 2003, 19:46:12.0"
          :dateformat "EEE, MMM d, yyyy, kk:mm:ss.S" >

// Declare a time dispatcher agent, which ensures that the time is
// not advanced while the application is busy.
<TimeDispatcherInit :name "timeDispatcher" >
```

```
// Instantiate and register the agents
<AgentInit :agent_type "hello.Speaker1" :name "ralph2" >
<AgentInit :agent_type "hello.Speaker1" :name "ralph1" >
<AgentInit :agent_type "hello.Speaker2" :name "world" >

// Declare a time source agent, which is responsible for advancing
// time in a synchronised manner with a time dispatcher agent.
<TimeSourceInit :name "timeSource"
                :dispatcher "timeDispatcher"
                :verbose 1
                :realtime :true
                :delay 0
>
```

The meanings of the various fields are discussed in the later chapters.

To execute the simulation, enter

```
java aos.jack.sim.run.Loader scenario.def
```

In the preceding example, all agents execute within a single process. However, this is not a requirement of JACK Sim – agents can be distributed across multiple processes (and multiple machines) in a JACK Sim application. As a simple illustration of this, suppose that for the preceding example, it becomes desirable to run the application agents in a separate process to the time source agent. Each process will require a portal name and a scenario definition file:

Process	**Portal**	**Filename**
Time source agent	jacksim0	scenario0.def
Application agents	jacksim1	scenario1.def

Table 2-2: Multi-process configuration

The content of the two scenario definition files are as follows:

1. `scenario0.def`

```
// We use the JACK Sim time management
<Include :dict "aos.jack.sim.time.Init__base" >

<TimeInit :date "Fri, Apr 6, 2001, 19:46:12.0"
          :dateformat "EEE, MMM d, yyyy, kk:mm:ss.S" >

// Declare a time dispatcher agent for this process
<TimeDispatcherInit :name "timeDispatcher0" >

// Declare a relay for the time dispatcher agent in the second
// process
<TimeRelayInit :name "timeDispatcher1@jacksim1" >

// Declare a time source agent, which is responsible for advancing
// time in a synchronised manner with a time dispatcher agent.
<TimeSourceInit :name "timeLord"
                :dispatcher "timeDispatcher0"
                :realtime :true
                :delay 0
>
```

2. `scenario1.def`

```
// We use the JACK Sim time management
<Include :dict "aos.jack.sim.time.Init__base" >

<TimeInit :date "Fri, Apr 6, 2001, 19:46:12.0"
          :dateformat "EEE, MMM d, yyyy, kk:mm:ss.S" >

// Declare a time dispatcher agent, which ensures that the time is
// not advanced while the application is busy.
<TimeDispatcherInit :name "timeDispatcher1" >
```

```
    // Instantiate and register the agents
    <AgentInit :agent_type "hello.Speaker1" :name "ralph2" >
    <AgentInit :agent_type "hello.Speaker1" :name "ralph1" >
    <AgentInit :agent_type "hello.Speaker2" :name "world" >
```

To run this version of the example, first start the process containing the application agents:

```
java aos.jack.sim.run.Loader "-dci.new:jacksim1=7821"
    scenario1.def
```

In a separate window, start the process containing the time source agent:

```
java aos.jack.sim.run.Loader "-dci.new:jacksim0=7820"
    "-dci.con:jacksim0->jacksim1=7821" scenario0.def
```

If an explanation of the dci arguments is required, refer to the *Agent* Manual.

Note that in this example, no modification of the application agent code was required, as all inter-agent communication is still within the same process. If the simulation agents were distributed across multiple processes, full agent names (*agent@portal*) would need to be used for communication between agents in the different processes.

3 A reference model

In developing agent based simulations, we have found it useful to view a simulation as consisting of the following models:

- Agent

- Equipment

- Environment.

The agent model in turn consists of behaviour and embodiment sub-models. Behaviour models encapsulate the reasoning that underpins agent activity. In performing this reasoning, an agent has knowledge of the the actions that it can perform. For example, a soldier agent might be be able to

- look

- walk

- run

- crawl

- crouch

- shoot

Depending on the application, the behaviour model might be implemented using a scientifically grounded cognitive architecture as in (Jarvis et al., 2005).

Embodiment models implement the the actions that are available to the behaviour model. These actions could be modelled at differing levels of fidelity depending on the application. For example, if a soldier was a component of a larger system and our interest lay in the modelling of the behaviour of that system and not that of the soldier, one could perhaps model the above actions as delays and ignore looking. However, in other situations we might need to model the looking process and the detailed dynamics of movement and shooting.

Having this distinction between the actions that an agent can perform and the realisation of those actions is extremely useful when agents are used to augment behaviours in existing simulation environments, such as OTB (OneSAF Testbed Baseline). In these situations, the primitive actions that the agent can perform correspond to the behaviours that are supported for its corresponding entity in the existing environment and realisation of agent function resides with the existing simulation environment. Another benefit that arises from this separation is that replacement of simulated functionality with actual functionality becomes straightforward. For example, one could evaluate alternative control strategies for a manufacturing cell using an embodiment model based on delays. Having determined a suitable control strategy, one could then replace the embodiment model with the (suitably interfaced) machines.

In general, the embodiment model will contain an execution manager that is responsible for the handling of requests from the behaviour model and the monitoring of the progress of those requests. With respect to the latter point, the emodiment model maintains an explicit representation of the execution state for each agent that is updated as the requested actions are executed. Note that action execution may be delegated to an external system such as OTB and that depending on the action requested, task decomposition may be required. If required, agent embodiment can be visualised using the JACK Sim visualisationinfrastructure. In this case, separate visualisation models would be provided that interact with the embodiment model by accessing the agent execution states. The process involved is discussed in the *Visualisation* chapter.

Equipment models provide physical models of any equipment managed by an agent. Control of equipment is mediated by the agent embodiment; the behaviour model does not directly control equipment. An item of equipment differs from an agent in that an item of equipment does not reason about its actions. For example, a soldier agent may have binoculars and a gun; both of these would normally be modelled as equipment.

Environment models provide models of the environment in which the agents are situated. In a military application, this could include both 'static' environment factors (such as terrain and landscape) and 'dynamic' environment factors, such as time of day, wind, rain etc. Perception of the environment is mediated by the agent embodiment; the behaviour model does not directly perceive the environment.

The figure below is a representation of a simulation in terms of these models. Its purpose is to make the distinctions between the models explicit, and in particular to stress that behaviour models are separate software components from both embodiment models and equipment models. The interaction paths are indicated by lines between the model classes. Note that the behaviour models only interact with embodiment models, which in turn interact with both equipment and environment models.
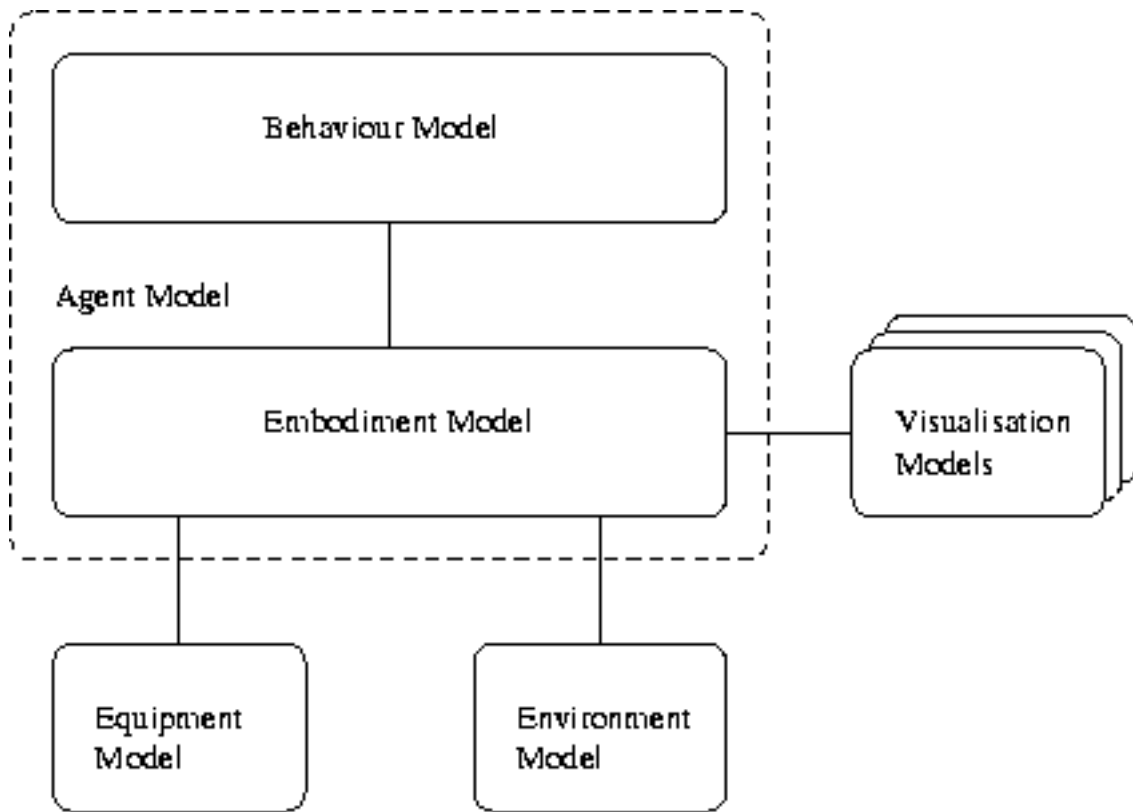
Figure 3-1: A reference model for agent-based simulation

Behaviour models will always be implemented in JACK. However, depending on the application, the remaining components may be implemented in JACK/Java or in an external modelling environment, such as C++, MATLAB or OTB. If the latter case applies, an interconnection layer is needed to provide the linkage between the behaviour model and the embodiment model. In terms of software, the interconnection layer is split into two parts with one part residing with the external modelling environment and the other residing with JACK. Technically, the layer manages the interconnection between JACK and the external environment. Conceptually, it mediates the interactions between embodiment functions and equipment and environment models. A macroscopic view of this architectural concept is illustrated in the figure below.
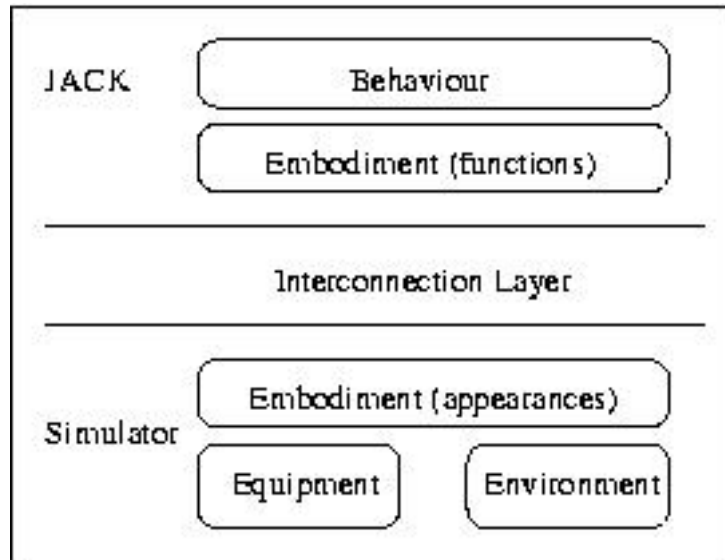
Figure 3-2: Macroscopic architecture for agent-augmented simulations

JACK Sim currently provides no support for the preceding reference model because of its simplicity and generality. In future releases, support may be be provided to assist in the interfacing to particular simulation environments.

4 Basic application development

A JACK Sim application consists of the following components:

- the infrastructure agents and classes that are required by the application

- user developed agents and classes

- a scenario definition that allows the user to initialise agent and object instances and to configure the application on a scenario by scenario basis.

Note: JACK Sim provides support for repeatability (through the time management infrastructure) and for 2D visualisation and animation (through the visualisation infrastructure). In this chapter, we focus on the development of applications which only use the time management infrastructure. The visualisation infrastructure is discussed in the *Visualisation* chapter of this manual.

4.1 Infrastructure agents

The following infrastructure agents can be created and initialised from entries in a scenario definition file:

4.1.1 Time management

4.1.1.1 `TimeSource`

The `TimeSource` agent is responsible for advancing the simulation time. The time control facility requires an application to have a **single** `TimeSource` agent per application, even if the application is distributed across multiple processes. If one is not present in an application, time will not be advanced.

4.1.1.2 `TimeDispatcher`

On receipt of a time update from the `TimeSource` agent, the `TimeDispatcher` agent enters a time control loop which enables all entities that have registered with the time management infrastructure and implement either the `TimeManaged` or `TimeSyncManaged` interfaces to execute until they are all blocked. Note that because of dependencies between entities, multiple passes through the loop may be required. When all entities are blocked, `TimeDispatcher` agent sends the earliest stop time to the `TimeSource` agent, which then advances the simulation time. The time control facility requires an application to have a **single** `TimeDispatcher` agent per process. In multiple process applications, `TimeRelayInit` objects can be used to add the dispatcher agent for a remote process to the list of entities managed by the local dispatcher agent. In this way, repeatability is guaranteed even with multi-process simulations.

4.2 User developed agents and classes

Agent behaviours are programmed using the *JACK Agent Language* – this is the subject of the next chapter. As with infrastructure agents, user-defined agents can be created and initialised from entries in a scenario definition file.

4.3 Scenario definitions

Scenario definitions contain entries that

- specify JACOB dictionaries and scenario definition files that are to be included in the scenario definition

- group related entries into sections

- specify the agents and classes that are required for a scenario and their initialisation for that scenario

- initialise global data.

Entries are specified using the JACOB object modelling language and are contained in a scenario definition file. Each process in an application must have its own scenario definition file. However as noted above, this file may include other definition files. The scenario definition file for a process is processed by the JACK Sim loader:

```
java aos.jack.sim.run.Loader <file>
```

The loader creates and initialises all the agents and objects specified in the scenario definition file and if required, registers them with the time management infrastructure.

4.3.1 Dictionary and scenario inclusion

Dictionary and scenario inclusion within a scenario definition file is achieved through `Include` objects. An `Include` object has a `dict` attribute and a `file` attribute, both of type `String`. The `dict` attribute identifies an additional JACOB dictionary to make available to the scenario loading. Typically, this dictionary will contain the agent and object initialisation classes that appear in the subsequent agent and object initialisation entries. The `file` attribute identifies a scenario definition file for inclusion in the scenario definition. That file may include further dictionaries and scenario definition files.

The following is an example of using an `Include` object in a scenario definition.

```
<Include :dict "cell.machines.Init__defs">
```

In the example, the scenario definition dictionary is first extended by adding the classes contained in `class cell.machines.Init__defs`. The class definitions for this dictionary were defined in the JACOB file `cell/machines/defs.api`. Once the above dictionary appears in a scenario definition file, its classes can then be used for agent and object creation and initialisation. If the resulting scenario definition file was called `cell/meterbox.def` it could be included into another scenario definition file via the following entry:

```
<Include :file "cell.meterbox.def">
```

JACK Sim has a standard simulation definitions file named `aos/jack/sim/standard.def`. Thus, a scenario definition file will often have the following line

```
<Include :file "aos/jack/sim/standard.def" >
```

at the beginning of the file. `standard.def` includes the following dictionaries:

Dictionary	Description
`aos.jack.sim.visual.awt.Init__awt`	Standard drawable components
`aos.jack.sim.visual.Init__visual`	Standard visual model extension
`aos.jack.sim.time.Init__base`	Standard time manager

Table 4-1: Dictionaries included by `standard.def`

If visualisation is not required in a particular simulation, one may choose to include only the standard time manager. This can be achieved with the inclusion of the following line in the scenario definition file:

```
<Include :dict "aos.jack.sim.time.Init__base" >
```

In addition, if the application uses JACK Teams, the following dictionaries must be included in the scenario.def file.

```
<Include :dict "aos.jack.sim.team.Init__base" >
<Include :dict "aos.team.init.Init__teammap" >
```

4.3.2 Entry grouping

Grouping of entries in a scenario definition file can be achieved with `Folder` objects. A `Folder` object has two attributes a `name` attribute of type `String` and an `items` attribute.

4.3.3 Agent Initialisation

The agents required by a simulation are usually created from entries in the scenario definition file which contain agent initialisation objects. These objects are either of type `AgentInit` or are subtyped from `AgentInit`.

The `AgentInit` class can be used directly for creating an agent which requires no further initialisation at construction time. It has the three attributes `agent_type`, `name` and `is_persistent` (described later). The class has an `initialise(Loader loader)` method, which creates an agent of the given type and name, and adds this to the `Loader.entities` `Hashtable`. The agent is created by calling `Loader.newAgent()` (which in turn calls `aos.jack.Kernel.createAgent()`) with the `AgentInit` object as initial data.

The following entry would enable the loader to create an agent called `hirata` of type `cell.machines.Robot` and add it to its list of entities to be managed by the time management infrastructure. By default, this entity would not keep its running state between simulation runs. Note that no additional initialisation is performed.

```
<AgentInit :name "hirata" :type "cell.machines.Robot" >
```

In many situations, the agent that we wish to create will require data fields other than the name to be initialised. In this case, we need to create a subclass of `AgentInit` that defines the required fields, which can be of any type. In these situations, the type of the agent can be specified in the class definition. If this is done, the type does not need to be specified in the object definition.

As an example, suppose that we wanted to create an instance of a Hirata robot and set its cycle time to five seconds. The following dictionary definition would suffice:

```
<Class :name "HirataInit"
    :extends "aos.jack.sim.run.AgentInit"
    :fields (
        <Field :name "name" :type :string :inherited :true >
        <Field :name "agent_type" :type :string :inherited :true
                :value "cell.machines.Hirata"
        >
        <Field :name "cycleTime" :type :long >
    )
>
```

The example defines an initialisation object type named `HirataInit` that maps to agent type `cell.machines.Hirata`, and includes an attribute named `cycleTime`. This definition could then be included in a file such as `cell/machines/defs.api`.

The initialisation object could then be used in a scenario definition file:

```
<HirataInit :name "hirata" :cycleTime 5 >
```

Note: When `AgentInit` is extended by a new initialisation type, then the `name` and `agent_type` attributes are marked as `inherited`. The `agent_type` attribute is further assigned an initialisation value, which is the (default) agent type to create when the new initialisation type is used in a scenario definition.

If no agent name is provided in the initialisation object, then no agent will actually be created at the time that the scenario file is read. This may be useful to provide an initialisation template that can be used within code that creates agents on the fly.

4.3.4 Infrastructure agent initialisation

The following initialisation objects are available to enable infrastructure agents to be initialised:

4.3.4.1 `TimeSourceInit`

A `TimeSourceInit` object in the scenario definition results in the creation of a `TimeSource` agent. The initialisation attributes are:

Attribute	Type	Description	Default
name	String	The agent name for the `TimeSource` agent. This must be a unique name on that portal.	
dispatcher	String	The agent name of the root `TimeDispatcher` that will receive clock advance messages.	
console	TimeConsoleInit	An object that defines the appearance of the time console.	
verbose	int	The reporting level for time control loop logging. The allowed values are 0 (quiet), 1 (time only), or 2 (time and delays).	1

Attribute	Type	Description	Default
realtime	boolean	Indicates whether the simulation time advancement should (if possible) be synchronised with real time, or as fast as possible. The former is further qualified by the 'realtimefactor' attribute below and the latter is qualified by the 'delay' attribute below.	true
realtimefactor	double	This specifies the relative speed by which to advance time when synchronised with real time. Note that a time advance is always to the next time being waited for by some agent, and any real time relative synchronisation is achieved by holding back that time advance until an appropriate amount of real time has passed.	1
delay	long	The realtime delay rate, in milliseconds, to use for a non realtime simulation. It specifies how many milliseconds to delay in between time advances.	0
keep_alive	boolean	If keep_alive is set to true and no agent is active and no agent is waiting for time to advance, then the current simulation run will still remain active and wait for something to occur. If it is set to false, then the current simulation run will finish. This usually means the entire process will exit.	true

Table 4-2: Initialisation attributes for a `TimeSourceInit` object

4.3.4.2 `TimeDispatcherInit`

A `TimeDispatcherInit` object in the scenario definition results in the creation of a
`TimeDispatcher` agent. The initialisation attributes are:

Attribute	Type	Description	Default
name	String	The agent name for the `TimeDispatcher` agent.	
loop_edge	int	The number of repetitions of the time control loop that are allowed in any `TimeSource` agent/`TimeDispatcher` agent exchange before warning messages are displayed.	10
loop_limit	int	The number of repetitions of the time control loop that are allowed in any given `TimeSource` agent/`TimeDispatcher` exchange. If the limit is reached, the dispatcher either exits the application or if the scenario is running under the control of a `Monitor`, it arranges for the current scenario iteration to be terminated.	1000
exit_on_idle	boolean	If this flag is true, it specifies that the time dispatcher agent should invoke `System.exit(0)` on a `RuntimeControl(END)` message, after having dealt with the message. In particular, it is intended for a multi-process set up, to terminate all processes at the end of the simulation. If the scenario is running under the control of a `Monitor`, this flag will only have an effect after all scenario iterations have terminated.	true

Table 4-3: Initialisation attributes for a `TimeDispatcherInit` object

4.3.5 Infrastructure object initialisation

A number of initialisation objects are provided to facilitate particular infrastructure
interactions. These objects do **not** create agents. The available objects are:

4.3.5.1 `TimeInit`

A `TimeInit` object is used to set the initial simulation time, and it can have the following
attributes:

Attribute	Type	Description
date	String	Specifies the date and time. The format of this string is defined by the `dateformat` attribute.
dateformat	String	The format string to be used for specifying the `date` attribute. Refer to `java.text.SimpleDateFormat` for the available formats – the default is `"EEE, MMM d, yyyy, kk:mm:ss.S"`. If the abbreviated year pattern is used in `dateformat`, a given year will be interpreted to be within 80 years before and 20 years after the date the time is set.
value	long	Sets the time in terms of the number of milliseconds since midnight, 1 Jan 1970. This attribute overrides `date` and provides an alternative way to set time.

Table 4-4: Initialisation attributes for a `TimeInit` object

The following is an example to set the initial simulation time in the default format:

```
<TimeInit :date "Fri, Apr 6, 2001, 19:46:12.0" >
```

Note: The simulation time is set by the initialisation method of the `TimeInit` object, and takes effect immediately on being processed. If there are multiple instances of a `TimeInit` object in the scenario definition file, the last one will override the earlier instances.

The `TimeInit` class can also be used within the simulation to convert the simulation time into a `String` according to the `dateformat` attribute. The relevant method is

```
public static String TimeInit.toString(long time);
```

Note: If a `TimeInit` object has only the `dateformat` attribute set, the actual time is not changed.

4.3.5.2 `TimeRelayInit`

A `TimeRelayInit` object is used to link dispatcher agents in a multi-process application. The resulting linkage structure should take the form of a multi-way tree. The following attribute can be set:

Attribute	Type	Description
name	String	The name of the 'destination' dispatcher agent. The full agent name is required, i.e. *agent@portal*.

Table 4-5: Initialisation attributes for a `TimeRelayInit` object

4.3.5.3 `TimeConsoleInit`

A `TimeConsoleInit` object is used to specify the attributes of a time console. The console enables the user to control the progress of time within a simulation.

Note: In the current version of JACK, running multiple iterations of a simulation is not supported when using an interactive time console.
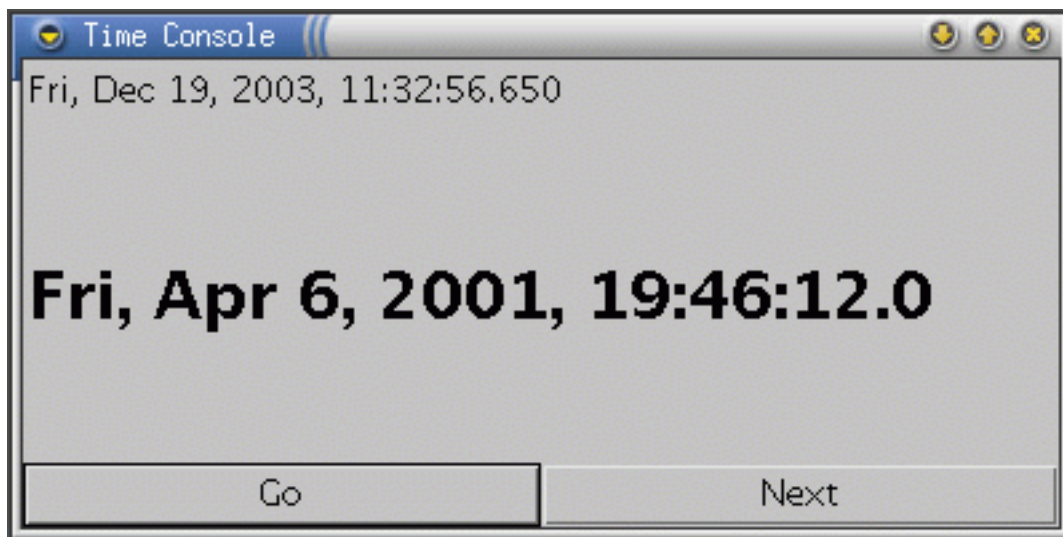
It has the following appearance:

Figure 4-1: Time console appearance

The following attributes can be set:

Attribute	Type	Description	Default
title	String	The title for the console.	"Time Console"
font	String	The font to be used for text displayed in the console.	"arial-bold-24"
x	int	The horizontal coordinate for the location of the console on the screen.	0
y	int	The vertical coordinate for the location of the console on the screen.	0
width	int	The width of the console.	200
height	int	The height of the console.	90
interval	long	This is the interval (in milliseconds) between updates of the time console window attributes (especially its time presentations).	1000
stopped_at_start	boolean	This specifies whether or not the time console should should start in 'stopped' mode.	false
exit_on_close	boolean	This specifies whether or not the time console should invoke System.exit(0) when the window is closed.	true
enabled	boolean	This specifies whether or not the time console should be used. This makes it possible to set up a useful configuration (especially location and font) to have available for intermittent use of the time console.	true

Table 4-6: Initialisation attributes for a `TimeConsoleInit` object

If a console is required for an application, a separate `TimeConsoleInit` entry is not created in the scenario definition. Rather, the `TimeConsoleInit` object definition is inserted inline in the `TimeSourceInit` object definition, as shown below:

```
<TimeSourceInit
    <AgentInit
        :name  "time source"
    >
    :keep_alive  :false
    :realtime  :false
    :realtimefactor  50.0
    :verbose  0
    :dispatcher  "time dispatcher"
    :console
        <TimeConsoleInit
            :font  "arial-bold-14"
            :x  10
            :y  500
            :width  300
            :height  120
            :interval  60000
            :stopped_at_start  :true
        >
>
```

4.3.5.4 `SimAgent`

The `aos.jack.sim.run.SimAgent` class extends the `aos.jack.jak.agent.Agent` class. It should be used as a base class for simulation entities that may persist over a number of simulation iterations.

It contains two state flags that can be changed as required.

Name	Type	Default	Description
readyAtStart	boolean	true	Whether the object should be automatically activated at the start of a run
stopAtEnd	boolean	true	Whether any active tasks for this agent should be forcibly stopped when the run is (forcibly) finished

These are the extra methods available through a `SimAgent`.

```
public boolean isReadyAtStart();

public void setReadyAtStart(boolean value);

public boolean isStopAtEnd();

public void setStopAtEnd(boolean value)
```

If the `readyAtStart` attribute is false, then the agent will not be registered with the loader when it is created via the scenario definition file. It will be necessary to use one of the `Loader` methods such as `deploy()` to register it and thus place it under the control of JACK Sim.

At the end of each simulation iteration, any registered entity for which this attribute is false, will be deregistered from the JACK Sim loader. This means it will not be able to run until it is registered with the loader again.

If this attribute is true, then at the start of the each iteration, the entity will be immediately able to run and will receive the usual `RuntimeControl.SETUP` and `RuntimeControl.BEGIN` events at the initial time.

Note: Using *Loader.deploy()* to create and/or register an agent will overridethis attribute and set it to false. For this reason `Loader.deploy()` can only be used with agents derived from `SimAgent`.

If the `stopAtEnd` attribute is false, then at the end of a simulation iteration, nothing special happens to it. It simply continues to run as it was already doing. However, if this attribute is true (the default), then at the end of an iteration, any tasks that are still running within the entity will be immediately terminated as an automated part of the clean-up phase between simulation iterations.

Note: Because this attribute is true by default, simulation entities will be forcibly stopped when each iteration ends unless they arrange otherwise. Any agent registered with the `Loader` which is not derived from `SimAgent` will also be forcibly stopped at the end of each iteration.

For example:

```
<AgentInit
    :name "SubControlAgent"
    :agent_type "my.models.SubmarineController"
    :stopAtEnd :false
>
```

If `agent_type` is not ultimately derived from `aos.jack.sim.run.SimAgent`, then it will cause an error when the scenario file is loaded.

4.3.5.5 `MonitorInit`

A `MonitorInit` object is used to control the running of a scenario so that it automatically repeats for a given number of iterations. This is useful for running Monte Carlo style randomised simulations. The following attributes can be set:

Attribute	Type	Description
type	String	The name of a fully qualified Java type that extends `aos.jack.sim.run.Monitor`. It will be automatically created and associated with the current `Loader` object. It will be initialised using the other (following) attributes.
iterations	int	The number of iterations to perform of the current scenario.
seed	long	A seed value that will be passed to initialise a random number generator. The way random number seeding is handled can be customised by overriding methods in the `Monitor` object.

Within a scenario definition file, the following will cause a `Monitor` object of specific type `aos.myproject.IterationController` to be created and associated with the current loader. The number of scenario iterations to be run will be 3 and the created object will have its `initRandomSeed(long value)` method immediately called with the value of 1234.

```
<MonitorInit
    :type "aos.myproject.IterationController"
    :iterations 3
    :seed 1234
>
```

Note: It only makes sense to have one `Monitor` object per scenario definition file. After processing the first `MonitorInit`, subsequent `MonitorInit` objects will be silently ignored.

Table 4-7: Initialisation attributes for a `TimeRelayInit` object

4.3.6 Global data initialisation

The `ConfigurationBase` class is available as a base class for creating a global configuration class for an application. Objects of the derived class can then be used in a scenario definition to provide scenario specific initialisation for the configuration. The derived class is defined in a JACOB `.api` file and the resulting dictionary must be included in the scenario before any reference is made to the derived class. A definition for a derived class is shown below:

```
<Class :name "Global"
    :extends "aos.jack.sim.run.ConfigurationBase"
    :fields (
        <Field :name "buffer1_in_use" :type :bool :value "true" >
        <Field :name "buffer2_in_use" :type :bool :value "false" >
    )
>
```

When the `ConfigurationBase` extension is used in a scenario definition, it will be installed as a global data object with default name `conf`. Thus, if the file that contained the above definition was named `models/global.api`, then the scenario definition could include the following declaration:

```
<Include :dict "models.Init__global" >
<Global :buffer1_enabled :true  :buffer2_enabled :true >
```

This would then be accessible in a plan, for example:

```
import models.Global;
plan ... {
    #uses data Global conf;
    ...
        if (conf.buffer1_enabled && conf.buffer2_enabled) ...
    ...
}
```

The name of the configuration object is `conf` by default. This name is a `String` member of the `ConfigurationBase` class, and thus an alternative name can be specified in the scenario definition by referencing the superclass field directly:

```
<Include :dict "models/global.api">
<Global <ConfigurationBase :name "cell">
        :buffer1_enabled :true :buffer2_enabled :true >
```

Alternatively, the default name can be changed by means of the `inherited` attribute. The following example illustrates how this may be done.

```
<Class :name "Global"
    :extends "aos.jack.sim.run.ConfigurationBase"
    :fields (
       <Field :name "name" :type :string :inherited :true :value "cell">
       <Field :name "buffer1_enabled" :type :bool :value "true">
       <Field :name "buffer2_enabled" :type :bool :value "false">
    )
>
```

4.4 Creating Agents Programatically (On The Fly)

In order to create an agent on the fly, you simply need to call a method on the current loader object with initialisation data for the new agent.

There is a static method within `Loader` to get the current loader.

```
public static aos.jack.sim.run.Loader getLoader()
```

The `Loader` object has several methods which can be used to create agents, initialise them and register them with the loader.

```
public void addAgent(String name, String type, InitialData data)
```

This method will create an agent with the given name and Java type. The agent will be initialised using the given initialisation template, which will usually be defined in the scenario

file. This method will throw an exception if an agent with that name already exists or is already registered with the loader.

If you are running a scenario which involves a number of iterations in which the agents are never destroyed but simply reinitialised, the following method will prove useful.

```
public void readyAgent(String name, String type, InitialData data)
```

This works in the same way as `addAgent()` except that it will not bother to create the agent if it already exists. It will, however, throw an `Error` if the agent is currently active or registered with the loader.

Both of these methods will leave the agent in an initialised but blocked state. The agent will not run or respond to events until it is unblocked. If there is any other initialisation to be done, it is safe to do it while the agent is blocked unless it involves event handling (which would require the agent to be running).

The very last thing both of the above methods do before returning is to call the agent's `initialize(InitialData data)` method.

To manually unblock an agent so that it is ready to receive events, you can use one of the following `Loader` methods:

```
public void unblockAgent(Agent agent)
```

A very useful method is:

```
public Agent deploy(String name, String type, InitialData data)
```

The `deploy()` method takes care of all the internal housekeeping for adding a new entity to the simulation at an arbitrary time. It creates the agent if required and registers it with the loader using `readyAgent()` with the given parameters. It then unblocks the agent and organises for the required `RuntimeControl` events to be sent to it so it can initialise itself correctly and join the simulation in the same way that entities defined in the scenario definition file do.

Note: Using the `deploy()` method will unset the `readyAtStart` flag on the returned agent so it must be derived from `SimAgent`. If you don't want this then you will need to manually call `agent.setReadyAtStart(true)` after the call to `deploy()`. In that case, the entity will be active immediately when the next iteration begins.

4.4.1 Programatically Created Agents in a Multiple Iteration Scenario

If the scenario will be performing multiple iterations, you may want to clean up the agents that were created on the fly. They could be completely destroyed and later created again as needed. If it is simple enough to reinitialise the agent rather than recreate it, all that is required is to

deregister the agent from the loader when an iteration finishes. This will stop it from being executable.

JACK Sim provides a simple way for this to be done automatically. If your agent class extends `aos.jack.sim.run.SimAgent` instead of a regular agent, it will have two additional methods:

```
public void setReadyAtStart(boolean value)

public boolean isReadyAtStart()
```

If the agent has this attribute set to "false" then when a scenario iteration ends, it will be automatically deregistered from the loader and thus be disabled until it is registered again (and unblocked).

If you arrange for the agent not to be deregistered or destroyed when the current scenario iteration ends, then as soon as the next iteration starts, it will receive a `RuntimeControl(BEGIN)` event and start executing immediately, as will all other registered agents.

Note: Using the `Loader.deploy()` method to start the agent will automatically set this attribute to "false".

The default state for any `SimAgent` is that when a simulation ends, any tasks it is still executing will be forcibly terminated. If your new agent should continue to execute undisturbed between runs, you will need to call `agent.setStopAtEnd(false)` to change alter this behaviour.

```
public void setStopAtEnd(boolean value)

public boolean isStopAtEnd()
```

Alternatively, the `stopAtEnd` attribute could be set to "false" in the `AgentInit` structure in the scenario definition file.

4.5 Early Termination of the Simulation Run

The `Monitor` object (described above) provides a method that can be called to trigger an early end to the current simulation iteration.

```
public void
    aos.jack.sim.run.Monitor.setForcedQuit(boolean val)
```

There is also a static convenience method in the Loader class.

```
public static void
    aos.jack.sim.run.Loader.StopCurrentRun()
```

These methods will cause JACK Sim to start the process of sending END events to each agent. On receiving these events, the agents must clean up whatever they have been doing and get ready for the next run.

Before starting the next run, any outstanding plans that are still being executed by agents that have not taken any steps to prevent it (by unsetting the `stopAtEnd` attribute, see above), will be forcefully terminated.

Any entities that have the `readyAtStart` attribute set to "false" (see above), will be deregistered from the loader and thus not automatically receive the `SETUP` and `BEGIN` events when the next iteration begins.

4.6 Randomisation of Simulation Runs and Repeatability

Adding randomisation to a simulation is simply a matter of calling methods that will return random data according to whatever criteria you desire.

To provide repeatability, the seed provided in the `Monitor` object in the scenario definition file, is passed to the following method in the `Monitor` class:

```
public void Monitor.initRandomSeed(long val)
```

By extending the `Monitor` class and overriding the above method, you are able to control what method to use to initialise the random number generator you will be using within your simulation. For example:

```
static long current_seed;

void initRandomSeed(long val) {
    SomeRandomNumberClass.mySetSeed(current_seed = val);
}
```

This will save the seed to a class variable, `current_seed` and initialise the seed for random number generation.

The `Monitor` object has two other methods that we can utilize to help with the repeatability of simulation runs:

```
public void Monitor.runBefore()

public void Monitor.runAfter()
```

JACK Sim organises to execute the `runBefore()` method before a simulation iteration begins (before the `SETUP` and `START` events are sent to the agents) and the `runAfter()` method is called at the end of the simulation iteration (after the `END` events have been sent to the agents).

For example, the `runBefore()` method could be used to simply note which seed has been used to initialize the random number generator. (i.e. `current_seed`).

The `runAfter()` method could be used to reset the seed (and store it in `current_seed` for the next run.

Thus, every iteration will have a new and well-defined seed that can be used later to repeat a run if so desired.

5 Agent behaviours

5.1 Modelling actions with JACK

In the BDI world view, agents respond to events issued by the environment, other agents or by themselves. The behaviours that are triggered by these events result in computations (which take no time) and delays (which consume time). Thus, an action will in general be modelled as a computation and a delay, and will be triggered by an event. JACK provides programming constructs (events, plans, beliefsets, views and agents) and a generic execution model to support this world view.

For example, consider a robot that can perform pick and place operations. Assume that the source and destination for the operations are fixed and are potentially different for each part type to be moved. Furthermore, there are three part types involved, labelled A, B and AB. If the operation times are stored in a beliefset called `timings`, one could use the following event and plan types to model the robot's pick and place behaviour:

```
public event RobotOperation extends MessageEvent
{
    public String operation;     //"on", "off", "pick_and_place"
    public String part;          //null, "A", "B", "AB"

    #posting method pickAndPlace(String p) {
        operation = "pick_and_place";
        partType = p;
    }

    #posting method on(String p) {
        operation = "on";
        partType = null;
    }

    #posting method off(String p) {
        operation = "off";
        partType = null;
    }
}

public plan PickAndPlace extends Plan
{
    #handles event RobotOperation ev;
    #uses data RobotTimings timings;

    public static boolean relevant(RobotOperation ev)
    {
        return ev.operation.equals("pick_and_place");
    }

    body()
    {
        // check for preconditions here

        // "perform" the operation
        @sleep(timings.get(ev.partType).int_value());
```

```
        // apply postconditions here
    }
}
```

Note that the robot does not advance the simulation clock – that is the responsibility of the time source agent. However, the robot always has access to the current simulation time.

5.2 Behaviour execution

As noted in the *Basic application development* chapter, if an agent is fully managed by the time management infrastructure then agent creation and the triggering of execution is managed by the infrastructure. Contrast this with a normal JACK application, where one would write a small Java program to create agent instances and then perhaps invoke a method on one of the agents. This method would trigger agent activity through the posting of an event.

In certain situations, partial management of an agent is appropriate – this is discussed in the *Basic application development* chapter. In order to be fully managed, an agent must

- be registered with the time management infrastructure

- implement the `TimeManaged` (or `TimeSyncManaged`) marker interface

- have the `SimulationTiming` capability

- provide plans (and capabilities) to handle `RuntimeControl` events.

Registration with the time management infrastructure was discussed in the *Basic application development* chapter – if the agent has an initialisation entry in the scenario definition file for the process in which it will reside, then the loader will automatically register the agent.

With respect to the marker interfaces, an agent would normally implement the `TimeManaged` interface – use of the `TimeManaged` interface is discussed in the *Basic application development* chapter. Note that as we are dealing with marker interfaces, no methods are defined for implementation. By implementing the `TimeManaged` interface, an agent indicates to the time management infrastructure that it wants its execution to be managed by the infrastructure according to a synchronous protocol between the agent and its time dispatcher. The `SimulationTiming` capability is provided to hide the agent-side details of this protocol. If an agent incorporates this capability, the agent deals with `RunTimeControl` events rather than the `TimeControl` events that are issued by the time dispatcher. This is illustrated below:
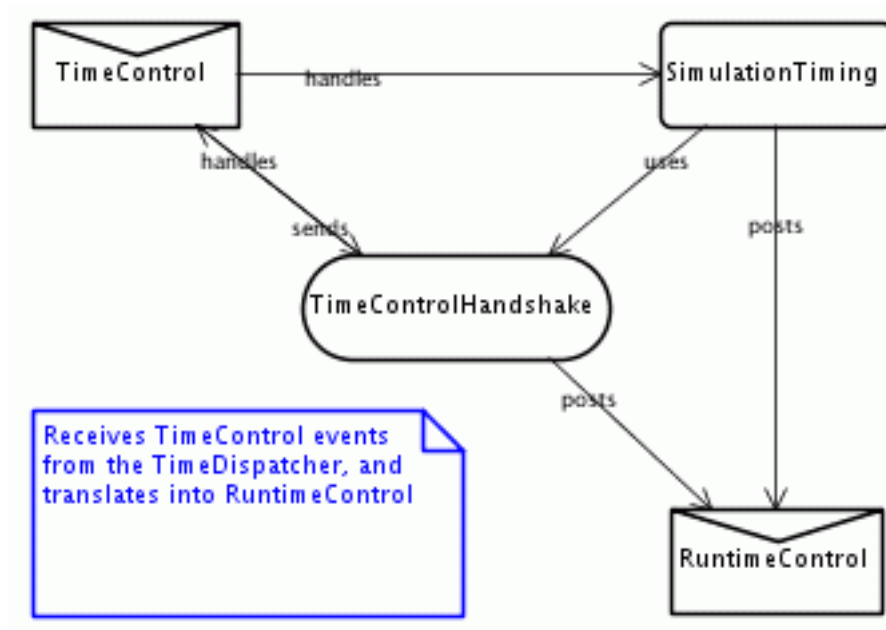
Figure 5-1: SimulationTiming capability

RuntimeControl events are sent at the following points in the agent's lifecycle:

- the start of the setup phase

- the start of the execution phase

- the end of the execution phase.

These points are distinguished by the RuntimeControl event's mode member. This variable can assumes a value of SETUP, BEGIN or END.

Normally SETUP and END events can be ignored by the agent. However, if running multiple iterations of the simulation, they should be used to initialise and take down, respectively, the simulation entities so that each iteration can begin in a clean and reset state.

BEGIN events **must** always be handled or no execution will occur. Note that the protocol is synchronous – the sending of a BEGIN event to the next agent will **not** proceed until processing of the preceding BEGIN event has completed. Therefore, the plan that handles this event should eventually post an event (that will actually initiate execution within the agent) and simply exit.

The protocol sends events to indicate the start and end of execution. During that time, there may be periods when the agent becomes blocked and is then unblocked. Blocking is managed by the agent through the use of, for example, @sleep or @waitFor statements. Unblocking is managed by the time dispatcher and requires no actions to be performed by the agent. Also

note that clock advancement is handled by the infrastructure and not the agent. However, the agent always has access to the current simulation time.

5.3 Example 2

This example describes the simulation framework that was used to test an agent-based control system for a robotic assembly cell prior to its commissioning. The cell control system was developed using Teams and is described in more detail in (Jarvis et al., in press).

5.3.1 Description

The assembly cell consisted of the following components:

- Two buffers, which presented components (A, B and C) to the system.

- A Fanuc robot (robot 1) fitted with a vacuum activated gripper. This was able to pick a component from a buffer and place it on a jig on the table.

- A Hirata robot (robot 2) with a screwing capability.

- A rotating table with two assembly jigs mounted at 180 degrees. The table could move between two positions – one with jig1 adjacent to the Fanuc robot and one with jig1 adjacent to the Hirata robot.

- A flipper unit that accepted a sub-assembly (AB) and turned it over.

The purpose of the cell was to assemble meter boxes. These consisted of three components – an open-faced metal box, a metal plate and a cover. The cover and the plate were attached to the box with screws. We refer to the box as component A, the plate as component B, the cover as component C and the box plus plate sub-assembly as component AB. Components A, B and C were available from their respective buffers. Component A could only be placed by the Fanuc robot in an empty jig. Component B could only be placed in a jig that contained a component A. These activities could only take place when the target jig was in the position closest to the Fanuc robot. When a jig contained components A and B, it could be rotated to position 2, where the components were screwed together by the Hirata robot to form component AB. AB could then be rotated to position 1, where the Fanuc robot could remove it from the system. Alternatively, a fully assembled meter box (ABC) could be made. In this case, the Fanuc robot removed AB from the jig and placed it in the flipper unit. The robot then placed component C in the empty jig and, at the same time, the flipper unit flipped the assembly through 180 degrees. The upside down AB was then placed on top of component C by the robot and the jig was rotated to position 2, where AB and C were screwed together by the Hirata robot. The completed assembly was then rotated to position 1, where it was removed from the system by the Fanuc robot.

The cell was being used to explore the issues involved in implementing agent-based control strategies using existing manufacturing controllers. Consequently, infrastructure had been developed that enabled external access to machine functionality via a simple machine state

abstraction. These abstractions were managed by a Visual BASIC program called BBS. (In the code provided with the distribution, a simplified Java version of BBS is provided).

A machine state consists of two words – a status word and a control word. The key components of the machine state from an execution perspective are

- the four program bits in the control word. These specify the function that the machine is to execute.

- the go bit in the control word. If this is true, the specified function is ready to start. Once the function has started, the go bit and the idle bit are set to false.

- the idle bit in the status word. If the machine is performing a function, this is set to false. Otherwise it is set to true.

Agent interaction with the BBS program was via a UDP connection. Machine control was implemented in a hierarchical manner; the actual machines were controlled by an Omron PLC which in turn was controlled by the BBS program. Furthermore, the robots had their own controllers under the control of the PLC.

BBS has two modes of operation, *operational mode* corresponding to operation of the actual cell and *simulation mode* corresponding to operation of a virtual cell. In both cases the interface between BBS and the cell control agents was identical.

5.3.2 Architecture

The cell architecture followed the reference model presented in the *Simulation architecture* chapter. Equipment and environment models were not required. The behaviour model was represented as a JACK team (called `CellBehaviour`) that required the following roles in order to make meter boxes:

Role	Description
PickAndPlace	Load a component into a fixture (jig or flipper) or unload a component from a fixture.
Transfer	Move a fixture (jig) containing a part to a processing station (Fanuc or Hirata) establishment.
Fasten	Assemble components (A+B or AB+C).
Flip	Turn a component upside down (AB).

Table 5-1: Roles required by the `CellBehaviour` team

The machines that form the cell (the two robots, the table and the flipper) are represented in both the behaviour model and the embodiment model. In the behaviour model, they are represented as teams; the roles that these teams perform are listed below.

Team	Role
FanucBehaviour	PickAndPlace
HirataBehaviour	Fasten
TableBehaviour	Transport
FlipperBehaviour	Flip

Table 5-2: Teams available to form the `CellBehaviour` team

Two separate embodiment models are defined, corresponding to the actual cell or the virtual cell. Both use BBS to manage the interaction with the behaviour model. BBS then controls either the actual machine behaviours or simulated machine behaviours. In the latter case, each machine in the virtual cell is represented as a JACK agent that interacts with BBS via a JACK view. The agents that form the the virtual cell are listed below.

Agent	Machine
FanucEmbodiment	Fanuc
HirataEmbodiment	Hirata
TableEmbodiment	Table
FlipperEmbodiment	Flipper

Table 5-3: Agents that form the virtual cell

The interconnection layer between the behaviour model and the embodiment model consists of

- a UDP client on the behaviour side. Each machine team has a JACK view that mediates access to the client.

- a UDP server on the embodiment side, encapsulated within the BBS program. BBS then manages the interaction with either the actual cell or the virtual cell. In the latter case, each agent has a JACK view that mediates access to the server.

5.3.3 The virtual cell

The interaction between a virtual cell agent and BBS is encapsulated in a JACK view. This
enables the agent to manage its status through the following queries:

```
// Receive notification when the idle bit becomes value.
outputIdle(boolean value)

// Receive notification when the go bit becomes value.
// The bit is tested every rate milliseconds
inputIdle(int rate, boolean value)
```

The function to be executed is accessible through bits 1 – 4 of the view's input data member.

The agent can then use the following plan to execute a machine operation:

```
public plan VirtualLife extends Plan {
    static long RATE = 1000;
    #handles event VirtualStart ev;
    #posts event VirtualControl vc;
    #uses data BBSConnection bbs;

    static boolean relevant(VirtualStart ev)
    {
        return true;
    }

    context()
    {
        true;
    }

    #reasoning method
    body()
    {
        for (; ; ) {
            // set the idle bit
            @waitFor(bbs.outputIdle(true));
            // wait until the go bit is set
            @waitFor(bbs.inputIdle(RATE,true));
            // clear the idle bit
            @waitFor(bbs.outputIdle(false));
            // wait until the go bit is cleared
            @waitFor(bbs.inputIdle(RATE,false));
            // execute the requested operation
            if (@subtask(vc.control(bbs.input))) ;
        }
    }
}
```

Execution of an operation by a particular machine is initiated by the posting of a
`VirtualControl` event within the `VirtualLife` plan. Plans are provided with each virtual
machine agent to handle this event. The plan provided for `FanucEmbodiment` is presented
below.

```
public plan FanucLife extends Plan {

    // delays are in msecs
    static long[] delays = {
            15000,        // A from buffer to jig
            15000,        // B from buffer to jig
            15000,        // C from buffer to jig
            10000,        // AB from jig to flipper
            10000,        // AB from flipper to jig
            15000,        // ABC from jig to buffer
    };
    #handles event VirtualControl ev;

    static boolean relevant(VirtualControl ev)
    {
        return true;
    }

    context()
    {
        true;
    }

    #reasoning method
    body()
    {
        // convert bits 1-4 of the control word to the program
        // number
        int program = (ev.value >> 1) & 0xF;
        // wait the appropriate length of time
        @waitFor(elapsedMillis(delays[program]));
    }
}
```

The agents in the virtual cell are managed by JACK Sim. A StartEmbodiment plan is provided
that handles a JACK Sim BEGIN event. The StartEmbodiment plan simply posts a
VirtualStart event and exits so that JACK Sim can continue execution. The VirtualStart
event is then handled by the VirtualLife plan discussed earlier. The generic aspects of the
above behaviour are encapsulated in the StartingUp capability:
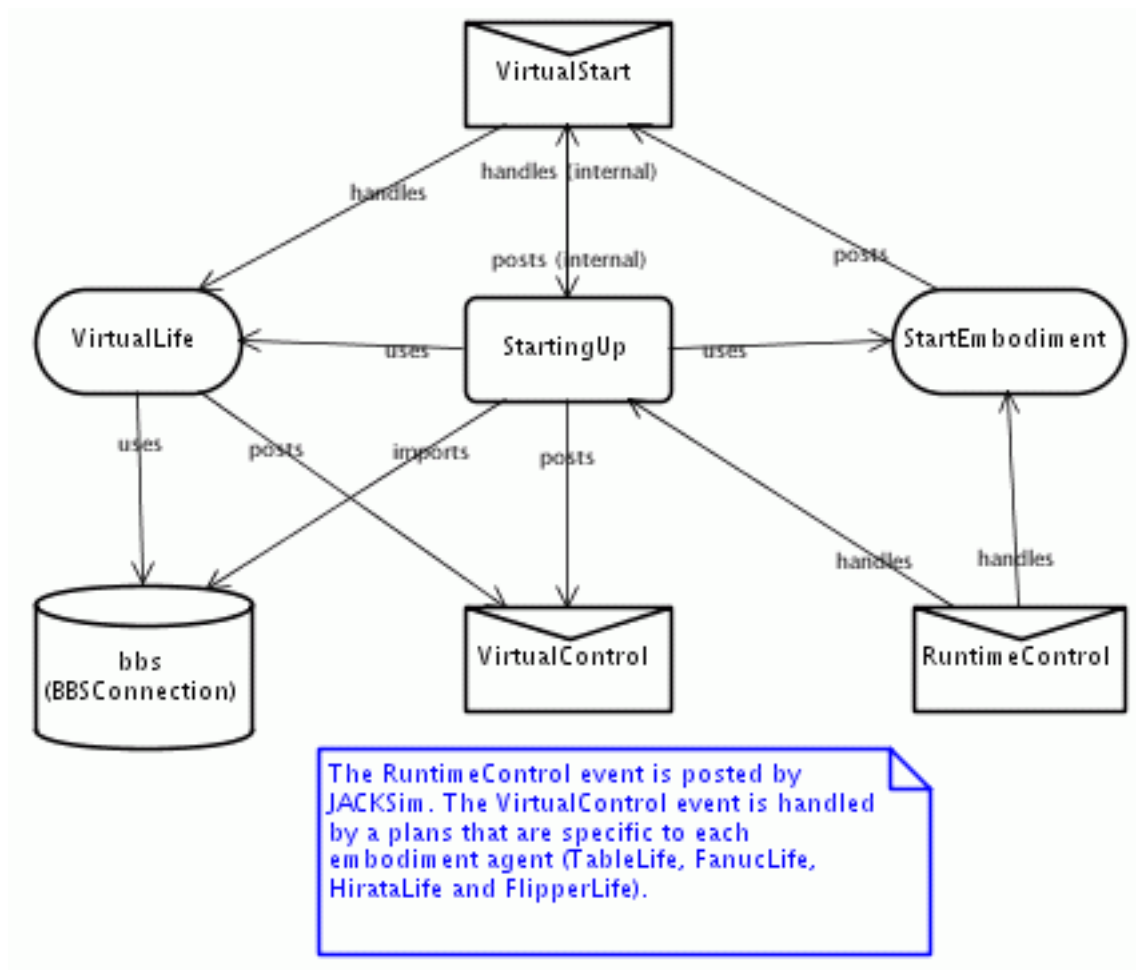
Figure 5-2: StartingUp Capability

The following scenario definition file could be used to start up the virtual cell:

```
/**
 * This file defines and runs the virtual cell
 */

// We use the JACK Sim time management
<Include :dict "aos.jack.sim.time.Init__base" >

// Set simulation time.
<TimeInit :date "Fri, June 25, 2004, 12:00:00.0" >

// Declare a time dispatcher agent. It ensures that the time is
// not advanced while the application is busy.
<TimeDispatcherInit :name "time dispatcher" >

// Instantiate the agents.
<AgentInit :agent_type "virtual.fanuc.FanucEmbodiment"
        :name "fanuc" >
```

```
<AgentInit :agent_type "virtual.hirata.HirataEmbodiment"
        :name "hirata" >
<AgentInit :agent_type "virtual.table.TableEmbodiment"
        :name "table" >
<AgentInit :agent_type "virtual.flipper.FlipperEmbodiment"
        :name "flipper" >

// Declare a time source agent. It is responsible for advancing
// time in a synchronised manner with a time dispatcher agent.
<TimeSourceInit :name "time source" :dispatcher "time dispatcher"
    :verbose 0
    :realtime :false
    :delay 0
>
```

6 Visualisation

6.1 Introduction

The JACK Sim visualisation layer provides a convenient mechanism for JACK developers to provide external visibility for entities within their agent system. A common usage would be to visualise entities in a physical system (either real or simulated). For example in an air-traffic control simulation, aircraft agents may be visualised on screen, with JACK Sim visualisation entities reflecting the current position and heading of aircraft.

6.2 The graphics model

The graphics model that is used by the infrastructure is that of AWT. In this model, the origin of the coordinate system is at the top left hand corner of the display. X increases to the right and y increases downwards. Rotation is of the coordinate system; a positive angle of rotation is in the direction from the positive x axis to the positive y axis. Translation is performed relative to the enclosing coordinate system.

6.3 The software model

It has long been recognised as good software engineering practice to separate model behaviour from model visualisation. One of the practical consequences of such a separation is that different visualisation models can then be applied to the same behaviour model. The JACK Sim visualisation infrastructure facilitates such a separation by supporting a loose coupling between the behavioural aspects of an application with its visualisation aspects.

On the visualisation side, the infrastructure requires appearance objects to have been explicitly constructed. The functionality of these objects is concerned solely with presentation; they contain no behavioural aspect. The infrastructure also manages the actual display of the visualisation model.

On the behaviour side, there is an expectation that the reference model presented in Chapter 3 will have been adopted. That is the application consists of separate behaviour and embodiment models and that the embodiment model maintains explicit representations of the execution states of the agents. The infrastructure then supports the updating and display of the visualisation model as the execution states change. The information needed to update the visualisation model is of course application specific and the developer needs to specify how the visualisation model is to be updated. However the actual updating of the model is managed by the infrastructure.

Consequently, the following models would normally be present in a JACK Sim application that uses visualisation:

• Behaviour model

- Embodiment model

- Visualisation model

The interaction between the models and the infrastructure is summarised in the figure below. The visualisation model is updated by the visualisation infrastructure at regular time intervals; the updating is on the basis of the current state of the embodiment model's execution state.
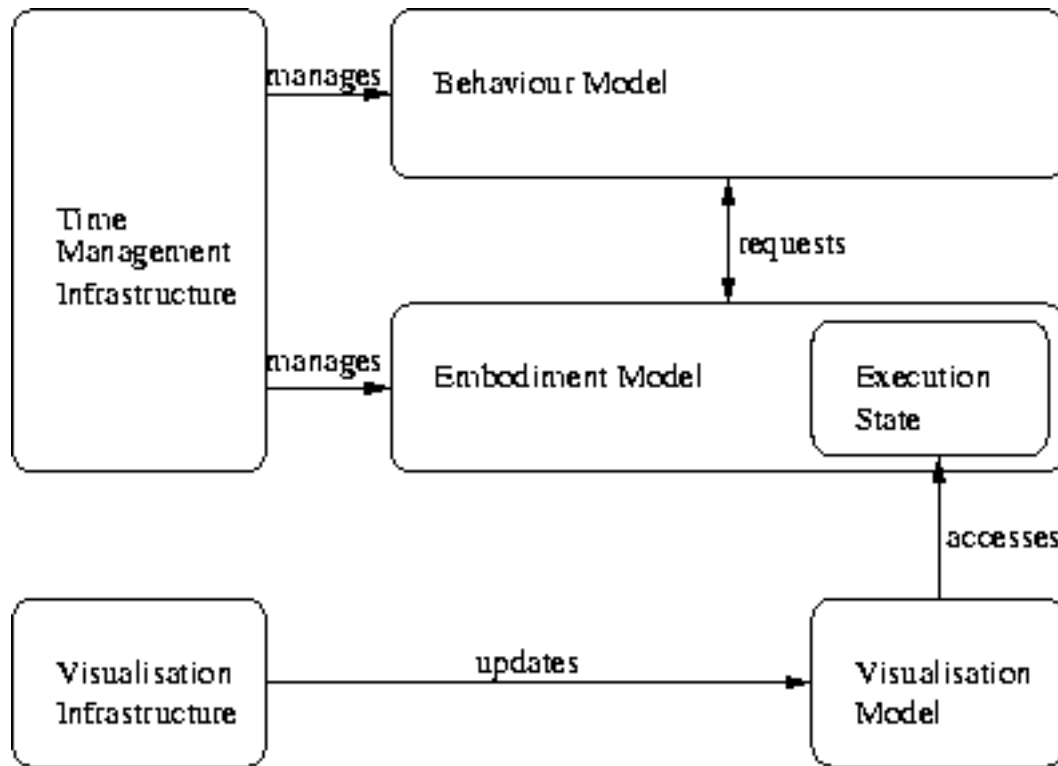
Figure 6-1: Interaction between JACK Sim models and the infrastructure

6.4 Basic Visualisation Model Development

Visualisation model development involves two phases – the development of the appearance model for the application and the development of the updating model. The appearance model consists of the frame that is to be used for visualisation and appearance objects that specify the appearance structure for the entities that are to be visualised. Both the visualisation frame and the appearance objects for the application are specified using JACOB initialisation objects that are contained in a file known as the graphical definition file. Appearance objects contain transformation fields that can be initialised by the user and updated dynamically. An appearance object can contain appearance objects; the values of the transformation fields of parent objects are passed on to their children. However, the transformation fields of a child

object can be modified independently of its parent object. Thus a helicopter can be modelled as a shell and a rotor and while the helicopter is flying, the rotor can be rotated.

The updating model is responsible for the drawing of entities on the visualisation frame and is achieved by visual entity objects. These objects are normally created dynamically and have their associated appearance object bound at construction time. In addition to providing this binding, the developer must also specify how the transformation fields of the appearance object are updated. The actual updating process is triggered by the visualisation infrastructure; an `Updater` agent and a `VisualsControl` view are provided to facilitate this activity.

6.4.1 The appearance model

The visualisation frame and the appearance objects together constitute the appearance model for the application. These objects are specified using the JACOB Object Modelling Language in a file known as the graphical definition file. As with the scenario definition file, `Include` objects are available to include dictionaries and other definition files. In order to utilise the visualisation infrastructure, the following entries must appear at the start of the graphical definition file:

```
// Standard drawable components.
<Include :dict "aos.jack.sim.visual.awt.Init__awt" >

// Standard visual model extension
<Include :dict "aos.jack.sim.visual.Init__visual" >
```

6.4.1.1 The visualisation frame

The visualisation frame is created using a `VisualFrameInit` object. The following fields are available for customisation of the display:

Field	Type	Description
title	String	The title for the frame's window
x	int	The x location (in pixels) of the window, relative to the screen origin. The default is 10.
y	int	The y location (in pixels) of the window, relative to the screen origin. The default is 10.

Field	Type	Description
width	int	The width (in pixels) of the frame. The default is 800.
height	int	The height (in pixels) of the frame. The default is 600.
image	String	A JPEG image to be used as background. The default is "aos/jack/sim/ bg800x600.jpeg"
scale	String	The scaling factor to be applied to the frame.

Table 6-1: Initialisation attributes for a `VisualFrameInit` object

For example, suppose that we have created a graphical definition file called `graphical.def` that contains the following entries:

```
// Standard drawable components.
<Include :dict "aos.jack.sim.visual.awt.Init__awt" >

// Standard visual model extension
<Include :dict "aos.jack.sim.visual.Init__visual" >

<VisualFrameInit :title "JACK Sim Visualisation Test"
    :x 120
    :y 120
    :width 600
    :height 400
>
```

If we were to invoke the JACK Sim loader as follows:

```
java aos.jack.sim.run.Loader graphical.def
```

then an empty frame with the title "JACK Sim Visualisation Test" would appear on the screen at (120,120).

6.4.1.2 Appearance object definition

Appearance objects are of type `Named`. The visualisation infrastructure provides the `DefineNamed` class to enable the developer to declaratively specify the structure and initial appearance of the appearance objects that are to be employed in an application. These definitions are contained in a JACOB initialisation file (the graphical definition file) that is processed by the JACK Sim loader at startup. The loader stores the resulting `Named` objects internally; these are then available for binding with the visual entity objects that are created in the updating model.

The `DefineNamed` class extends the `Transform` class through the provision of a `name` field. As noted above, this name is used in the updating model to bind an appearance object to a visual entity object. It is also used in appearance object definitions to incorporate appearance objects

into appearance object definitions. The following fields are available from the `Transform` class when a `DefineNamed` object is specified:

Field	Type	Description
label	String	A name that uniquely identifies the transform. This is used by the updating model to identify sub-trees of an appearance object structure that are to be explicitly updated.
drawable	Drawable	An object that defines the actual appearance of the appearance object. This object must implement the `Drawable` interface.
x	double	The x origin of the coordinate system for the object
y	double	The y origin of the coordinate system for the object
theta	double	the angle of rotation for the coordinate system
scale	double	the scale factor for the object

Table 6-2: Initialisation attributes of the `Transform` class that are available to a `DefineNamed` object definition

The following set of primitive drawable objects are provided by the infrastructure for specifying the appearance of an appearance object:

- `Arc`

- `Area`

- `CachedImage`

- `Colored`

- `Ellipse`

- `Figure`

- `Font`

- `Line`

- `Point`

- `Polygon`

- `Rectangle`

- `RoundRectangle`

- `TextLine`

- `Transform`

These objects, their initialisation attributes and related classes are described in Appendix B.

Note that the `Figure` object is available to provide a container for drawable objects. Since appearance objects (i.e. `Named` objects) are drawable objects, they can be incorporated into a `Figure` object definition, as well as the primitive objects listed above. Like the `DefineNamed` class, the `Named` class extends `Transform` and the following fields can therefore be referenced in a `Named` object specification

Field	Type	Description
x	double	The x origin of the coordinate system for the object
y	double	The y origin of the coordinate system for the object
theta	double	the angle of rotation for the coordinate system
scale	double	the scale factor for the object

Table 6-3: Initialisation attributes of the `Transform` class that are available to a `Named` object definition

The `drawable` field is not initialised in a `Named` specification, as it is set by the infrastructure to be a reference to the appropriate `DefineNamed` object.

Note that each appearance object has its own coordinate system. The origin of this coordinate system is specified by the object's `x` and `y` fields. Rotation is of the coordinate system. Modification of the `x` and `y` fields results in a translation of the object; this is relative to the origin of the coordinate system of the enclosing `DefineNamed/Named` object if there is one, or of the visualisation frame otherwise.

6.4.2 Example 3

As an example of appearance model construction, consider the rotating table of Section 5.3. It contains two diametrically opposed jigs, and for pedagogical purposes, we will allow these jigs to be rotated and translated independently of the table. Also we will want to visualise the contents of the jigs. Thus the table appearance object (`n-table` will contain two jig appearance objects (`n-jig1` and `n-jig2` and each jig appearance object will contain a status appearance object (`n-status1` and `n-status2`).

The table appearance could be defined as follows:

```
<DefineNamed
    <Transform
        :label "l-table"
        :drawable
            <Figure
                :elements
                (
                    <Ellipse
                        :width 100
                        :height 100
                        :x -50
                        :y -50
                    >
                    <Named
                        :name "n-jig1"
                        :x -37
                        :y   0
                    >
                    <Named
                        :name "n-jig2"
                        :x  37
                        :y   0
                    >
                )
            >
    >
    :name "n-table"
>
```

The locations of the three objects that form the table are specified relative to the origin of the enclosing coordinate system, which will be set at runtime. If no origin is specified, (0,0) is used. Recall that rotation is of the coordinate system. Thus when we rotate the coordinate system of the table, the components of the table will be rotated about the centre of the table.

The jigs are defined similarly; the definition for jig1 could be:

```
<DefineNamed
    <Transform
        :label "l-jig1"
        :drawable
            <Figure
                :elements
                (
                    <Rectangle
                        :width 20
                        :height 20
                        :x -10
                        :y -10
                    >
                    <Named
                        :name "n-status1"
                        :x -2
                        :y  2
                    >
                )
```

```
            >
        >
        :name "n-jig1"
    >
```

Again, the location of the components is defined relative to the enclosing coordinate system – in this case, that of the `Named` object `n-jig1`. As before, when the coordinate system of the jig is rotated, the components of the jig will rotate about the centre of the jig.

The contents of the jig are represented for simplicity as a digit in the range 0 to 3. 0 corresponds to an empty jig, 1,2 and 3 correspond to the progressive stages of assembly. Unlike the other components of the table appearance, this component is not statically defined and will change dynamically. As we shall see in the next section, this is achieved by dynamically creating a new `TextLine` object when the jig contents change and assigning it to the `drawable` field of the appropriate `Named` object. Consequently there is no need to specify an initial appearance and the definition for the jig1 status is simply

```
    <DefineNamed
        <Transform
            :label   "l-status1"
        >
        :name "n-status1"
    >
```

In order for an appearance object to be displayed by the visualisation infrastructure, it needs to be incorporated into a visual entity object. In the next section, we shall explain how the updating model together with the visualisation infrastructure updates and displays visual entity objects. For now, we will just display the initial appearance of the table as defined in the graphical definition file.

`VisualEntityInit` initialisation objects are provided by the visualisation infrastructure for the creation of static visual entity objects. Definitions for these objects are incorporated in the graphical definition file and the corresponding `VisualEntity` objects are then created by the JACK Sim loader. An `Updater` agent is provided by the infrastructure to manage the display of visual entity objects. Its definition is incorporated in the graphical definition file with a `ScreenUpdaterInit` object.

If the following entries were added to the complete graphical definition file that has been used in the preceding discussion, the appearance of the table can be checked without having to provide any agent behaviours.

```
    <VisualEntityInit
        :name "v-table"
        :drawable <Named :name "n-table">
        :x 100
        :y 100
    >

    <ScreenUpdaterInit>
```

Entering the following command

```
java aos.jack.sim.run.Loader graphical.def
```

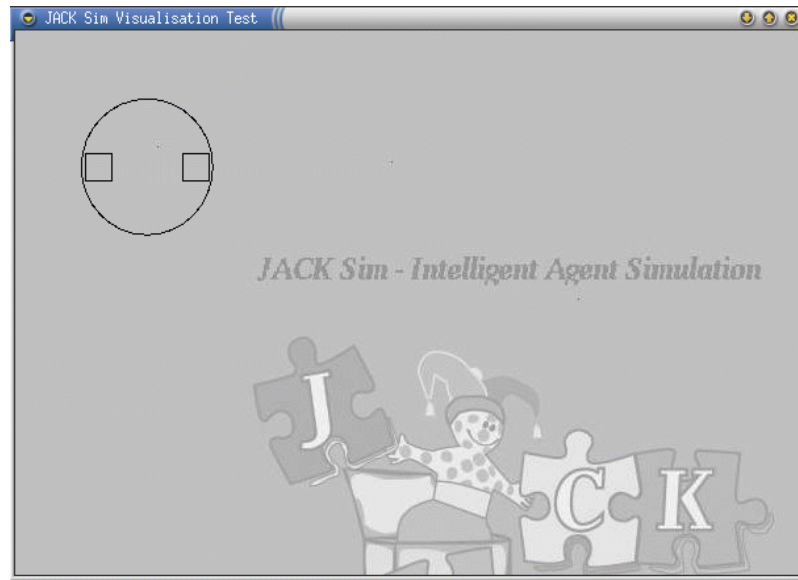will result in the appearance of a display similar to the following:

Figure 6-2: Initial table appearance.

6.4.3 The updating model

The JACK Sim visualisation infrastructure provides an `Updater` agent to manage the display of the visual entities associated with an application. As noted previously, visual entities are of type `VisualEntity` and each entity is associated with an appearance object that was defined as part of the appearance model. The infrastructure provides a `VisualsControl` view to assist in the lifecycle management of `VisualEntity` objects. The updating and display of the visual entities is managed by the infrastructure, but the developer must provide the code that is invoked by the infrastructure at each iteration in the update/display loop.

6.4.3.1 The `Updater` agent

The `Updater` agent is responsible for regularly updating the visual entities for the application and displaying them on the screen. A `ScreenUpdaterInit` object is provided to allow the JACK Sim loader to create and initialise an `Updater` agent from a JACOB definition provided in the graphical definition file. The available initialisation attributes are:

Attribute	Type	Description	Default
name	String	The name for the `Updater` agent.	"screen"
realtime	boolean	Indicates whether the simulation time advancement is (if possible) synchronised with real time.	false
rate	double	This specifies the rate at which the screen is updated.	1

Table 6-4: Initialisation attributes for the `ScreenUpdaterInit` object

6.4.3.2 The `VisualsControl` view

Appearance objects implement the `Drawable` interface and can therefore be drawn on the visualisation frame. However, drawing management is a complicated issue and the visualisation infrastructure uses visual entity objects to facilitate the efficient drawing of appearance objects. Visual entity objects are of type `VisualEntity`.

As noted previously, objects of type `VisualEntity` can be created from `VisualEntityInit` entries in the graphical definition file. However the preferred method to create such objects is through the `VisualsControl` view that is provided by the infrastructure. The `VisualsControl` view provides the `newVisual` method for this purpose; it takes the following arguments:

Field	Type	Description
name	String	The name for the visual entity object that is to be created
appearance	String	The name of the appearance object that is to be incorporated into the visual entity object
x	double	The x origin of the coordinate system for the visual entity object
y	double	The y origin of the coordinate system for the visual entity object
theta	double	the angle of rotation for the coordinate system
scale	double	the scale factor for the visual entity object

Table 6-5: Arguments for the `newVisual()` method

Visual entity objects are stored by the `VisualsControl` view in a `Hashtable` called `entities`. The `newVisual()` method does not return an object reference; if a reference is required, it can

be obtained from the hashtable by specifying the name of the visual entity as the key to an invocation of `entities.get()`. This reference can then be used to interrogate the visual entity object and its appearance object.

Having created a visual entity object, the initial state of the object can be displayed by the infrastructure, but subsequent changes to its state will not be reflected on the display. In order for the visualisation infrastructure to display the updated entity state, it needs to be provided with an updating method for each visual entity. The infrastructure will then invoke these methods at regular intervals and update the display.

The linkage between a visual entity and its updating method is provided by the `bindVisualComponent()` method in the `VisualsControl` view. This method takes the following arguments:

Field	Type	Description
component	VisualComponent	A `VisualComponent` object that defines an `update()` method that will be invoked by the infrastructure.
name	String	The name of the visual entity object that is to be updated.
label	String	The label of the appearance object that is to be updated. If only the visual entity object is being updated, this is set to `null`.

Table 6-6: Arguments for the `bindVisualComponent` method

Recall that appearance objects can contain appearance objects, thus allowing hierarchical appearance structures to be defined. The `bindVisualComponent()` method enables updating to be defined for any appearance object within an appearance structure. As expected, the updating of a node in an appearance structure is applied to all of its sub-nodes.

The label of the appearance object is the name that has been assigned to the object's `label` field. The name of the visual entity object is the name that was assigned to the visual entity object when it was created (using the `newVisual()` method), and **not** the name of the appearance object that was provided to that method as its `appearance` parameter.

The object that is provided as the first argument to `bindVisualComponent` must implement the `VisualComponent` interface. This interface consists of the single method

```
public void update(Transform whole,Transform part);
```

`whole` provides access to the data members of the visual entity object that are provided by the `Transform` class, namely its location, orientation, scaling and its appearance object. `part` provides similar access for the appearance object that is being updated. The expectation is that

within the `update` method the current state of the appearance object being updated (and that of its root visual entity object) will be obtained from the execution state for the application. A decision is then made as to whether or not the transform fields of the appearance object should be updated.

6.4.4 Example 4

In this example, we will use the JACK Sim visualisation infrastructure to create an updating model for the rotating table whose appearance model was defined in Example 3. Recall that the table has two diametrically opposed jigs and it is rotated between a loading/unloading station and a joining station. Only the behaviour of the table and its jigs is modelled; we do not model the operation of the two stations. For pedagogical reasons, we allow the jigs to be rotated and also to be advanced and retracted along a slide. The complete source code for Examples 3 and 4 can be found in `examples/jacksim/table`.

6.4.4.1 Design Overview

The focus of this example is the use of the visualisation infrastructure and not behaviour modelling. Consequently we have chosen to use the behaviour model as a test harness for the remaining aspects of the application and have not attempted to provide a detailed or realistic model of table behaviour. The model is implemented as a capability called `BatchMaking`. We assume that only a single batch of one type of part is to be produced and that only one of the jigs is used. Therefore only a single plan is provided to model the table behaviour during assembly. This plan is defined in terms of operations that are performed by the table (rotate) and the two jigs (set contents, rotate, advance and retract). These operations are performed by jig and table controller objects of type `JigController` and `TableController` respectively. The controller objects collectively form the embodiment model for the table and for convenience are incorporated into a view called `TableModel`. Each controller object has a set of state variables that encapsulate the execution state of its corresponding entity. When an operation is performed by a controller object, a delay is initiated and its state variables are updated. Independently of this process, the visualisation infrastructure updates and displays the visualisation model at regular intervals using the current state of the table as defined by the state variables of the controller objects.

As noted above, the embodiment model is a view that incorporates three objects – two of type `JigController` one of type `TableController`. These classes extend a base class of type of `VirtualController`, which provides generic rotation, translation and scaling operations. Subclasses of `VirtualController` either provide specialisations of these operations or new operations. For example, the `JigController` class provides operations for slide and contents management.

The visualisation model is encapsulated in a view called `TableVisualisation`. It creates the visual entity objects required by the model using methods provided by the `VisualsControl` view, which is part of the visualisation infrastructure. Each visual entity object that is created incorporates a corresponding appearance object. Appearance object definition for this

application was addressed in Example 3 – the definitions are provided as JACOB initialisation objects in the graphical definition file for the application. Also for each visual entity, an object that contains the updating method for the visual entity is created. Note that the `TableVisualisation` view is responsible only for the creation of the objects that will form the visualisation model – the actual management of the model (object updating and display) is the responsibility of the `VisualsControl` view and the `Updater` agent.

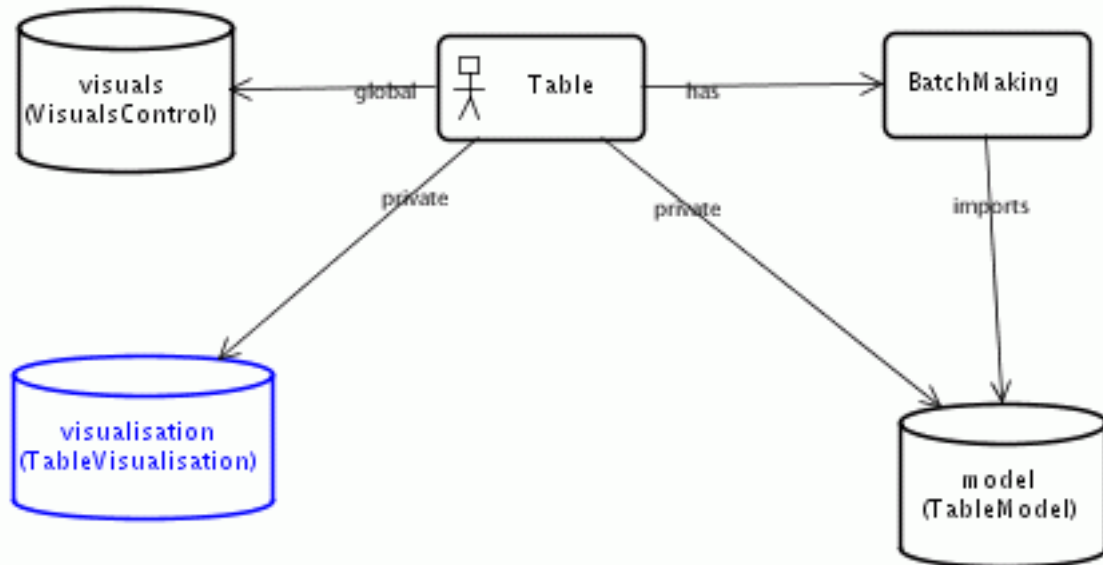A design diagram illustrating the overall agent architecture is shown below:

Figure 6-3: Application Architecture

6.4.4.2 The embodiment model

The embodiment model is provided by three controller objects corresponding to the two jigs and the table. For convenience, these objects are contained within a single JACK view. From a visualisation perspective, the key aspects of the embodiment execution state are

- table orientation

- jig contents

- jig orientation

- jig location

The basic transformations that are supported by the visualisation infrastructure are rotation, translation and scaling. A base class of type `VirtualController` provides basic rotation, translation and scaling operations. `JigController` and `TableController` then extend

VirtualController. The aspects of the VirtualController.java that relate to rotation are shown below:

```java
package jsv.table;

import aos.jack.util.cursor.Action;

public class VirtualController {

    // only the rotation behaviour is shown

    double orientation;      // orientation of the coordinate system
    double da;               // step angle
    int dat;                 // rotation time for a step (msecs)

    // constructors go here

    public double getOrientation()
    {
        return orientation;
    }

    public void setOrientation(double o)
    {
        orientation = o;
    }

    // rotate coordinate system

    public Action rotate(double a)
    {
        return new RotateAction(a);
    }

    // Implementation of RotateAction omitted
    // It updates the rotation state variable (orientation) at
    // regular time intervals during the rotation.
}
```

The TableController class does not add any additional functionality to the base class, as in this example the table is only capable of rotation. However the JigController class adds advance/retract functionality and also maintains knowledge of the jig contents. The code for the JigController.java is shown below:

```java
package jsv.table;

import aos.jack.util.cursor.Action;

public class JigController extends VirtualController {

    // jig content states
    public static final int STATE0 = 0;      // empty
    public static final int STATE1 = 1;      // A
    public static final int STATE2 = 2;      // A+B
    public static final int STATE3 = 3;      // AB

    String[] contentStates = { "0","1","2","3" };

    // slide movements
```

```
public static final int XINCREASING  = 1;
public static final int XDECREASING  = -1;
public static final int YINCREASING  = 2;
public static final int YDECREASING  = -2;

String contents;        // indicates the contents of the jig
int xOffset;    // distance from home for an x-slide
int yOffset;    // distance from home for a y-slide
int length;     // slide length
int dt;         // time for a step (msecs)

public JigController(double da, int dat,int l, int st)
{
    super(0.0,0.0,0.0,da,dat,0.0,0);
    contents = "0";
    xOffset = 0;
    yOffset = 0;
    length = l;
    dt = st;
}

public String getContents()
{
    return contents;
}

public int getXOffset()
{
    return xOffset;
}

public int getYOffset()
{
    return yOffset;
}

public void setContents(int s)
{
    contents = contentStates[s];
}

public void setOffsets( int dx, int dy)
{
    xOffset = dx;
    yOffset = dy;
}

public Action advance(int d)
{
    int p1 = (d < 0) ? -length : length;
    return new SlideAction(0,p1,d);
}

public Action retract(int d)
{
    int p0 = (d > 0) ? -length : length;
    return new SlideAction(p0,0,d);
}

// implementation of SlideAction omitted
```

```
        // It updates the slider state variables (xOffset and yOffset)
        // at regular time intervals during slide advancement/retraction.

    }
```

Note that the advance/retract movement is defined as movement relative to the home position of the slide, unlike the movement function provided by the base class, in which movement is in between two points in the behaviour model coordinate system.

6.4.4.3 The visualisation model

In our example, visualisation model updating is encapsulated in a view called `TableVisualisation`. It provides a single method `bind()` that is invoked by the `Table` agent as part of its initialisation.

```
    package jsv.table;

    import aos.jack.sim.run.Loader;
    import aos.jack.sim.visual.VisualsControl;
    import aos.jack.sim.visual.VisualComponent;
    import aos.jack.sim.visual.VisualEntity;
    import aos.jack.sim.visual.awt.Transform;
    import aos.jack.sim.visual.awt.TextLine;
    import aos.jack.sim.visual.awt.DefineNamed;

    public view TableVisualisation {

        #uses data Loader loader;
        #uses data VisualsControl visuals;
        #uses data TableModel model;

        class JigState {

            String status = "";
            String previousStatus = "";
            int xOffset = 0;
            int previousXOffset = 0;
            double homeX = 0.0;
        };

        // only updating for the table and jig1 are shown here

        JigState j1;

        TableController table;
        JigController jig1;

        public void bind()
        {
            j1 = new JigState();

            table = model.getTableController();
            jig1 = model.getJigController(TableModel.JIGPOINT1);

            // create the table visualisation object
            visuals.newVisual(
                "v-table","n-table",table.getX(),table.getY(),0,1);
```

```
              // set homeX for jig1 (sliding is relative to this point)
              VisualEntity ve =
                  (VisualEntity) visuals.entities.get("v-table");
              Transform t1 = ve.findTransform("l-jig1");
              j1.homeX = t1.x;

              // define how the table visualisation is to be updated
              visuals.bindVisualComponent(
                  new VisualComponent() {
                      public void update(Transform w, Transform p) {
                          p.theta = table.getOrientation();
                      }
                  },
                  "v-table",
                  null
              );

              // define how the jig1 visualisation is to be updated
              visuals.bindVisualComponent(
                  new VisualComponent() {
                      public void update(Transform w, Transform p) {
                          p.theta = jig1.getOrientation();
                          j1.xOffset = jig1.getXOffset();
                          if (j1.xOffset != j1.previousXOffset)
                          {
                              p.x = j1.homeX+j1.xOffset;
                              j1.previousXOffset = j1.xOffset;
                          }
                      }
                  },
                  "v-table",
                  "l-jig1"
              );

              // define how the jig1 status is to be updated
              visuals.bindVisualComponent(
                  new VisualComponent() {
                      public void update(Transform w, Transform p) {
                          j1.status = jig1.getContents();
                          if (!j1.status.equals(j1.previousStatus)) {
                              TextLine t = new TextLine();
                              t.string = j1.status;
                              p.drawable = t;
                              p.instantiate(null);
                          }
                          j1.previousStatus = j1.status;
                      }
                  },
                  "v-table",
                  "l-status1"
              );

          }
      }
```

6.4.4.4 The behaviour model

The focus of this example is the use of the visualisation infrastructure and not behaviour modelling. Consequently we have chosen to use the behaviour model as a test harness for the

remaining aspects of the application and have not attempted to provide a detailed or realistic model of table behaviour. We assume that only a single batch of one type of part is to be produced and that only one of the jigs is used. Therefore only a single plan is required to model the table behaviour during assembly. This plan is defined in terms of operations that are performed by the table embodiment (rotate()) and the two jig embodiments (setContents(), rotate(), advance() and retract()). A design diagram for the behaviour model is shown below:
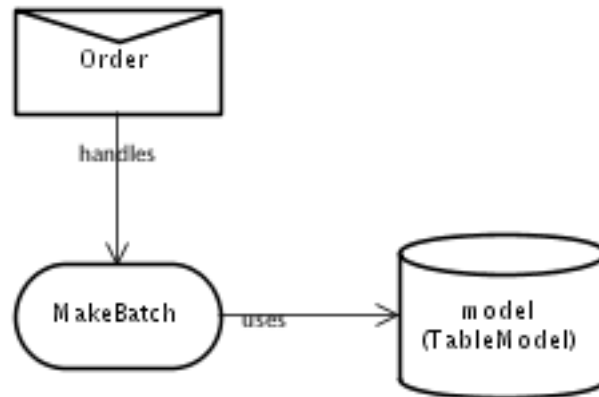

Figure 6-4: Behaviour Model

The plan for modelling the table behaviour during batch execution is shown below.

```
package jsv.table;

public plan MakeBatch extends Plan {
    #handles event Order oe;
    #uses data TableModel model;

    static boolean relevant(Order oe)
    {
        return oe.batchType.equals("AB");
    }

    context()
    {
        true;
    }

    #reasoning method
    body()
    {
        JigController jig1 = model.getJigController(TableModel.JIGPOINT1);
        JigController jig2 = model.getJigController(TableModel.JIGPOINT2);
        TableController table = model.getTableController();
        @waitFor(jig1.advance(JigController.XDECREASING));
        @sleep(10);
        for (int i = 0; i<oe.batchSize; i++) {

            // load an A
```

```
            jig1.setContents(JigController.STATE1);
            @sleep(50);

            //load a B
            jig1.setContents(JigController.STATE2);
            @sleep(50);

            //retract jig
            @waitFor(jig1.retract(JigController.XINCREASING));
            @sleep(10);

            //rotate table to joining position
            @waitFor(table.rotate(Math.PI));
            @sleep(10);

            //orient jig for joining
            @waitFor(jig1.rotate(Math.PI));
            @sleep(10);

            //advance jig
            @waitFor(jig1.advance(JigController.XDECREASING));
            @sleep(10);

            // join A to B
            jig1.setContents(JigController.STATE3);
            @sleep(50);

            //retract jig
            @waitFor(jig1.retract(JigController.XINCREASING));
            @sleep(10);

            //rotate table to the load/unload position
            @waitFor(table.rotate(Math.PI));
            @sleep(10);

            //orient jig for unloading/loading
            @waitFor(jig1.rotate(Math.PI));
            @sleep(10);

            //advance jig
            @waitFor(jig1.advance(JigController.XDECREASING));
            @sleep(10);

            // unload AB
            jig1.setContents(JigController.STATE0);
            @sleep(50);
        }
    }
}
```

In the interests of simplicity, jig advancement/retraction requires the direction of movement to be specified. However note that the direction of advancement is the same at both jig points. The reason for this is that at jig point 2, the jig's coordinate system has undergone a 180 degree rotation immediately prior to the jig advancement.

6.4.4.5 Scenario definition and execution

The scenario definition file for this example is as follows:

```
// We use the JACK Sim time management
<Include :dict "aos.jack.sim.time.Init__base" >

// Bring in visualisation
<Include :file "graphical.def" >

// Set simulation time.
<TimeInit :date "Sat, May 12, 2001, 15:15:00.0" >

// Declare a time dispatcher agent, who ensures that the time is
// not advanced while the application is busy.
<TimeDispatcherInit
    :name "time dispatcher"
>

<AgentInit :agent_type "jsv.table.Table" :name "table" >

// Declare a time source agent, who is responsible for advancing time
// in a synchronised manner with a time dispatcher agent.
<TimeSourceInit :name "time source" :dispatcher "time dispatcher"
    :verbose 0
    :realtime :false
    :delay 0
>
```

Note that the graphical definition file is included by the scenario definition file. The scenario is executed in the usual manner:

```
java aos.jack.sim.run.Loader scenario.def
```

Recall that execution of the scenario is managed by JACK Sim and that the agents are notified they can begin execution through the posting of a `RuntimeControl` event with a mode of `BEGIN`. The `Table` agent must have a plan to handle this event (`StartTable`. All that this plan does is to post an `Order` event, which triggers execution of the `Table` agent's behaviour model.

Appendix A: Example 1

This appendix contains the code for the original JACK application that was converted to a JACK Sim application in chapter 2.

```
/***************************************************************
 *
 *       Speaker1.agent (generated by the JDE)
 *
 ***************************************************************/

package eg0;

public agent Speaker1 extends Agent {
    #posts event Start sf;
    #handles event Start;
    #sends event Utterance ev;
    #uses plan Speak;

    public Speaker1(String name)
    {
        super(name);
    }

    public void converse()
    {
      postEvent(sf.start());
    }

}

/***************************************************************
 *
 *       Speaker2.agent (generated by the JDE)
 *
 ***************************************************************/

package eg0;

public agent Speaker2 extends Agent {
    #handles event Utterance;
    #sends event Utterance ev;
    #uses plan Respond;

    public Speaker2(String name)
    {
        super(name);
    }

}
```

Appendix A: Example 1

```
/****************************************************************
 *
 * Start.event (generated by the JDE)
 *
 ****************************************************************/

package eg0;

public event Start extends Event {

    #posted as
    start()
    {
    }

}

/****************************************************************
 *
 * Utterance.event (generated by the JDE)
 *
 ****************************************************************/

package eg0;

public event Utterance extends MessageEvent {
    public String utterance;
    public String speaker;

    #posted as
    utter(String s,String u)
    {
      speaker = s;
      utterance = u;
    }

}

/****************************************************************
 *
 *      Respond.plan (generated by the JDE)
 *
 ****************************************************************/

package eg0;

public plan Respond extends Plan {
    #handles event Utterance u1;
    #sends event Utterance uf;
    #uses interface Speaker2 speaker2;

    static boolean relevant(Utterance ev)
    {
        return true;
    }

    context()
    {
        true;
    }
```

```
    #reasoning method
    body()
    {
        String message = "Hello "+u1.speaker;
        System.out.println(speaker2.name()+": "+message);
        Utterance u2 = uf.utter(speaker2.name(),message);
        @reply(u1,u2);
    }

}


/***************************************************************
 *
 * Speak.plan (generated by the JDE)
 *
 ***************************************************************/

package eg0;

public plan Speak extends Plan {
    #handles event Start ev;
    #sends event Utterance uf;
    #uses interface Speaker1 speaker1;

    static boolean relevant(Start ev)
    {
        return true;
    }

    context()
    {
        true;
    }

    #reasoning method
    body()
    {
      for (;;)
      {
        String message = "Hello World";
        System.out.println(speaker1.name()+": "+message);
        Utterance u1 = uf.utter(speaker1.name(),message);
        @send("world",u1);
        @waitFor(u1.replied());
        Utterance u2 = (Utterance) u1.getReply();
        @sleep(10);
      }
    }

}
```

Appendix A: Example 1

```
/*************************************************************
 *
 * Driver.java
 *
 *************************************************************/

import eg0.Speaker1;
import eg0.Speaker2;

public class Driver
{
  public static void main(String[] args)
  {
    Speaker1 s11 = new Speaker1("ralph1");
    Speaker1 s12 = new Speaker1("ralph2");
    Speaker2 s2 = new Speaker2("world");
    s11.converse();
    s12.converse();
  }
}
```

Appendix B: Drawable Objects

This appendix describes the set of primitive drawable objects that are provided by the JACKSim infrastructure for specifying the appearance of an object in the visualisation. The `Drawable` interface is implemented by entities that can be displayed and thus drawn on a canvas. The following classes all implement the `Drawable` interface, and can be used in JACK Sim wherever drawables are required:

- `Arc`

- `Area`

- `CachedImage`

- `Colored`

- `Ellipse`

- `Figure`

- `Font`

- `Line`

- `Point`

- `Polygon`

- `Rectangle`

- `RoundRectangle`

- `TextLine`

- `Transform`

The following sections describe each class in more detail.

Arc

The `Arc` class extends `java.awt.geom.Arc2D.Double`. An `Arc` is a segment of an ellipse. Like an `Ellipse`, it contains fields that define an enclosing rectangle for the ellipse – the coordinates of the top left hand corner, and the width and height. Additionally, an arc has a start and an extent, which defines the segment of the ellipse that is required.

Field	Type	Description
x	double	The x coordinate of the upper left corner of the bounding box for the ellipse.
y	double	The y coordinate of the upper left corner of the bounding box for the ellipse.
width	double	The width of the bounding box for the ellipse.
height	double	The height of the bounding box for the ellipse.
start	double	The starting angle of the segment of the ellipse that defines the arc.
extent	double	The extent of the segment of the ellipse that defines the arc.

Table B-1: Initialisation attributes for an `Arc` object

Area

The `Area` class extends `java.awt.geom.Area`. An `Area` object performs certain binary CAG (Constructive Area Geometry) on other area-enclosing geometries, namely `Ellipse`, `Polygon`, `Rectangle`, `RoundRectangle` and `Area`. The supported CAG operations are add, subtract, intersect and exclusive or.

Field	Type	Description
shape	class	The `Shape` object that is to be used to construct the `Area` object.
modifiers	aggregation	An aggregation of `AreaModifier` objects that describe the CAG operations to be performed on the `Area` object.
filled	boolean	Specifies whether the interior of the `Area` object should be filled. It is initially set to `false`. An aggregation of `AreaModifier` objects that describe the CAG operations to be performed on the `Area` object.

Table B-2: Initialisation attributes for an `Area` object

The CAG operations that are to be performed on an `Area` object are specified using objects of the following types:

Type	Description
AddArea	Performs an add operation on an `Area` object.
SubtractArea	Performs a subtract operation on an `Area` object.
IntersectArea	Performs an intersect operation on an `Area` object.
XOR	Performs an exclusive or operation on an `Area` object.

Table B-3: The object types available for the specification of CAG operations

The above classes provide no fields for initialisation. However, they all extend the `AreaModifier` class, which provides a single `name` field for initialisation purposes. This field is of type `Area` and is used to specify the object that is to be applied to the base object using the specified area modifiers. Note that this object remains unchanged after the operation has been performed.

CachedImage

The `CachedImage` class is used to buffer an image. In situations where an image is very large and/or requires a lot of processing, buffering the image can result in a significant reduction in processing time.

Colored

The `Colored` class is used to set the colour of a `Drawable` object.

Ellipse

The `Ellipse` class extends `java.awt.geom.Ellipse2D.Double`. An ellipse is defined in terms of its bounding box.

Field	Type	Description
x	double	The x coordinate of the upper left corner of the bounding box for the ellipse.
y	double	The y coordinate of the upper left corner of the bounding box for the ellipse.
width	double	The width of the bounding box for the ellipse.
height	double	The height of the bounding box for the ellipse.
filled	boolean	Specifies whether the interior of the ellipse is to be filled. The default is false.

Table B-4: Initialisation attributes for an `Ellipse` object

Figure

The `Figure` class defines an aggregation of `Drawable` objects, thus allowing the grouping of multiple drawable objects into a single drawable object. `Figure` implements `Drawable`, and so `Figure` objects can be incorporated into a `Figure` object.

Line

The `Line` class extends `java.awt.geom.Line2D.Double`. It represents a line segment in (x,y) coordinate space.

Field	Type	Description
x1	double	The x coordinate of the starting point for the line segment.
y1	double	The y coordinate of the starting point for the line segment.
x2	double	The x coordinate of the finishing point for the line segment.
y2	double	The y coordinate of the finishing point for the line segment.

Table B-5: Initialisation attributes for a `Line` object

Point

The `Point` class extends `java.awt.geom.Point2D.Double`. It defines a point representing a location in (x,y) coordinate space.

Field	Type	Description
x	double	The x coordinate of the point.
y	double	The y coordinate of the point.

Table B-6: Initialisation attributes for a `Point` object

Polygon

The `Polygon` class draws a path along `Point` objects.

Field	Type	Description
points	aggregation	The `Point` objects that define the polygon.
closed	boolean	Specifies whether a line is to be drawn from the last point to the first point.
filled	boolean	Specifies whether the interior of the polygon is to be filled. The default is false.

Table B-7: Initialisation attributes for a `Polygon` object

Rectangle

The `Rectangle` class extends `java.awt.geom.Rectangle2D.Double`. A rectangle is defined in terms of the location of its upper left corner, its width and its height.

Field	Type	Description
x	double	The x coordinate of the upper left corner of the rectangle.
y	double	The y coordinate of the upper left corner of the rectangle.
width	double	The width of the rectangle.
height	double	The height of the rectangle.
filled	boolean	Specifies whether the interior of the rectangle is to be filled. The default is false.

Table B-8: Initialisation attributes for a `Rectangle` object

RoundRectangle

The `RoundRectangle` class extends `java.awt.geom.RoundRectangle2D.Double`. A round rectangle is defined in terms of its bounding box and an arc specification for the corners.

Field	Type	Description
x	double	The x coordinate of the upper left corner of the bounding box for the rectangle.
y	double	The y coordinate of the upper left corner of the bounding box for the rectangle.
width	double	The width of the bounding box for the rectangle.
height	double	The height of the bounding box for the rectangle.
arcwidth	double	The width of the arc that is to be used to form the corners.
archeight	double	The width of the arc that is to be used to form the corners.
filled	boolean	Specifies whether the interior of the rectangle is to be filled. The default is false.

Table B-9: Initialisation attributes for a `RoundRectangle` object

TextLine

The `TextLine` class is used to draw a line of text.

Field	Type	Description
attribute	String	The text to be displayed.
string	String	The text to be displayed if `attribute` is `null`.
x	double	The x coordinate of the start of the text line.
y	double	The y coordinate of the start of the text line.
font	Font	The `Font` class to use for the drawing of the text line.
real_font	java.awt.Font	The Java font class to use for the drawing of the text line.

Table B-10: Initialisation attributes for a `TextLine` object

Font

The `Font` class specifies details of the `Font` that is to be used when a `TextLine` is drawn.

Field	Type	Description
name	String	The name of the font to be used.
style	int	The style of the font.
size	int	The size of the font.

Table B-11: Initialisation attributes for a `Font` object

Transform

The `Transform` class provides data members that will contain a `Drawable` object and the values of any translation, rotation or scaling operations to be applied to the object during program execution. As a transform is a drawable object, a transform object can be the drawable member of a transform object.

Field	Type	Description
drawable	class	The contained `Drawable` object.
label	String	The name of the `Transform` object (if it is nested).
x	double	The x coordinate of the `Drawable` object.
y	double	The y coordinate of the `Drawable` object.
theta	double	The rotation to be applied to the `Drawable` object.
scale	double	The scaling factor to be applied to the `Drawable` object.

Table B-12: Initialisation attributes for a `Transform` object

References

Jarvis J., Fletcher, M., Howden, N., Ronnquist R. and Lucas, A., Human Variability in Computer Generated Forces – Application of a Cognitive Architecture for Intelligent Agents. In Proceedings of SimTecT 2005, Melbourne, 2005.

Jarvis J., Ronnquist R., McFarlane D. and Jain L., A Team-Based Holonic Approach to Robotic Assembly Cell Control. Journal of Computer and Network Applications, in press

Kreutzer W., System Simulation Programming Styles and Languages. Addison Wesley, 1986.

Index

standard time manager 23
standard.def 23

T
Teams 23
time management 7, 21
time management infrastructure 9
TimeConsoleInit object 29
    attributes 30
TimeControl event 40
TimeDispatcher agent 9, 21, 27
TimeDispatcherInit object 27
    attributes 27
TimeInit object 27
    attributes 27
TimeManaged interface 9, 21, 40
TimeRelayInit object 21, 28
    attribute 28
TimeSource agent 9, 21, 25
TimeSourceInit object 25
    attributes 25
TimeSyncManaged interface 9, 11, 21

U
Updater agent 57

V
visualisation 21
VisualsControl view 58