JACK Intelligent Agents®
Agent Manual

Copyright

US Government Restricted Rights

Trademarks

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

1.1 Background

JACK Intelligent Agents® (JACK) is an *Agent Oriented* development environment built on top of and integrated with the Java programming language. It includes all components of the Java development environment as well as offering specific extensions to implement agent behaviour. JACK's relationship to Java is analogous to the relationship between the C++ and C languages. C was developed as a procedural language and subsequently C++ was developed to provide programmers with object-oriented extensions to the existing language. Similarly, JACK has been developed to provide agent-oriented extensions to the Java programming language. JACK source code is first compiled into regular Java code before being executed.

In the same way that object-oriented programming introduces a number of key concepts that influence the entire logical and physical structure of the resulting software system, so too does agent-oriented programming. In agent-oriented programming, a system is modelled in terms of *agents*. These agents are autonomous reasoning entities capable of making pro-active decisions while reacting to events in their environment.

1.2 Agent Oriented Concepts

Agent oriented programming is an advanced software modelling paradigm that arose from research in distributed artificial intelligence. It addresses the need for software systems to exhibit rational, human-like behaviour in their respective problem domains. Traditional software systems make it difficult to model rational behaviour, and often programs written in these systems experience limitations, especially when attempting to operate in real-time environments.

Agent oriented programming is highly suited to many application areas, including distributed business systems, command and control, intelligent appliances and simulation. Although still young and under development, it has already shown particular promise in a variety of distributed problem solving tasks such as fleet organisation, air traffic management and air combat simulation. Because it offers such a modular and elegant solution to many of the problems faced in reactive processing, agent-oriented programming is ideally suited to these environments.

The *Agent Oriented* model follows the same underlying principle as *Object Oriented* programming – that reliable and scalable development can be enhanced by encapsulating the desired behaviour in modular units which contain all the definitions and structures required for them to operate independently. Agents extend the concept of encapsulation to include a representation of behaviour at a higher level than object-oriented approaches.

1.2.1 What is an Agent?

The term *agent* is widely used to describe a range of software components, varying in capability from procedural wizards, found in popular desktop applications, to information agents that are used to automate information search and retrieval, and, finally, to intelligent agents capable of reasoning in a well-defined way. The agents used in JACK are *intelligent agents*. They model reasoning behaviour according to the theoretical **B**elief **D**esire **I**ntention (BDI) model of artificial intelligence.

Following the BDI model, JACK intelligent agents are autonomous software components that have explicit goals to achieve or events to handle (desires). To describe how they should achieve these desires, BDI agents are programmed with a set of plans. Each plan describes how to achieve a goal under varying circumstances. Set to work, the agent pursues its given goals (**desires**), adopting the appropriate plans (**intentions**) according to its current set of data (**beliefs**) about the state of the world. This combination of desires and beliefs initiating context-sensitive intended behaviour is part of what characterises a BDI agent.

A JACK agent is a software component that can exhibit reasoning behaviour under both pro-active (goal directed) and reactive (event driven) stimuli. Each agent has:

- a set of beliefs about the world (its data set),
- a set of events that it will respond to,
- a set of goals that it may desire to achieve (either at the request of an external agent, as a consequence of an event, or when one or more of its beliefs change), and
- a set of plans that describe how it can handle the goals or events that may arise.

When an agent is instantiated in a system, it will wait until it is given a goal to achieve or experiences an event that it must respond to. When such a goal or event arises, it determines what course of action it will take. If the agent already believes that the goal or event has been handled (as may happen when it is asked to do something that it believes has already been achieved), it does nothing. Otherwise, it looks through its plans to find those that are relevant to the request and applicable to the situation. If it has any problems executing this plan, it looks for others that might apply and keeps cycling through its alternatives until it succeeds or all alternatives are exhausted.

Thus, an agent can be thought of as analogous to a person with access to a Procedures Manual. The Procedures Manual (set of plans) describes the steps that the agent should take when a certain event arises or when it wants to achieve a certain outcome. At first glance, this may seem like ordinary Expert System behaviour – with all the limitations that this implies. However, the crucial difference in agent-oriented systems is that the agent is able to be programmed to execute these plans just as a rational person would. In particular, it is able to exhibit the following properties associated with rational behaviour:

Goal-directed focus – the agent focuses on the objective and not the method chosen to achieve it.

Real-time context sensitivity – the agent will keep track of which options are applicable at each given moment, and make decisions about what to try and retry based on present conditions.

Real-time validation of approach – the agent will ensure that a chosen course of action is pursued only for as long as certain maintenance conditions continue to be true.

Concurrency – the agent system is multi-threaded. If new goals and events arise, the agent will be able to prioritise between them and multi-task as required.

1.2.2 Why program using Agents?

The capability of intelligent agents to autonomously perform simple tasks has aroused much interest. The key characteristics that make them attractive are their:

- ability to act autonomously,
- high-level representation of behaviour – a level of abstraction above object-oriented constructs,
- flexibility, combining pro-active and reactive behavioural characteristics,
- real-time performance,
- suitability for distributed applications, and
- ability to work co-operatively in teams.

JACK agents are highly suited to the development of time and mission-critical systems, as the BDI approach provides for the verification and validation of the model. The agent's goals may include keeping human users informed of what the agent is trying to achieve, what its current intentions are, and what progress it has been able to make. Giving BDI agents pre-compiled plans is a method of ensuring predictable behaviour under critical operational conditions, and of ensuring performance.

1.3 The Components of JACK

1.3.1 The JACK Agent Language

The JACK Agent Language is the actual programming language used to describe an agent-oriented software system. The JACK Agent Language is a super-set of Java – encompassing the Java syntax while extending it with constructs to represent agent-oriented features.

Each of the Java extensions that are included in JACK, along with their expected usage and semantic behaviour, are described in detail in the following chapters.

1.3.2 The JACK Agent Compiler

The JACK Agent Compiler pre-processes JACK Agent Language source files and converts them into pure Java. This Java source code can then be compiled into Java virtual machine code to run on the target system.

1.3.3 The JACK Agent Kernel

The JACK Agent Kernel is the runtime engine for programs written in the JACK Agent Language. It provides a set of classes that give JACK Agent Language programs their agent-oriented functionality. Most of these classes run behind the scenes and implement the underlying infrastructure and functionality that agents require, while others are used explicitly in JACK Agent Language programs, inherited from and supplemented with callbacks as required to provide agents with their own unique functionality.

1.4 Developing a JACK Application

An integrated development environment known as the JACK Development Environment is available for the development of JACK applications. Use of the JACK development environment is described in a separate manual. Alternatively, JACK applications can be developed from the command line using an editor of choice and explicitly invoking the JACK Agent compiler.

If the latter choice is used, the environment will need to be set up. The following section explains how this is done for the more common operating environments. The instructions that follow assume that all commands are invoked from a command line. Thus, under Windows or NT, a DOS window will need to be created.

1.4.1 Setting up your Environment

Before a JACK application can be compiled and run, ensure that the `PATH` and `CLASSPATH` variables are set correctly. `PATH` needs to be set so that `java` and `javac` are accessible. `CLASSPATH` needs to be set so that `java` and `javac` can access the JACK class files and the classes that are created.

The actual settings for `PATH` and `CLASSPATH` will depend upon where the java executables and JACK classes have been installed. It is assumed that:

1. The `PATH` variable is set correctly.
2. The `CLASSPATH` variable has not been set. (If it has, it may need to be extended. This procedure is operating system dependent.)

- DOS / Windows / NT

  In this example, it is assumed that the JACK classes are installed in `C:\aos\lib`. Within a session the `CLASSPATH` can be set by typing:

  ```
  set CLASSPATH=C:\aos\lib\jack.jar;.
  ```

  at a DOS prompt.

  This should be added to the `C:\AUTOEXEC.BAT` file so that this setting is permanently available. In Windows/DOS, this can be done using `SYSEDIT`. In NT it can be changed via `<Control Panel> -> <System> -> <Environment>`. Note that the `CLASSPATH` should be specified in lower case.

- UNIX

  In this example, it is assumed that the JACK classes are installed in `/aos/jack/lib`. Within a session, the `CLASSPATH` can then be set by typing:

  ```
  CLASSPATH=/aos/jack/lib/jack.jar:.
  export CLASSPATH
  ```

  If using `csh`, to make the setting permanent, add:

  ```
  setenv CLASSPATH=/usr/local/aos/jack/lib/jack.jar:.
  ```

  to the `.cshrc file`.

1.4.2 Source Code Creation

This can be achieved either using the JACK development environment, or an editor of choice. When developing a JACK application, source code will be created for some or all of the following entities:

- JACK *event*(s);
- JACK *plan*(s);
- JACK *agent*(s);
- JACK *capability*(s);
- JACK *view*(s);
- JACK *beliefset*(s);

plus a Java class that contains the application `main()` function that is the entry point for the Java virtual machine and any other Java file required by this application.

The files that are created for these entities must have the same base name as the entity defined in the file. They may have an extension designating the type of JACK entity contained, or simply a `.jack` extension.

Extension	Usage
`.jack`	Any JACK object definition.
`.agent`	JACK agent definition.
`.cap`	JACK capability definition.
`.plan`	JACK plan definition.
`.event`	JACK event definition.
`.bel`	JACK beliefset definition.
`.view`	JACK view definition.
`.java`	Java class or interface definition.

Table 1-1: JACK file extensions

1.4.3 Compilation

Assuming that all the source files are in your current directory, the application can be compiled by invoking `JackBuild`:

```
java aos.main.JackBuild
```

This runs the `JackBuild` utility which in this invocation compiles all of the JACK source files in the current directory into Java source. The Java compiler is then automatically invoked on all Java source files in the current directory. `JackBuild` recognises JACK files by their file name extensions as listed in the previous section.

Without arguments, the `JackBuild` utility assumes that all files in the directory belong to the application. If the application is organised into subdirectories, it can be compiled by invoking `JackBuild` from the parent directory as follows:

```
java aos.main.JackBuild -r
```

The `-r` option recursively enters subdirectories to compile code.

Refer to *Appendix A* for more information about `JackBuild`.

1.4.4 Running a JACK Application

If one assumes that the `main()` method was in a file called `Test.java`, the compilation process will have produced a file called `Test.class`. The application can then be run from the directory containing `Test.class` by typing

```
java Test
```

Note: It is possible to configure Windows so that files with the extension `.class` are runnable by point-and-click.

The programmer can specify command line arguments for use within an application. Note that there are some command line arguments which are processed internally by JACK. In particular, a DCI network from the command line can be set up (this is discussed further in the chapter on *Inter-agent Communication*). If such arguments are present, the method `aos.jack.Kernel.init()` must be used to process these arguments **before** any user specified command line arguments are processed. `init()` extracts and processes command line arguments intended for JACK, then returns a modified argument list containing the remaining user defined arguments. This list can then be accessed in the normal way, as shown in the example below:

```
public class Example
{
  // The user supplies a single numeric command line argument
  // in addition to those which will be handled by JACK
  public static void main(String args[])
  {
    args = aos.jack.Kernel.init(args);

    int snum = Integer.parseInt(args[0]);
     :
     :
  }
}
```

All the standard Java command options are available. Some JACK functionality such as the interaction diagram and debugging is configurable via the system properties file. The `-D` option can be used in these circumstances to set properties. For more details on the interaction diagram and debugging refer to the *Tracing and Logging Manual*.

1.4.5 Debugging a JACK Application

There are a number of tools available to assist the developer during application development. These range from graphical tracing tools to logging tools which provide a detailed trace of system execution. These tracing and logging tools are described in the *Tracing and Logging Manual*.

2 JACK Agent Language Overview

2.1 The JACK Agent Language

The JACK Agent Language is built on top of Java. Like C++, the JACK Agent Language does more than extend the functionality of Java – it also provides a framework to support an entirely new programming paradigm. The JACK Agent Language is an *Agent Oriented* programming language and is used for implementing Agent Oriented software systems.

The JACK Agent Language extends Java to support Agent Oriented programming:

- It defines new base classes, interfaces and methods.
- It provides extensions to the Java syntax to support new agent-oriented classes, definitions and statements.
- It provides semantic extensions (runtime differences) to support the execution model required by an agent-oriented software system.

All the language extensions are implemented as Java plug-ins. This makes the language as extensible and flexible as possible. Flexibility is important in the JACK Agent Language because it facilitates ongoing research into agent-oriented programming. Developers may, for example, want to investigate how different beliefset implementations affect agent performance. Because the beliefset component is supplied as a plug-in, this can be altered with minimal changes to the JACK development environment. All that is required is to replace the beliefset implementation in the kernel package.

Each of the JACK Agent Language extensions is strictly typed. This minimises implicit type casting and the opportunity for programmer error. Strict typing also allows for more efficient program compilation by the JACK Agent Compiler.

2.1.1 Class, Interface and Method Extensions

The JACK Agent Language introduces five main class-level constructs. These constructs are:

Agent – The agent construct is used to define the behaviour of an intelligent software agent. This includes capabilities an agent has, what type of messages and events it responds to and which plans it will use to achieve its goals.

Capability – The capability construct allows the functional components that make up an agent to be aggregated and reused. A capability can be made up of plans, events, beliefsets and other capabilities that together serve to give an agent certain *abilities*. An agent can, in turn, be made up of a number of capabilities, each of which has a specific function attributed to it.

BeliefSet – The beliefset construct represents agent beliefs using a generic relational model. It has been specifically designed so that a beliefset can be queried using **logical members**. Logical members are like normal data members, except that they follow the rules of logic programming (as in programming languages like Prolog).

View – The view construct allows general purpose queries to be made about an underlying data model. The data model may be implemented using multiple beliefsets or arbitrary Java data structures.

Event – The event construct describes an occurrence in response to which the agent must take action.

Plan – An agent's plans are analogous to functions. They are the instructions the agent follows to try to achieve its goals and handle its designated events.

For a detailed description of each of these extensions, including the specific interfaces and methods provided with them, refer to the appropriate chapters in this manual.

2.1.2 Syntactic Extensions

JACK Agent Language provides a number of variations and extensions to the standard Java syntax. These extensions exist purely to support the syntactic and semantic differences between object-oriented and agent-oriented programming.

An example piece of code written in the JACK Agent Language to implement an agent plan is given below. The syntactic elements that are unique to JACK Agent Language have been highlighted in bold. All other elements follow normal Java syntax.

```
plan MovementResponse extends Plan
{
   #handles event RobotMoveEvent moveResponse;
   #uses agent implementing RobotInterface robot;

   static boolean relevant (RobotMoveQEvent ev)
   {
      return (ev.ID == RobotMoveQEvent.REPLY_SAFE ||
          ev.ID == RobotMoveQEvent.REPLY_DEAD);
   }

   body()
   {
      if (moveResponse.ID==REPLAY_SAFE)
      {
         System.err.println("Robot Safe to Move");
         robot.updatePosition(moveResponse.Lane,
         moveResponse.Displacement);
      }
      else
      { // robot didn't make it
         System.err.println("Robot is Dead");
```

```
            robot.die();
        }
    }
}
```

In this example, the plan being defined inherits its core functionality from the JACK Agent Language class: `Plan`. It then identifies how the plan will be used through a number of JACK Agent Language *plan declarations*. JACK Agent Language declarations are each preceded with a `#` symbol to distinguish them from Java syntax elements, for example `#handles event` and `#uses agent implementing`.

The `#handles event` declaration identifies the goal or event to which this plan will respond. The `#uses agent implementing` declaration constrains the agent(s) that can use this plan. Only those agents that present the specified interface (`RobotInterface`) can include this plan.

The block of code in this example contains only regular Java code. The JACK Agent Language however provides its own statements that can be used when required. These statements are known as *reasoning method statements*, and are identified by a preceding @ character. Reasoning methods specify operations that are only meaningful in agent-oriented programming, such as posting an event or waiting until the agent acquires a particular belief. They are called reasoning method statements because you can only use them in a reasoning method belonging to a plan. These reasoning methods describe the reasoning and behaviour that an agent undertakes when it executes an instance of that plan.

Hence, JACK Agent Language extends Java syntax at three levels:

the class definition level, providing a set of classes that agent-oriented constructs such as agent, plans and events can inherit from;

the declaration level, providing a set of statements that identify relationships between the classes mentioned above; and

the statement level, providing a set of statements that can operate on JACK Agent Language-specific data structures.

For more information on the JACK Agent Language syntax extensions to Java, see the following chapters.

2.1.3 Semantic Extensions

Because agent-oriented programming follows a different modelling paradigm to object-oriented programming, there are significant differences in how programs written in each language behave at runtime.

The JACK Agent Language extends the Java execution engine in the following ways:

- Multi-threading is built into the kernel and removed from programmer control. JACK Agent Language programs should not be written with explicit multi-threading. This is transparently provided by the JACK Agent Language.
- Execution follows the model of agents processing a set of plans and having access to a number of beliefset relations. The agent executes these plans in *tasks* to handle events when they arise, consulting its beliefset relations where necessary. These plans may initiate subtasks, which may in turn initiate their own subtasks if the agent requires a sizeable and complex response.
- A new data structure called a *logical member* is introduced. Unlike normal members, logical members have an unknown (unbound) value until this value has been determined. Once determined, the value cannot change.
- The agent's beliefset can be queried using logical members by attempting to **unify** the logical member with a desired result. If the query succeeds, the logical member contains the desired value. If not, it remains unbound (unknown) and can be used later.

A detailed description on the semantic consequences of each new JACK Agent Language construct can be found in the appropriate chapters of this manual (*Agents*, *Capabilities*, *Events*, *Plans*, *Beliefset Relations* and *Views*).

2.2 JACK Agent Language Summary

The JACK Agent Language is closely related to Java and extends the regular Java syntax. It allows the programmers to develop the components that are necessary to define BDI agents and their behaviour. These functional units are:

Agents – which have methods and data members just like objects, but also contain capabilities that an agent has, beliefset relations that they can use to store beliefs, descriptions of events that they can handle and plans that they can use to handle them.

Capabilities – which serve to encapsulate and aggregate functional components of the JACK Agent Language for use by agents. Capabilities can include events, plans, beliefsets or even other capabilities.

Beliefsets – which are used to store beliefs and data that the agent has acquired. Agents can also use regular Java data structures for storing information, but an advantage of a beliefset is that it will generate events when particular changes are made. This can be used to initiate further action within the agent and hence make it more responsive to its own internal state. Also beliefsets can be queried using logical members.

Views – which provide a powerful way of modelling any data in a way easily manipulated by JACK.

Events – which identify the circumstances and messages that it can respond to.

Plans – which are executed in response to these events.

Each of the events, plans and beliefsets used are implemented as Java classes. They inherit certain fundamental properties from a base class and extend these base classes to meet their own specific needs. The base classes are defined within the kernel and form the 'glue' that holds a JACK agent-oriented program together. However, the JACK Agent Language is more than just a specific organisation of Java objects and inheritance structures – it provides its own extended syntax, which has no analogous representation in Java.

JACK Agent Language constructs can be categorised as follows:

- Classes (types)
- Declarations (#-declarations)
- Reasoning Method Statements (@-statements)

In addition, each of the JACK Agent Language classes supplies a number of normal Java members and methods that can be made use of in JACK programs.

The constructs available in each of these categories are listed in the following sub-sections.

2.2.1 JACK Agent Language Classes

These classes define functional units within JACK. The functional units are implemented as Java classes, with their agent-oriented properties embedded within the class as private methods. They also provide some base members and methods that JACK Agent Language programmers can use. Each JACK Agent Language type and the base members and methods it provides are listed below:

`Agent` – which models the main reasoning entities in JACK.

`Capability` – which aggregates functional components (events, plans, beliefsets and other capabilities) for agents to make use of.

`Event` – which models occurrences and messages that these agents must be able to respond to. Events may arise externally from messages from other agents, or internally as a consequence of one of the agent's own actions, or in response to one of its internal goals.

`Plan` – which models procedural descriptions of what an agent does to handle a given event. All the action that an agent takes is prescribed and described by the agent's plans.

`BeliefSet` – which models an agent's knowledge as beliefs that follow either a closed world (omniscient) or open world (with unknown) semantics. Beliefsets represent an agent's beliefs as first order relational tuples, and maintain their consistent logical and key constraints.

Note that a `View` does not have a base class.

2.2.2 JACK Agent Language Declarations (#-Declarations)

JACK Agent Language #-declarations define agent-oriented properties of a JACK Agent Language type, or relationships and dependencies that exist between JACK Agent Language types. They are used to specify relationships between classes in a JACK program. For example, JACK Agent Language declarations specify which plans an agent uses and which event a plan handles.

The range of #-declarations available in JACK Agent Language are listed below:

`#chooses for event` – is used in *plan definitions* to identify a plan that allows the agent to reason about which applicable plan instance it will execute for a given event occurrence. This is in contrast to normal plans, which describe how the agent will respond to a given event instance when that plan is selected. Because it provides the agent with the ability to reason about how it will respond to (reason about) a given event, plans that contain a `#chooses for event` declaration are known as *meta-level reasoning plans*.

`#complex query` – is used in *view and beliefset definitions* to define a complex query.

`#exports data` – is used in *capability definitions* to declare that a user-defined data structure or a JACK beliefset is exported from the capability so that it is accessible from its parent capability. The export statement can also be used to make a user-defined data structure or JACK beliefset available at the agent level, and hence accessible from other capabilities in the agent.

`#function query` – is used in *view and beliefset definitions* to define a function query.

`#handles event` – is used in *agent definitions* and *plan definitions*. In both cases, it identifies an event that an agent or plan handles. Event handling is declared in the agent that uses a plan and the plan itself. This allows JACK to perform type checking and ensure that the agent has at least one plan to handle every event that it claims to handle, and conversely that an agent doesn't have access to a plan that is beyond its stated event-handling responsibilities.

`#handles external event` – is used in *capability definitions* to declare that there are plans within the capability that handle events of a given type.

`#has capability` – is used in *agent definitions* and *capability definitions*. In an *agent definition,* it gives the agent access to all of the functional components enclosed by the capability. In a *capability definition*, it declares the use of an inner capability.

`#imports data` – is used in *capability definitions* to declare a user-defined data structure or JACK beliefset that is to be used within the capability, but is brought in from the enclosing capability or agent.

`#indexed query` / `#linear query` – are both used in *beliefset definitions* to specify how a relation can be queried. This declaration indicates which parameters it expects to be supplied in the query and which parameters it will need to return (through binding logical members) when the query succeeds.

`#key field` / `#value field` – are both used in *beliefset definitions* to specify the fields that belong to a relation. *Key fields* identify an object that the relation describes, while *value fields* identify attributes of that object.

`#posted as` – is used in an *event definition* to define a posting method for that event. An event's posting methods are used to construct instances of the event when the event needs to be posted. Multiple posting methods allow the event to be posted in different ways (even having different parameters) in different circumstances.

`#posts event` – is used in *agent definitions*, *capability definitions* and *plan definitions* to identify events that agent or plan is capable of posting. Neither an agent nor a plan can create an instance of an event unless they declare that they can post (`#posts event`) or send (`#sends event`) it. Whether an event is posted or sent depends on its type.

`#posts external event` – is used in *capability definitions* to declare that there are plans (or Java code) within this capability that post events of a given type.

`#posted when` – is added to an event definition to specify the condition which must arise for the event to be posted automatically.

`#private data` / `#agent data` / `#global data` – are used in *agent definitions* and *capability definitions* to identify a user-defined data structure or JACK beliefset that the agent can use to store information. A `private data` instance is unique to each agent, and hence can be read or modified by that agent and that agent only. An `agent data` instance is available to all agents of a given agent type (i.e. instances of the same Agent class). A `global data` instance is available to all agents in a given process. Agents should only modify the data that appears in their private user-defined data structures or in their private JACK beliefsets.

`#propagates changes` – marks that a beliefset may be a source beliefset in a team belief connection, and it provides an implementation of the connection dynamics, so that changes to the beliefset are propagated correctly. Belief propagation is only available when using JACK Teams. Refer to the *Teams* manual for more details.

`#reasoning method` – is used in *plan definitions* to define methods that an agent may execute when it runs this plan. Reasoning methods are different from normal Java methods in that they execute as finite state machines, and may succeed or fail, depending on whether the agent can complete each statement that they contain. The top-level reasoning method is called `body()`. This is the only reasoning method that must be present in all plans and is **not** preceded by an `#` symbol. The `body()` method can call other reasoning methods if they have been included amongst the plan's `#reasoning method` declarations. It can also initiate the execution of other tasks and subtasks by executing JACK Agent Language statements that post new events. Examples of such statements are: `@subtask`, `@maintain` and `@achieve`.

`#reasoning method pass` / `fail` – is used in *plan definitions* to identify processing that should occur after an instance of the plan has either succeeded or failed. When either of these methods is present in a plan, that plan's execution does not end when the plan has succeeded or failed. First, the relevant `pass()` or `fail()` method is executed.

`#sends event` – is used in *agent definitions* and *plan definitions* to identify message events that the agent or plan is capable of sending to other agents. Neither an agent nor a plan can send a message event to another agent unless it include a corresponding `#sends event` declaration for that class of message event.

`#set behavior` – is used in BDI event definitions to declare how an agent processes an instance of this event when it arises. Options that can be configured include whether the agent will re-try the event if an attempted plan fails and whether or not the agent can reason about which plan it executes in response to the event.

`#set transport` – is used to signal which transport format is to be used for message events.

`#uses agent implementing` – is used in *plan definitions* to declare that any agent using these plans must implement the required interface. Interfaces allow multiple agents to use a given plan.

`#uses data` / `#reads data` / `#modifies data` – are used in *plan definitions* to identify user-defined data structures and JACK beliefsets that the plan can access. If the plan reads a data instance it should only perform queries on it. If the plan modifies a data instance, it can read and modify the data. Similarly, if the plan uses a data instance, it can read and modify the data.

`#uses data` – is used in *view definitions* to declare that a view requires data of a specified type. It can also be added to *event* and *plan definitions* to provide access to a belief structure.

`#uses interface` – is used in plans to declare that one of the enclosing capabilities (or the enclosing agent) implements the required interface. Note that this declaration supersedes `#uses agent implementing` for most purposes.

`#uses plan` – is used in *agent* and *capability definitions* to specify that an agent or capability includes this plan in its set of available plans. This is a containment relationship.

`#uses taskManager` – is used in *agent definitions* to specify how an agent shares its processing capacity between active tasks. JACK provides a simple task manager, which persists with the current task until completion, and a round robin task manager which allows each task to execute a specified number of plan steps before switching to another waiting task.

2.2.3 Reasoning Method Statements (@-Statements)

Reasoning statements are JACK Agent Language specific statements that can only appear in *reasoning methods*. They describe actions that the agent can perform to execute behaviour. Steps such as posting events, sending messages to other agents or waiting until a particular condition is true are expressed using reasoning method statements.

Each reasoning method statement is listed below.

`@action` – is a reasoning statement that can be used if there is a need to include a lengthy computation within a plan.

`@achieve` – tells the agent to make a certain condition true. If the condition is already true, the agent does nothing, but if not a `BDIGoalEvent` will be posted. Typically, this event will initiate the execution of a plan to make the condition hold. The current plan is suspended while this plan is being executed, and the `@achieve` statement succeeds or fails based on whether the plan that is executed succeeds or fails.

`@determine` – tells the agent to find a binding for a logical condition that causes a `BDIGoalEvent` to succeed. Typically, the logical condition will contain one or more beliefset queries. For each binding that these queries return, the agent posts a `BDIGoalEvent`. As soon as one of these `BDIGoalEvents` succeeds, the `@determine` statement succeeds. If all bindings are tried and fail, the `@determine` statement fails.

`@insist` – tells the agent to ensure that a certain condition holds true. Like the `@achieve` statement, the agent will do nothing if the condition is already true and will post a `BDIGoalEvent` otherwise, but when the `BDIGoalEvent` has been handled, the agent re-tests the condition to ensure that it holds. If the condition holds, the `@insist` statement succeeds, but if not the `BDIGoalEvent` is posted again. The agent will keep testing the condition and posting the `BDIGoalEvent` until the specified condition is satisfied or the event processing fails.

`@maintain` – is used in a reasoning method to specify that a *subtask* be performed. However, while the agent is performing the subtask, it must ensure that a particular logical condition is never violated. If the logical condition is found to be false between plan steps, the subtask will fail immediately. Other than specifying a maintenance condition, the `@maintain` statement operates in exactly the same way as the `@subtask` statement (see below for details).

`@parallel` – allows concurrent sub-tasking of a set of statements within reasoning methods. The `@parallel` statement suspends execution of the calling plan while all enclosed statements are executed in parallel.

`@post` – posts an `Event` or `BDIFactEvent` within an agent. This initiates another task execution within the agent. The event is posted using one of its posting methods and is handled asynchronously by the current task execution thread.

`@reply` – sends a message event (`MessageEvent` or `BDIMessageEvent`) in response to another message event that the agent has received. It is only available in plans that are executed within a task initiated by the arrival of a message event. If the task execution was initiated by any other kind of event, the `@reply` statement will fail.

`@send` – sends a message event to another agent. Unlike the `@post` method which posts an event internally, an event is sent to another agent (potentially running in another process). The receiving agent then has the option of replying to the message. The sender has a mechanism for checking that a reply has been sent, and has methods for handling replies when they arrive.

`@sleep` – suspends execution of the task for a given period of time.

`@subtask` – posts an `Event` or `BDIFactEvent` , but instead of handling the event asynchronously in a separate task, the agent handles it synchronously as part of the same task.

`@test` – tests a condition. If the condition is true, it returns `true`. If the condition is false, it returns `false`. If the condition is unknown, it posts a `BDIGoalEvent` to find out whether it is true or false.

`@wait_for` – identifies a condition that the agent should wait for. The plan cannot proceed until this condition is true. The task is suspended, and waits until some other task performs an action that makes the condition true. To prevent the agent from waiting indefinitely, a timeout condition can also be specified.

2.2.4 Base Members and Methods

Each JACK Agent Language class offers a number of public members and methods that can be called in the user's programs. These base members and methods are listed below:

2.2.4.1 Agent Members

`Timer` *`timer`* – a public agent member that keeps the timer for this agent. This timer is used by the plan methods `elapsed()` and `after()`, and by the `@sleep` reasoning method statement.

2.2.4.2 Agent Methods

`finish()` – terminates the current agent instance. This is distinct from, and should not be confused with, the standard Java `finalize()` callback (used by the garbage collector to allow objects to take a final action before termination).

`String name()` – returns the full name of a given agent instance. An agent's full name takes the following form: *`local_name@portal_name`*, where:

- *`local_name`* is the name that identifies the agent at a given *portal* on the remote agent communications network. This is the name that was passed to the agent constructor (`super("`*`agent_name`*`")` call).
- *`portal_name`* is the name of the portal that the agent listens on and uses to communicate with other agents on the remote agent communication network.

`void postEvent()` – posts an event. The event is handled *asynchronously* by a separate task in the agent. This method behaves in the same way as the `@post` reasoning method statement. The `postEvent()` method allows Java code other than reasoning methods to initiate task execution within an agent by posting an event.

`boolean postEventAndWait()` – also posts an event. The event must belong to an event class that the agent has declared it can post (i.e. by identifying this class in one of its `#posts event` declarations). However, instead of posting the event asynchronously, the event is posted *synchronously* (as though it has been posted by a `@subtask` statement, for example), to be handled as a 'subtask' of the calling thread. The current execution thread stops what it is doing while the event is handled and waits for the event to either succeed or fail.

An attempt to call `postEventAndWait()` from a JACK thread (i.e. through a call chain originating in a plan) will be caught by the JACK kernel, which will issue a warning message. `PostEventAndWait` provides a means to post events synchronously from outside reasoning method execution (i.e. from a Java thread).

`void reply(MessageEvent` *`query`*`, MessageEvent` *`reply`*`)` – replies to a message event that an agent has received from another agent. It takes two message events as arguments:

- *`query`*, which is the message event to which the agent is responding; and
- *`reply`*, which is the message event that should be returned to the sender of *`query`* as the agent's response.

An agent can reply to any message event that it receives. This method behaves identically to the `@reply` statement in the JACK Agent Language. Like `send()` and `postEvent()` , this method allows code outside plans to engage in inter-agent communication and to initiate task execution within an agent.

`void send(String` *`to`*`, MessageEvent` *`message`*`)` – sends a message event to another agent. It takes a message event and the (full) name of the agent to which it should be sent. Partial names resolve to the same portal as the sender.

2.2.4.3 Event Members

`String from` – appears in message events only, and enables identification of the agent that sent the message carried by this message event.

`String message` – appears in message events only (`MessageEvents` and `BDIMessageEvents`), and contains a message that will be passed to the application's Agent Interaction Diagram, so that each instance of this event can be identified when it appears in the diagram. The Agent Interaction Diagram in discussed in the *Tracing and Logging* Manual.

`String mode` – appears in `BDIGoalEvents` only and identifies the type of JACK Agent Language statement that posted the instance of this goal event (`@achieve`, `@insist`, `@test` or `@determine`).

2.2.4.4 Event Methods (for MessageEvents only)

`Cursor replied()`– appears in `MessageEvents` only, and enables the user to determine whether the agent has received at least one reply to a given message event instance. It returns a triggered cursor, which will test whether the given message event's reply queue is empty. Because the cursor is triggered, it can be used in a `@wait_for` statement to wait for message event replies to arrive.

`MessageEvent getReply()` – appears in message events only and allows the user to obtain a reference to each message event that has been returned to the agent as a reply to a given message event instance. To be a reply, a message event must have been sent by the destination agent for the original message, using the `reply()` method, from within the task that the destination agent was using to handle the original message event.

2.2.4.5 Plan Members

`Agent agent` – identifies the agent that this plan belongs to. Whenever an instance of a plan is created, this member is assigned a reference to the agent that created the plan instance.

Note: The preferred way to access the agent's methods and members is through the `#uses interface` statement described in the *Plans* chapter.

2.2.4.6 Plan Methods

`PlanInstanceInfo getInstanceInfo()` – a callback that has been supplied so that agents can perform *meta-level reasoning* about plan instances. It provides a base class for recording information about plan instances. This base class can be extended and accessed in plans that perform meta-level reasoning. For example, one application may extend the `PlanInstanceInfo` class so that it assigns a *precedence* rating to each plan instance. This precedence rating may simply be a constant, or it may be calculated in some way. The agent can choose between its available plan instances on the basis of this precedence rating, taking the highest precedence plan it has available as its first choice.

`static boolean relevant()` – used to determine if a plan is relevant to the actual event. When the agent is looking for a plan to execute in response to an event, it executes the plan's `relevant()` method to determine whether the given plan is relevant. A plan is relevant if, and only if, it handles the given event and the event's parameters match the pattern specified in the plan's `relevant()` method. Since events are polymorphic, this can be used to weed out those plans that can handle this event type, but not this particular instance of the event type. Unless the event's parameters satisfy the `relevant()` method, the agent will deem the plan not relevant to this event.

`context()` – used to determine if the plan should be executed in the current context. The context specifies a logical condition that must be satisfied if the plan is to be applicable for handling a given event in the current situation. Context often refers to values in an agents knowledge base, which are its beliefs about the state of the world. When the agent needs to handle an event, it looks for a *plan instance* that is applicable to this event. A plan instance is applicable if it satisfies the plan's context. Typically, the `context()` method will include logical members in beliefset queries. When an applicable instance of the plan is found, it indicates that the query found a tuple and bound the logical member(s) to its value(s). If there is more than one way to satisfy a `context()` method's logical expression, there will be multiple instances of the plan that are applicable. One applicable instance will be generated for each set of bindings that satisfy the `context()` condition.

`body()` – the plan's main or 'top-level' reasoning method. It describes what it is that the agent actually does when it executes an instance of this plan.

2.2.4.7 BeliefSet Methods

`add()` – adds new tuples to an agent's private beliefset, or modifies its existing tuples by supplying updated information.

`remove()` – removes tuples from an agent's beliefset.

`public int nFacts()` – returns the number of facts (tuples) that are currently held in a given beliefset. For *Open World* relations, this is the sum of both positive (true) and negative (false) beliefs, while for *Closed World* relations this is only the number of positive (true) beliefs.

2.2.4.8 Capability Methods

`void postEvent(Event e)` – the `postEvent()` method is used to post events within capability code. This method is actually just a convenience method that refers to `getAgent().postEvent()`.

`Agent getAgent()` – this method is called on a capability instance to return the containing agent.

`void autorun()` – this method can be overridden in order to provide some initialisation when the capability is actually brought into being.

3 Agents

3.1 Introduction

The Agent class embodies all the functionality associated with a JACK intelligent agent. To define agents, extend this class, adding members and methods that are applicable to the agents current application domain.

3.2 Agent Definition

Agent definitions take the form shown below:

```
agent AgentType extends Agent [implements Interface]
{
   // JACK Agent Language statements specifying containment
   // relationships.
   // These are described in the following sub-sections.
}
```

Each component of this definition is explained in the following table:

Syntax Term	Description
`agent`	A JACK Agent Language keyword used to introduce an Agent definition.
`AgentType`	The name of your derived `Agent` class (which can not be further subclassed).
`extends Agent`	This part of the statement plays the same role as in Java – it indicates that the agent being defined inherits from a JACK Agent Language base class called `Agent`. The `Agent` base class implements all the underlying methods that provide an agent's core functionality.
`[implements Interface]`	This part of an agent definition is optional. When present, it states that the agent *implements a given Java interface*. Java interfaces are like classes that consist of method prototypes without code. When an agent implements an interface, it provides code to implement each of these methods.

Table 3-1: Components of an Agent definition

3.3 Agents and Interfaces

The optional *implements Interface* component in an agent definition is important when it comes to writing portable JACK Agent Language programs that allow for code re-use. Interfaces provide a common ground between agents that allows them to share plans.

When an agent executes a plan, this plan will often call ordinary Java methods. It is important to remember that when this occurs these methods must be included in the agent definition, not the plan definition. Therefore, any agent that uses this plan must include the defined methods or the plan will not be able to run properly.

This means that the JACK Agent Language places restrictions on which agents can use what plans. So that this restriction can be observed in a modular way, the JACK Agent Language allows these dependencies to be packaged up into a Java *Interface*. Any agent that wishes to include this plan must declare that it *implements this interface*. If an Agent class implements the interface, it provides all the methods necessary to run the plan.

3.4 Agent Declarations

An agent should fully describe the functionality it implements via JACK Agent Language definitions. In general, this definition needs to include the following conceptual statements:

- *BeliefSets and Views* which the agent can use and refer to.
- *Events* (both internal and external) that the agent is prepared to handle.
- *Plans* that the agent can execute.
- *Events* the agent can post **internally** (to be handled by other plans).
- *Events* the agent can send **externally** to other agents.

These definitions are handled by statements that occur at the field or member level of an agent definition. While there is no restriction on where they appear in Jack code, by convention these definitions appear before the definitions of any regular Java data members and methods that the agent may contain.

An example agent template showing **some** of the declarations that can appear in an agent appears below:

```
agent AgentType extends Agent [implements InterfaceName]
{

   // Knowledge bases used by the agent are declared here.

   #private data BeliefType belief_name(arg_list);

   // Events handled, posted and sent by the agent are
   // declared here.

   #handles event EventType;
   #posts event EventType reference;
   #sends event EventType reference;

   // Plans used by the agent are declared here.
   // Order is important.

   #uses plan PlanType;

   // Capabilities that the agent has are declared here.

   #has capability CapabilityType reference;

   // other Data Member and Method definitions

}
```

Each JACK Agent Language agent declaration is described in more detail in the following sub-sections.

#handles event *EventType*

This statement identifies the *events* that the agent will attempt to respond to if they arise. By handling the event, the agent claims to have at least one plan available that it can execute when this event arises. These plans may not be relevant to all forms of the event or applicable in all circumstances, but the agent must know how to handle the event in at least some situations.

Because it is really claiming that the agent's plans can handle the event, the `#handles event` agent definition statement is analogous to a function prototype. It is an explicit statement with which the runtime can check for completeness rather than functional necessity. However, defining the events that an agent handles up front allows agents to be prototyped and helps ensure that sound design practices are followed.

Including the `#handles event` definition is also important to ensure that task processing takes place in the agent when the event occurs. If an agent receives an event that it does not handle, a runtime warning is generated and the event is not processed. By claiming to handle the event, the agent looks through its plans to find one that has a matching `#handles event` statement. A suitable plan might not be found, but at least the agent looks to make sure. Claiming to handle an event is like an employee claiming that a situation falls under their

responsibility: they take notice when it occurs and try to do something about it. Whether they succeed or not is another matter.

When an agent definition includes a statement of the following form:

```
#handles event EventType;
```

The agent claims that when an event of *EventType* occurs, it has a plan to handle it. This plan should be declared with the `#uses plan` declaration. How this event is processed depends on whether it is a BDI event or a normal event. *Behaviour Attribute* settings can also influence how events are handled and particularly what happens on plan failure.

Refer to the *Events* chapter for more details on how different event types are handled and how this behaviour can be customised.

#posts event *EventType* [*reference*]

This statement describes an *event* that the agent can post. Posting an event means that an agent creates an instance of the event and posts it internally (i.e. sends the event to itself).

The `#posts event` declaration identifies those events that the agent posts **explicitly**, not those that arise from actions of other agents or indirectly from the agent's own actions or changes in internal state. Therefore, it is usually used to declare that the agent posts events of the types `Event`, `BDIFactEvent` and `BDIGoalEvent`. For more information on these event classes, refer to the *Events* chapter.

When an agent claims that it posts an event, this event will only arise if it is explicitly generated in one of the agent's methods.

An agent definition contains a statement of this form to indicate that the agent has reasoning methods or code that explicitly causes an event of this type to arise.

Each term in the previous definition is described in the following table:

Term	Meaning
`#posts event`	Identifies that the agent can post events of the given type. The event is always posted internally, and hence needs to be handled by the agent's own plans.
EventType	Identifies the type of event to be posted.
[*reference*]	When present, JACK creates an agent member called *reference* which can be used to create events of *EventType* using its posting methods.

Table 3-2: Terms in a `#posts event` declaration

When an agent posts an event, it does so by calling the method `postEvent()` as shown below:

```
postEvent (event)
```

The event being posted must be constructed using one of the event's posting methods. This is described in the section on posting / sending events in the *Events* chapter.

Events posted in this way are handled *internally* by the agent. No other agent is affected by the posting process. Hence, they are like 'thoughts' or 'ideas' that the agent has. The agent essentially tells itself that this event has occurred and needs to be dealt with.

Agents can also send external events to be handled by other agents. These events are called *message events* and are described in the *Events* chapter.

#sends event *EventType* [*reference*]

This declares that the agent is able to send a message event to another agent. Message events are events that extend either of the following event classes:

- `MessageEvent`
- `BDIMessageEvent`

For more information on these event classes, refer to the *Events* chapter.

This declaration identifies events that the agent sends *externally*. It is analogous to the `#posts event` statement in all respects other than the fact that the event arises in a different agent to the one that sends it.

When an agent includes a declaration of this type, it indicates that the agent has reasoning methods or code that can send an event of *`EventType`* to other agents. The following table describes each term in this definition:

Term	Meaning
`#sends event`	This agent has methods or code that can send events to other agents. The event is always sent to a different agent and hence needs to be handled within that agent's own task execution structure.
`EventType`	Identifies the type of event to be posted.
[*`reference`*]	When present, JACK creates an agent member called *`reference`* which can be used to create events of *`EventType`* using its posting methods.

Table 3-3: Terms in a `#sends event` declaration

When a method belonging to an agent definition needs to send a message event to another agent, it does so by executing the following statement:

```
send (agentName, event )
```

`send()` is a base method provided by the Agent class. It is almost identical to the `postEvent()` method except that it takes the name of the target agent as the first argument. This is the full name that the agent is known by on the JACK runtime network. The event being posted must be constructed using one of the event's posting methods. This is described in the section on posting/sending events in the *Events* chapter.

To obtain the name of an agent, use the Agent base method `name()`. This method returns the agent's name as a `String`. Note that the name returned is the full name of the agent which takes the form *`agent@portal`* . If a portal name is not supplied, the `send()` method appends an `@` symbol, followed by the portal name of the sending agent. This can cause some ambiguity if there is more than one process in the current application, so it is advisable to always supply the full name of an agent.

#uses plan *PlanName*

This statement identifies the *plans* that an agent can execute to handle events. An agent can only execute instances of a plan if it declares that it uses this plan with a `#uses plan` declaration. If a plan is defined, but no agent uses it, that plan will never be executed.

When an agent definition includes a `#uses plan` declaration, all instances of this agent that are created have access to the given plan. This plan is said to form part of that agent's *plan set*.

Note: Ordering of `#uses plan` statements in the body of an agent (or included capabilities) is important. For normal event handling, plans are tested for relevance and applicability in the order in which they are declared. For BDI event handling, after a candidate plan set is assembled, the plan declared first in the agent will be chosen first in the absence of meta-level reasoning or plan `rank` being set.

If an agent claims to handle an event, the agent should use a plan that also handles that event. If this is not the case, a warning will be generated when you start up the application. The warning is issued because if an agent has no plan to handle a given event, it cannot strictly claim that it is capable of handling that event.

#has capability *CapabilityType reference*

The capability concept brings structure to the functional elements of agents. The user declares an agent to have selected capabilities by using the `#has capability` declaration statement. Each declaration then requires both a capability type name and a reference name that identifies the particular instance of the capability.

If, for instance, `Painting` is a `Capability` type, an agent might include the following declaration:

```
#has capability Painting painting;
```

The declaration makes the agent capable of whatever the `Painting` capability brings, that is, the agent is given access to all of the functional components enclosed by the capability. The reference name `painting` allows agent code to refer into the capability instance. An agent may have more than one instance of the one capability type.

3.4.1 Beliefsets

3.4.1.1 Conceptual Model

In JACK, beliefs are modelled as beliefset relations which take the following form:

```
relationName(key1,key2, ..., data1,data2, ...)
```

That is, each relation is identified by a name and contains any number of fields. Some of these fields are *key fields*, uniquely identifying the *kind of object* that this relation describes, and others are *value fields*, identifying the *attributes of this object* that need to be recorded.

Each object described by a beliefset relation is represented as a *tuple*. A tuple is an instance of the relation where the fields represent the key fields and value fields of a particular object. For example, one may choose to model a bank account with the following beliefset relation:

```
bankAccount(account number, name, balance, credit rating)
```

A particular bank account would then be described as a tuple, such as

```
bankAccount(10019875, "Fred Jones", 101.95,"A1");
```

3.4.1.2 Beliefset Declarations

JACK beliefset declarations within an agent take the following general form;

```
#{private|agent|global} data BeliefType belief_name (arg_list)
```

which declares that a beliefset of type *`BeliefType`* is to be contained within the agent. Each declaration is described below.

`#private data` *`BeliefType belief_name`*`(`*`arg_list`*`)`

When an agent definition includes a statement of this form, it declares a named data that is private to the agent. Agents of this class have `private` access to the beliefset relation *`belief_name`* (or to a user-defined data structure as described in the next section). Private access means that the agent has its own copy of the relation, which it can read and modify independent of all other agents, even those of the same agent class.

Each parameter in the previous definition is described in the following table:

Term	Meaning
`#private data`	A JACK Agent Language field-level construct, which specifies that agents of this class have private access to the beliefset relation.
`BeliefType`	The type of beliefset relation that the agent will use. The beliefset type is analogous to the Agent class, and extends one of the underlying JACK Agent Language types. The beliefset type defines the general properties of the relation such as: • the number and type of fields it has; • the relation's key; and • the relation's query method.
`belief_name`	Used to identify the instance of the relation that the agent is using.
`arg_list`	An optional argument. When present, it specifies arguments to be passed to the constructor each time an instance of this beliefset relation is created. For more information on beliefset constructors and how to use them, see the section entitled *BeliefSet Construction*.

Table 3-4: Terms in a `#private` JACK beliefset declaration

When a beliefset relation is private, all the relation's tuples are unique to that agent. If the agent adds or removes tuple information, this is only reflected in its own belief state. Any other agent with access to a relation of the same name will not have its own set of tuples affected. Hence, any changes made to an agent's private relations have no effect on the belief state of other agents.

Private relations are the only beliefset relations where the agent can add, modify or remove tuples: agent and global relations are read-only. It should be noted that:

- An agent can *query* a private relation's tuples using the relation's *query method*.
- An agent can *modify* a private relation's tuples using the relation's `add()` and `remove()` base methods.

Refer to the *Beliefset Relations* chapter for further details.

`#agent data` *`BeliefType belief_name(arg_list)`*

When an agent definition includes a statement of this form, it declares a named data that is shared among all agents of this type in the same process. Agents of this class have `agent` access to the beliefset relation *`belief_name`* (or to a user-defined data structure as described in the next section). Although it is not enforced, `agent` access means that the agent should have shared read-only access to the relation with other agents of the same class.

Each term from the previous definition is described in the following table:

Term	Meaning
`#agent data`	A JACK Agent Language field-level construct, which specifies that agents of this class have shared access to the beliefset relation, but only with other agents of this class.
`BeliefType`	The type of beliefset relation that the agent will use. The beliefset type is analogous to the Agent class, and extends one of the underlying JACK Agent Language types. The beliefset type defines the general properties of the relation such as: • the number and type of fields it has; • the relation's key; and • the relation's query method.
`belief_name`	Used to identify the instance of the relation that the agent is using.
`arg_list`	An optional argument. When present, it specifies arguments to be passed to the constructor each time an instance of this beliefset relation is created. For more information on beliefset constructors and how to use them, see the section entitled *BeliefSet Construction*.

Table 3-5: Terms in a `#agent` beliefset declaration

As access to agent beliefsets is intended to be read-only, the beliefset should be populated by reading data from a file as part of the beliefset constructor. Details on how this can be achieved appear in the *BeliefSet Construction* section.

The first instance of an agent class that uses an agent beliefset causes the beliefset to be constructed. Each subsequent instance of that agent class is simply allowed access to the beliefset.

An agent can *query* an agent beliefset using the relation's *query method*.

Refer to the *Beliefset Relations* chapter for further details.

`#global data` *`BeliefType belief_name`*`(`*`arg_list`*`)`

When an agent definition includes a statement of this form, it declares a named data that is shared among all the agents in the same process. This means that all the agents in the process have `global` access to the beliefset relation *`belief_name`* (or to a user defined data structure as described in the next section). Although it is not enforced, global access means that the agent should have shared read-only access to the relation with all other agents in the same process.

Note: A JACK application can consist of one or more *processes*. However, by default (and unless otherwise specified) an application consists of a single process.

Each term in the previous definition is explained in the following table:

Term	Meaning
`#global data`	A JACK Agent Language field-level construct, which specifies that the agent shares this relation with all other agents in the process.
`BeliefType`	The type of beliefset relation that the agent will use. The beliefset type is analogous to the Agent class, and extends one of the underlying JACK Agent Language types. The beliefset type defines the general properties of the relation such as: • the number and type of fields it has; • the relation's key; and • the relation's query method.
`belief_name`	Used to identify the instance of the relation that the agent is using.
`arg_list`	An optional argument. When present, it specifies arguments to be passed to the constructor each time an instance of this beliefset relation is created.

Table 3-6: Terms in a `#global` beliefset declaration

As access to a global beliefset is intended to be read-only, the beliefset should be populated by reading data from a file as part of the beliefset constructor.

The first instance of an agent class that uses a global beliefset causes the beliefset to be constructed. Each subsequent instance of an agent class that uses the same global beliefset is simply allowed access to the beliefset.

An agent can *query* a global beliefset using the relation's query methods.

Refer to the *Beliefset Relations* chapter for further details.

3.4.2 Data stored in User-defined Data Structures

Agent beliefs or normal Java objects can be stored in JACK beliefset relations or in user-defined data structures as agent data members. User-defined data structure members are declared in the agent in an analogous way to JACK beliefset relations by using the `#private data`, `#agent data` or `#global data` statements as described below. The agent's plans gain access to the user-defined data object using the `#uses data` declaration described in the chapter on plans, and like JACK beliefsets, user-defined data structures can be exported, imported or declared private to capabilities as discussed in the *Capabilities* chapter.

In addition, a plan can gain access to its enclosing agent as a Java object, (and thus to the agent's data members) by using the `#uses interface` or the `#uses agent implementing` statements described in the *Plans* chapter.

JACK declarations for user-defined data structures within an agent take the following general form:

```
#{private|agent|global} data DataType data_name(arg_list)
```

which declares that a Java object of type *`DataType`* is to be contained within the agent. Each declaration is described below:

`#private data` *`DataType data_name(arg_list)`*

When an agent definition includes a statement of this form, it declares a named data that is private to the agent. Agents of this class have private access to *`data_name`*.

Private access means that the agent has its own copy of the data object, which it can read and modify independently of all other agents, even those of the same agent class.

Each item in the previous definition is described in the following table:

Term	Meaning
`#private data`	Specifies that agents of this class have private access to the data.
`DataType`	The *user-defined* data structure.
`data_name`	The name used to identify the instance of the user-defined data structure that the agent is using.
`arg_list`	An optional argument. When present, it specifies arguments to be passed to the constructor each time an instance of this data structure is created.

Table 3-7: Terms in a `#private` user-defined data structure declaration

`#agent data` *`DataType data_name`*`(`*`arg_list`*`)`

When an agent definition includes a statement of this form, it declares a named data that is shared among all agents of this type in the same process. Agents of this class have `agent` access to the Java object *`data_name`* of type *`DataType`*. Although it is not enforced, `agent` access means that the agent should have shared, read-only access to the data object (*`data_name`*) with other agents of the same class.

Each item in the previous definition is described in the following table:

Term	Meaning
`#agent data`	Specifies that agents of this class have shared access to the data, but only with agents of the same class.
`DataType`	The *user-defined* data structure.
`data_name`	The name used to identify the instance of the user-defined data structure that the agent is using.
`arg_list`	An optional argument. When present, it specifies arguments to be passed to the constructor each time an instance of this data structure is created.

Table 3-8: Terms in a `#agent` user-defined data structure declaration

As access to the object is intended to be read-only, the data object should be initialised when it is constructed. The first instance of an agent class that uses the object will cause the object to be constructed. Each subsequent instance of that agent class is then allowed to access the object.

`#global data` *`DataType data_name`*`(`*`arg_list`*`)`

When an agent definition includes a statement of this form, it declares a named data that is shared among all agents in the process. Although it is not enforced, `global` access means that the agent has shared read-only access to the data object *`data_name`* with all other agents in the same process.

Each term in the previous definition is described in the following table:

Term	Meaning
`#global data`	Specifies that agents of this class have shared access to the data, with all other agents in the process.
`DataType`	The *user-defined* data structure.
`data_name`	The name used to identify the instance of the user-defined data structure that the agent is using.
`arg_list`	An optional argument. When present, it specifies arguments to be passed to the constructor each time an instance of this data structure is created.

Table 3-9: Terms in a `#global` user-defined data structure declaration

As access to global data is intended to be read-only, the object should be initialised when it is constructed. The first instance of an agent class that uses the global data object causes the data object to be constructed. Each subsequent instance of an agent class that uses the same global object is allowed to access the data object.

3.4.3 Task Managers

Task managers govern how an agent handles concurrent execution when they have committed to more than one task execution.

By default, JACK uses the `SimpleTaskManager`. The `SimpleTaskManager` tells the agent to persist with its currently active task until one of the following situations occurs:

- it encounters a `@wait_for` statement (and the task blocks);
- it encounters a `@sleep` statement (and the task blocks); or
- it completes (either succeeding or failing).

Regardless of the number of tasks that are outstanding, the agent will continue with a single task until one of the above conditions occurs.

As soon as one of the above conditions occurs, the task is removed from the task queue and the agent moves on to the next applicable task. If the task is completed, it is removed completely from the task queue. If the task encountered a `@wait_for` or `@sleep`, the task is returned to the end of the task queue only when the statement is completed.

If a plan step of the active task takes a long time to complete, or an infinite-loop arises, the agent will not switch to another task.

In most applications, the `SimpleTaskManager` is sufficient. However, if the agent has tasks involving intensive processing that may need to be preempted by shorter, more urgent tasks, the task manager described below may be preferable.

`#uses taskManager SimpleRRTaskManager(<steps>)`

This statement declares that the "round robin" task manager is to be used in place of the `SimpleTaskManager`. This task manager offers a more "balanced" approach to managing the tasks that are currently active within an agent. Instead of persisting with the currently active task until it either pauses or is complete, the agent rotates its efforts between all tasks that are currently active.

Each active task is kept in a *round robin queue* and is allocated a number of plan steps that it can run. A plan step is meant to represent an atomic action within a plan. In some cases, this corresponds to a single statement (such as the `@send` statement). However, many JACK Agent Language statements actually involve more than one plan step (such as the `@wait_for`, where multiple plan steps are used to test each condition). When a plan statement covers more than one plan step, this statement can be suspended after any given plan step. When the round robin task manager lets the task run again, the statement is able to resume where it left off.

If a plan calls a normal Java method, that entire method executes as a single plan step. This is because the round robin task manager does not know at which point it can suspend the task while it is executing non-reasoning method code.

The number of plan steps allocated to each task is governed by the *`steps`* argument (an `unsigned long`) supplied to the `#uses taskManager` declaration. When specifying the number of plan steps, take care to choose a value that is appropriate to the application domain. Choosing a small value minimises the chances of a single task locking out all others, maximising the agent's responsiveness to new events. However, the smaller a value, the more of an agent's processing time is devoted to context switching, reducing the agent's overall efficiency. A good value for the number of plan steps is somewhere in the hundreds (100-200), depending on the response and throughput characteristics required.

If the task terminates or pauses (by reaching a `@sleep` or `@wait_for` statement) before its number of plan steps has been reached, the agent behaves as it would with the `SimpleTaskManager`. However, if it reaches its requisite number of plan steps before this happens, the active task is moved to the end of the task queue, and the task at the head of the

queue is activated. Therefore, all active tasks are given an opportunity to run, and the danger of one CPU-intensive, long-running task hampering the agent's overall performance is reduced.

3.5 Agent Members and Methods

Once the required set of #-declarations has been added to an agent's definition, the `event`, `plan` and `beliefset` components that the agent requires will need to be identified. Each of these components contains its own data members and methods. The remaining step in completing the agent definition is to specify the agent's data members and methods.

An agent's data members and methods are defined using normal Java. After all, agents are fundamentally implemented as Java classes, so the full Java functionality is available. The user may want to define data structures for the agent to use that are different from the beliefset construct provided, and you may want to include methods to post events when certain situations arise. You will certainly want to define constructors and destructors so that individual agents can be created and destroyed. Note that at least one constructor for the agent must be defined.

When defining an agent's methods, the user may wish to use some of the base methods that the Agent class provides. Since all agents extend the `Agent` class, these methods are always available. Typically, they implement useful low level functionality such as:

- constructing an agent instance;
- terminating an agent;
- posting an event (only those identified using a `#posts event` or `#sends event` declaration);
- sending messages to other agents;
- replying to messages from other agents;
- specifying which *timer* the agent will use; and
- determining the agent's name on the JACK network.

Agents also have a data member that allows you to specify the `Timer` (clock) that the agent uses to measure the passage of time.

Furthermore, if any of the agent's plans require it to implement a particular interface, all the methods specified by this interface must be included among the agent's methods and be fully implemented.

Note: Personalised `Agent` sub-classes can be defined by extending from other `Agent` classes that have been defined. It is not necessary to extend directly from the base `Agent` class.

Agent Construction

To construct an agent, follow the convention for constructors used in Java. An example of agent construction is shown below:

```
agent ExampleAgent extends Agent
{
   // #-statements
   // data members

   ExampleAgent (String name)
   {
      super(name);
      ...
   }
}
```

Agent Termination

To terminate an agent, use the `Agent` base method `finish()`. When this method is executed, all event processing within the agent terminates immediately and the agent is removed from the JACK runtime network. Actual removal of the agent and the freeing of its allocated memory is left to the Java garbage collector.

void postEvent(Event *e*)

The `postEvent()` method is used by the agent to post a new event. It can be used to post all types of events.

The prototype for this method is shown below:

```
public void postEvent (Event event_name);
```

where *event_name* is the name of the event to be posted. Note that to use this method:

- the event being posted must have already been constructed; and
- this event must have been included among the events that this agent can handle (by means of a `#handles event` declaration)

An agent executes a separate task to handle posted events. Hence, the `postEvent()` method does not return a result, even if the agent cannot handle the event (e.g. if none of its plans are relevant or applicable).

The event is handled *asynchronously* by the agent. Note that this is also true if the event posted is a goal event. Normally goal events are handled synchronously in separate subtasks but when posted using this method, they are treated and handled asynchronously as normal events.

Message events should not be posted in this way. If they are, they will appear as internal events and the `from` member will be set to null.

boolean postEventAndWait(Event *e*)

This base method is similar to the `postEvent()` method, except that instead of posting the event asynchronously, it is posted *synchronously.* The agent still executes the event as a separate task, but the calling method must wait until this task has been completed before continuing. The prototype for this method is shown below:

```
public boolean postEventAndWait(Event event_name);
```

The method returns a boolean result depending on whether the *task* that the agent performs to handle the event succeeds (true) or fails (false).

Note: Unlike most other agent methods, this method must not be called from any of an agents tasks. It can only be used from methods used by normal Java programs that are integrated with the JACK application.

void send(String *s*, MessageEvent *e*)

This method is used to send messages (`MessageEvents` or `BDIMessageEvents` to other agents. The prototype for this method is given below:

```
public void send (String name, MessageEvent message)
```

It takes two arguments:

- the *name* of the destination agent, and
- a `MessageEvent` to send to this agent.

The agent name can either be in short (*agent*) or full (*agent@portal*) form. The short name is simply the name that was specified in the constructor of the agent. If a short agent name is passed, the message event will only reach the agent if it is running within the same process as the agent that sends the message. If a fully-qualified name is passed, on the other hand, the message will reach the destination agent if it is running anywhere on the same DCI network.

void reply(MessageEvent *q*, MessageEvent *r*)

This method is used to send messages (`MessageEvents` or `BDIMessageEvents`) back to an agent from which a previous message originated.

If the agent has received a message and performed a task in response to this message, one of the steps in the plan that responds to this message may be to send another message back to the originating agent in the form of a reply. This may be to confirm that the task has been completed.

The prototype for this method is given below:

```
public void reply (MessageEvent query, MessageEvent response)
```

Unlike the `send()` method, the `reply()` method does not require specification of the destination agent as an explicit input argument. This is because the agent already knows which agent sent the original message. The sender's address is specified in the original message event's `from` member.

String name()

This method is used to retrieve the agent's full name, which includes its process portal name. It returns the name as a `String`.

The name returned consists of two components:

- the agent's name as it is known locally within the process; and
- the portal name assigned to the current process.

For example, suppose an agent called `kermit` is running in a process which has been assigned a portal name `sesameStreet`. The name returned by this method would be `kermit@sesameStreet`.

Timer timer

This data member specifies which timer (clock the agent will use to measure the passage of time. JACK includes a number of different `Timer` classes to give programmers more control over how agents respond to the passage of time in a program.

The definition of this member is given below:

```
Timer timer;
```

The `Timer` class is JACK specific. Timers can be categorised as: the real-time clock (measuring the passage of time as would a normal system clock); dilated clocks (enabling the agent to manipulate time with effect, such as fast forward, slow motion and pause); and simulation clocks (enabling manual ticking and even greater control than that offered by dilated clocks).

4 Capabilities

4.1 Introduction

The *capability* concept is a means of structuring reasoning elements of agents into clusters that implement selected reasoning capabilities. This technique simplifies agent system design, allows code reuse, and encapsulation of agent functionality.

Capabilities represent functional aspects of an agent that can be plugged in as required. This *capability as component* approach allows an agent system architect to build up a library of capabilities over time. These components can then be used to add selected functionality to an agent.

Additionally, capabilities can be structured so that a number of sub-capabilities can be combined to provide complex functionality in a parent capability. This capability can in turn be added to an agent to give it the desired functionality.

Capabilities are built in a similar fashion to simple agents – constructing them is merely a matter of declaring the JACK Agent Language elements required. Events, beliefsets, plans, Java code and other capabilities can all be combined to make a capability.

In this section, the capability concept is described, and capability declarations are explained. Each of the specific `#`-declarations, members and methods pertaining to capabilities is listed in following sub-sections.

4.2 Capability Definition

A capability is defined similarly to other type-level concepts as a code block by the keyword `capability`. A capability definition takes the form shown below:

```
capability CapabilityType extends Capability
        [implements Interface]
{
   // JACK Agent Language Statements specifying
   // the functionality associated with this
   // capability.
}
```

Each component of this definition is explained in the following table:

Syntax Term	Description
`capability`	A JACK Agent Language keyword used to introduce a capability definition.
`CapabilityType`	The name of the derived `Capability` class.
`extends Capability`	This part of the statement plays the same role as in Java – it indicates that the capability being defined inherits from a JACK Agent Language base class called `Capability`. The `Capability` base class implements all the underlying methods that provide a capability's core functionality.
`[implements` *`Interface`*`]`	This part of a capability definition is optional. When present, it states that a capability *implements a given Java interface*. Java interfaces are classes that consist of method prototypes without code. When a capability implements an interface, it provides code to implement each of these methods.

Table 4-1: Components of a Capability definition

In the body of each capability, events, beliefsets, plans and other capabilities that pertain to the functionality provided by the given capability are declared. Java entities (i.e. methods and members) may be declared in capabilities as in a class. Each capability is instantiated with each agent that contains the corresponding `#has capability` declaration.

The JACK `#`-declaration statements used for agents are all available for use in capabilities, except those that refer to task managers. In addition, there are new declarations that allow specifying events and beliefsets as shared with an enclosing capability.

4.3 Capabilities and Interfaces

The optional `implements Interface` component in a capability definition is important when it comes to writing portable JACK programs that allow for code re-use. Interfaces provide a mechanism for agents and capabilities to share plans.

When an agent executes a plan, this plan will often call ordinary Java methods. It is important to remember that when this occurs, these methods must be available to the plan through having been declared in the agent or one of the capabilities that it uses. Therefore, any capability that uses this plan must include the defined methods, or the plan will not be able to run properly.

The JACK Agent Language allows you to package up these dependencies into a Java `Interface`. Any capability that wishes to include this plan must declare that it *implements this interface*. If a capability implements the interface, it provides all the methods necessary to run the plan.

4.4 Capability Declarations

A capability should fully describe the functionality it implements via JACK Agent Language declarations. In general, a capability definition will require declarations for the following:

- *BeliefSets and Views* which the capability can use and refer to;
- *Events* (both internal and external) that the capability handles;
- *Plans* that the capability can execute;
- *Events* that the capability can post internally (to be handled by other plans); and
- *Events* that the capability sends externally to other agents.

These declarations are provided by statements that occur at the field or member level of a capability definition. By convention, these statements should appear before the definitions of any regular Java data members and methods that the capability may contain.

An example capability template showing **some** of the declarations that can appear in a capability appears below:

```
capability CapabilityType extends Capability
        [implements InterfaceName]
{

   // Knowledge bases used by the capability are declared here.

   #private data BeliefType belief_name (arg_list);
   #exports data BeliefType belief_name (arg_list);
   #imports data BeliefType belief_name ();

   // Plans used by the capability are declared here.
   // Order is important.

   #uses plan PlanType;

   // Events posted, sent and handled are declared here.

   #handles event EventType;
   #handles external [event] EventType;
   #posts event EventType reference;
   #posts external [event] EventType reference;
   #sends event EventType reference;

   // Sub-capabilities are declared here.

   #has capability CapabilityType reference;

   // other Data Member and Method definitions
}
```

Each JACK Agent Language capability declaration is described in more detail in the following sub-sections.

#handles event *EventType*;

As for agents, the `#handles event` statement declares that there are plans *within this capability* that handle events of the given type. The declaration also implies that these events are *local* to the capability and its sub-capabilities, which means that the connection between posting and handling does not cross the boundary between this capability and its enclosing capability. In other words, events of type *`EventType`* that are posted externally to this capability are not handled within this capability, and the events of type *`EventType`* that are posted within the capability are not visible for the enclosing capability. However, the declaration does not make the event type 'invisible' for inner capabilities; rather, this is decided in their definitions.

#handles external [event] *EventType*;

The `#handles external` statement declares that there are plans within this capability that handle events of the given type, and that this type is shared with the enclosing capability. This means that event posting and handling *does* cross the capability boundary upwards. That is, the `external` keyword declares that the event in question may be handled by the parent capability or any of its sub-capabilities that declare that they handle this external event, and these capabilities' plans, if any, contribute to the plan set assembled to handle this event. Note that the keyword `event` is optional.

#posts event *EventType reference*;

The `#posts event` statement declares that there are plans or code within this capability that post events of the given type. The declaration also indicates that these events are local to the capability, which means that the connection between posting and handling does not cross the boundary between this capability and its enclosing capability. A *`reference`* name is needed only if the event is to be posted from Java code within the capability.

#posts external [event] *EventType reference*;

The `#posts external` statement declares that there are plans or code within this capability that post events of the given type, and that this event type is shared with the enclosing capability. All plans from all capabilities that handle this shared event then compete to handle it. Note that the keyword `event` is optional.

#sends event *EventType reference*;

The `#sends event` statement declares that there are plans or code within this capability that send events of the given type. It is important to note that events are received at agent level (the addressable entity), which means events cannot be explicitly directed into a capability of another agent.

#private data *DataType data_name(arg_list)*;

The `#private data` statement declares that a Java object or a JACK beliefset of type *`DataType`* is *local* to the capability, and is accessible only from within the capability and its sub-capabilities. Note that the statement results in the instantiation of the beliefset or object using the specified constructor.

#agent data *DataType data_name(arg_list)*;

When a capability definition includes a statement of this form, it declares a named data, *`data_name`* of type *`DataType`*. The named data is shared among all agents in the same process that have the same type as the agent in which the capability instance is created. If the same capability type is used by two different agent types, there will be a different instance of the named data created for each of the agent types. Although it is not enforced, `agent` access should be shared, read-only access. As access to `agent` data is intended to be read-only, the beliefset or object should be initialised when it is constructed. The first instance of an agent class that uses the beliefset or data object, causes it to be constructed.

#global data *DataType data_name(arg_list)*;

When a capability definition includes a statement of this form, it declares a named data, *`data_name`*. The named data is shared among all the agents in the same process. Although it is not enforced, `agent` access should be shared, read-only access. As access to `global` data is intended to be read-only, the beliefset or object should be initialised when it is constructed. The first instance of an agent class that uses the beliefset or data object, causes it to be constructed.

#exports data *DataType data_name(arg_list)*;

The `#exports data` statement declares that a Java object or a JACK beliefset of type *`DataType`* is exported from the capability so that it is accessible from its parent capability. The export statement can also be used to make a data object or beliefset available at agent level, and accessible from other capabilities within the agent level. Note that the statement results in the instantiation of the beliefset or object using the specified constructor.

`#imports data` *`DataType data_name();`*

The `#imports data` statement declares that a Java data object or JACK beliefset of type *`DataType`* is shared with this capability and its enclosing agent or capability. Note that **no** instantiation occurs as a result of this statement.

`#uses plan` *`PlanType;`*

The `#uses plan` statement declares that this capability uses plans of the given type. The usage is unique for the capability in the sense that even if the same plan type is used in another capability, the plan instances extending from each use are wholly distinct. This means that even if they are used in a way so as to share a handled event (e.g. used by two 'sibling' capabilities where the handled event is external in both), the usages within the different capabilities generate their own plan instances independently.

`#has capability` *`CapabilityType reference;`*

The `#has capability` statement declares use of an inner capability. Note that the reference name is required. The inner capability is then accessible through the reference name.

4.5 Capability Members and Methods

The remaining step in completing the capability definition is to specify the capability's data members and methods. When defining a capability's methods, it is possible to use some of the base methods that the `Capability` class provides. Since all capabilities extend the `Capability` class, these methods are always available. If any of the capability's plans require it to implement a particular interface, all the methods specified by this interface must be included among the capability's methods and be fully implemented.

Note: `Capability` sub-classes can be defined by extending from other `Capability` classes, not just from the base `Capability` class.

Once the desired set of `#`-declarations has been added to a capability's definition, the `event`, `plan`, `beliefset` and `capability` components that the capability requires need to be defined. Each of these components contains their own data members and methods.

Capability Construction

While capabilities are not explicitly constructed by the user, they are brought into being when the enclosing agent is constructed. They can be initialised by overriding the `autorun()` method, described below.

Capabilities do not have a name in the same sense that agents do, but can be referred to through the chain of reference names used in the `#has capability` statements. This reference name can be retrieved by calling the `toString()` method on the `Capability` instance in question.

The reference name starts with the agent name, and thereafter the names used for traversing down the capability structure to the instance in question, where each name is separated with ':'. Thus, a capability name like `bob@builders:tiling:bathroom_tiling` says that this is the capability instance `bathroom_tiling` of the capability instance `tiling` of the agent instance `bob` at portal instance `builders`.

JACK Agent Language entities (i.e. plans, events and beliefsets) that belong to capabilities are each instantiated in slightly different ways. These differences mean that it is not always possible to determine the name of the capability to which the entities belong. For example, when a plan is instantiated, it is associated with an enclosing *`NameSpace`* object, which is either a `Capability` object or an `Agent`. This object is accessible through the public `Plan` member `__ns`. `Event` and `beliefset` instances are not explicitly associated with the enclosing capability instance. This means that a plan cannot query an event or beliefset instance about which capability they belong to.

public void postEvent(Event *e*)

The `postEvent()` method is used to post events within capability code. This method is actually just a convenient method that refers to `getAgent().postEvent()`.

See the discussion of the `postEvent()` method in the **Agents** section for more details.

public Agent getAgent()

The method is called on a capability instance to return the containing agent, which is generically typed.

protected void autorun()

The `autorun()` method is invoked before the agent is fully constructed. The invocation is at the end of the construction of the capability after all its sub-capabilities have been fully constructed. The `autorun` method can be overridden in order to provide some initialisation when the capability is constructed.

5 Events

5.1 What are Events?

Events motivate an agent to take action. There are a number of event types in JACK, each with different uses. These different event types help model:

Internal stimuli – events that an agent sends to itself, usually as a result of executing reasoning methods in plans that an agent has. These internal events are integral to the ongoing execution of an agent and the reasoning that it undertakes.

External stimuli – such as messages from other agents, or percepts that an agent receives from its environment.

Motivations – that the agent may have, such as goals that the agent is committed to achieving.

Events are the origin of all activity within an agent-oriented system. In the absence of events an agent sits idle. Whenever an event occurs, an agent initiates a task to handle it. This task can be thought of as a thread of activity within the agent. The task causes the agent to choose between the plans it has available, executing a selected plan or plan set (depending on the event processing model chosen) until it succeeds or fails.

If plan execution succeeds, the event that initiated it is said to have succeeded. If plan execution fails, there are two options. Under normal event handling, the event is said to have failed after the first instance of plan failure.

Under BDI event handling, a number of plans can be selected for execution and these are attempted in turn, in order to try to achieve successful plan execution. If the set of chosen plans is exhausted, the event is said to have failed.

Both cases are regarded as successful event handling.

There are a number of event classes in the JACK Agent Language, each representing different types of motivation to act. Additionally, there are event types available that facilitate analysis and debugging of inter-agent communication. In general, these events fall into two broad categories:

- Normal events, and
- BDI events.

5.2 Normal Events

Normal events in JACK are analogous to events in conventional event-driven programming. That is, they represent transitory occurrences that initiate a single, immediate response from an agent.

For example, suppose that an agent is being programmed to play soccer. If the agent receives a message from the simulation environment telling it where the ball is every second, this would be modelled as a normal event. It represents an occurrence that must be acted upon immediately or not at all. If the agent decides to do something (execute a plan instance) and that plan instance fails, it would not make sense to reconsider an alternative because the information that the agent is acting on is now out of date.

5.2.1 How an Agent handles Normal Events

When a normal event is received by an agent, the agent initiates a *task* to handle it. This task involves the agent selecting and executing the first plan that is both *relevant* and *applicable* to this event. These tests for relevance and applicability are performed on plans in the order that the `#uses plan` statements occur in the agent or capability code.

A plan is *relevant* to a given event if it has a `#handles event` declaration that matches the event, and its `relevant()` method succeeds when executed.

Note: If no plans are found that are relevant to a particular event, the event processing task fails and the system returns to a state where it is ready to process any other incoming events.

The agent then checks if the relevant plan is *applicable* in the current circumstance. It does this by executing the `context()` method of the plan. If the `context()` method succeeds, it is chosen as the plan that the agent will execute in response to the incoming event. If the `context()` method fails, the agent examines the next relevant plan.

Simple `context()` methods return either true or false, meaning that a single instance of the plan is either applicable or not applicable. However, more complex `context()` methods may include *cursor expressions* and involve binding of logical variables. These expressions attempt to unify a given expression with tuples in one or more beliefset relations. For every tuple that matches, the logical variables are bound. Each such match causes a separate plan instance to be generated.

In the case of normal event processing the first such plan instance is chosen as the plan that the agent will execute in response to the incoming event.

Each normal event is implemented as a JACK Agent Language class. Each event has a number of base members and methods to provide access to their functionality and data. Each of these base classes can be extended and unique members and methods added to events.

5.2.2 Normal Events in the JACK Agent Language

The JACK Agent Language normal event classes are listed below and are described in the following sub-sections:

Event Type	Description
`Event`	Base class for all events in JACK. Posted and processed *inside* an agent to invoke plans, that may in turn post further events.
`MessageEvent`	`MessageEvents` are received by agents from external sources – usually other agents. These are normally sent from one agent to another to facilitate inter-agent communication.

Table 5-1: JACK normal event classes

5.2.2.1 The Event Class

The base class for all events in JACK is the `Event` class. This class implements all the core event functionality required by the JACK runtime environment. A normal `Event` is only ever generated by processing that occurs within an agent.

Events are always *posted* explicitly by statements within agent code. To say an agent posts an event means that an event is constructed and added to the agents own event queue to await processing. If an agent is idle, no code is being executed and an agent cannot post any events. In this case, an agent can only take action if an external (i.e. `MessageEvent`) arrives.

Events are posted:

- when an agent executes a `@post` statement from within a plan;
- when an agent executes either the `postEvent()` or `postEventAndWait()` base method from agent code; and
- when an agent modifies a beliefset relation for which a beliefset modification callback has been defined.

In each case, an instance of the event is posted using one of the event's *posting methods*. The posting method acts as the event's constructor, creating an instance of the event and making it available to the agent's task manager.

5.2.2.2 The MessageEvent Class

MessageEvents represent events that are used to communicate with other agents. Whenever an agent needs to send a question, command or message to another agent, this information is packaged and sent as a MessageEvent. Instead of representing an internal occurrence that initiates activity within the agent, MessageEvents represent an *external occurrence* that initiates activity within the agent.

MessageEvents can also be used as a mechanism to allow an agent to experience or perceive the environment in which it is executing. That is, anything affecting the agent that does not originate from the agent's own internal processing should be sent to the agent as a MessageEvent.

MessageEvents often represent transient requests that the agent may want to act upon. For example, if an agent is programmed to play a game such as soccer in a simulation environment, the environment may send the agent MessageEvents periodically to tell it where the ball is. Clearly, this is a message that is relevant and applicable only at the moment that it arrives. It serves to inform the agent about an external situation that may concern it. If the agent is a mid-fielder and it is sufficiently close, it may invoke a plan to attempt to intercept the ball. If it is not close enough it may try to cover a designated opponent. Alternatively, if the agent is a goal-keeper, it may attempt to move itself between the ball and the goal.

In each case, the agent is taking action on information that is *applicable only at the moment at which it arrives*. The agent must decide what to do with the information immediately, and once a course of action is chosen the information is discarded.

MessageEvents are sent to other agents:

- when an agent executes a @send statement from within a plan; or
- when an agent executes its send() base method.

Note: Agents should only execute the send() method in code outside a reasoning method. The @send statement is meant to be used to send message events to other agents from within a reasoning method.

In order for agents running in different processes to send and receive message events, the JACK runtime environment must provide a communication infrastructure with message routing capabilities. This is done using the JACK DCI network. The JACK DCI network enables the establishment of portals between processes, through which agent messages can be directed as needed. Refer to the *Inter-agent Communications* chapter for further details.

Note: All data fields within a MessageEvent must be serializable.

5.2.2.3 The `TracedMessageEvent` Class

The `TracedMessageEvent` class contains information which enables inter-agent communication to be effectively displayed via an Agent Interaction Diagram (refer to the *Tracing and Logging Manual* for more details on the Agent Interaction Diagram). This information is now available within the message event classes; consequently, `TracedMessageEvent` was deprecated in JACK version 3.5.

5.3 BDI Events

Belief-Desire-Intention (BDI) events represent a different class of event to the `Event` and `MessageEvent` described in the previous section.

One of the important aspects of the BDI reasoning model at a conceptual level is that it models *goal-directed behaviour* in agents, rather than plan-directed behaviour. That is, an agent *commits* to the desired outcome, not the method chosen to achieve it. While normal events represent transient information that the agent reacts to (such as the changing of a dial reading, or the location of a ball), BDI events allow an agent to pursue long term goals.

When using the BDI reasoning model, an agent does not simply react to incoming information, but sets itself a goal which it then tries to achieve. Rather than distracting an agent from its goal, incoming events are added to an agents knowledge base and can subtly influence its behaviour.

5.3.1 How an Agent handles BDI Events

The key difference between normal events and BDI events is how an agent selects plans for execution. With normal events, the agent selects the first applicable plan instance for a given event and executes that plan instance only.

The handling of BDI events is more complex and powerful. An agent can assemble a plan set for a given event, apply sophisticated heuristics for plan choice and act intelligently on plan failure. At least one of the following characteristics applies to each type of BDI event under the BDI model:

- Meta-level reasoning – a technique for writing plans about how to select plans. This can help in refining plan selection to allow for selection of the most appropriate plan in a given situation.
- Reconsidering alternative plans on plan failure – if a course of action (plan) fails, this technique allows an agent system to consider other courses of action to achieve the goal that has been set.
- Recalculating the applicable plan set – it is possible when acting on plan failure to assemble a new plan set which excludes any failed plans.

Additionally, it is possible to further control BDI behaviour by setting *behaviour attributes*. These attributes are described later in this chapter.

5.3.1.1 Meta-level Reasoning

When handling a BDI event, meta-level reasoning allows precise control over how an agent chooses a plan for execution from the set of applicable plans. Whenever there is more than one applicable plan instance for a given BDI event, a `PlanChoice` event is posted within the agent. By choosing to handle this event, an agent can implement meta-level reasoning.

If the meta-level reasoning plan fails and does not select a plan for execution, the default plan selection method is invoked. This is discussed in more detail in the chapter on meta-level reasoning.

5.3.1.2 Reconsidering Alternative Plans on Plan Failure

When handling BDI events, an agent will not always assume that failure of a chosen plan instance means the goal cannot be achieved. Instead, it may reconsider other applicable plans and try one of them instead.

It is this property that allows agents to avoid many of the pitfalls of more primitive reasoning models. Rather than having 'one shot' at achieving a goal, an agent can try a number of approaches to solving the problem by attempting any number of applicable plans.

5.3.1.3 Recalculating the Applicable Plan Set

This property controls how an agent takes into account the passage of time and changing circumstances when choosing alternative plan instances to handle a BDI event after the previous selected plan has failed. An agent may select an alternative plan in one of the following ways:

- keep track of the plan instances that were initially applicable and select another member of this set; or
- recompute which plan instances are applicable and select one from the new set, excluding plan instances that have already failed.

If circumstances and the agent's beliefs haven't changed, both approaches are equivalent. However, if the applicable plan instances depend on the agent's current set of beliefs and these beliefs have changed, the recomputed set of applicable plans may be different from the original set. This is especially relevant if the agent is using the `SimpleRRTaskManager`, or if the failed plan itself changed one or more beliefset tuples that affect the context calculations of other relevant plans.

5.3.2 The BDI Events in the JACK Agent Language

The JACK Agent Language event classes are listed below and are described in the following sub-sections:

Event Type	Description
`BDIFactEvent`	Base class for all BDI Events in JACK. Only posted internally by an agent for its own use, but allows meta-level reasoning to occur for plan selection.
`BDIMessageEvent`	`BDIMessageEvents` are received by agents from external sources – usually other agents. These are normally sent from one agent to another to facilitate inter-agent communication. Meta-level reasoning can be performed for plan selection on receipt of this event type.
`BDIGoalEvent`	A `BDIGoalEvent` represents an objective that an agent wishes to achieve. Meta-level reasoning, alternative plan selection and plan set recalculation are all available for this event type.
`InferenceGoalEvent`	An `InferenceGoalEvent` uses `RuleBehavior`, which extends `BDIBehavior` by processing all applicable plans (rather than only one of them).
`PlanChoice`	Posted when there is more than one plan to choose from and optionally handled by the agent to implement meta-level reasoning.

Table 5-2: BDI event classes

5.3.2.1 The `BDIFactEvent` Class

`BDIFactEvents` represent internal events that arise due to the agent's own internal processing.

The main difference between a `BDIFactEvent` and a normal `Event` is that when there is more than one applicable plan instance for a given `BDIFactEvent`, the agent is able to perform meta-level reasoning to determine which instance it should execute. A `PlanChoice` event is posted to trigger this meta-level reasoning, and if the agent has a plan that can handle it, it will execute this plan to determine which applicable plan instance it should execute.

Once the agent has chosen a plan instance to handle a `BDIFactEvent`, it will commit to this plan.

If the plan instance succeeds, the `BDIFactEvent` succeeds and returns to the plan that generated it.

If the plan instance fails, the default behaviour is that the `BDIFactEvent` fails and returns control to the plan that generated the `BDIFactEvent`.

`BDIFactEvents` are usually posted under the following circumstances:

- when the agent executes a `@post` statement from within a plan,
- when the agent executes either the `postEvent()` or `postEventAndWait()` base method from agent code outside plan execution, or
- when the agent modifies a beliefset relation for which a *beliefset modification callback* has been defined.

For more information on beliefset modification callbacks, refer to the *Beliefset Relations* chapter.

In each case, an instance of the event is posted using one of the event's *posting methods*. The posting method acts as the event's constructor, creating an instance of the event and making it available to the agent's task manager.

The BDI event processing properties supported by `BDIFactEvents` by default are summarised in the following table:

BDI Event Property	Supported by Default
Allows meta-level reasoning.	Yes.
Allows reconsideration of alternative plans.	No.
Recalculates the applicable plan set when reconsidering alternatives.	No.

Table 5-3: Default BDI event processing properties, supported by `BDIFactEvents`

5.3.2.2 The `BDIMessageEvent` Class

`BDIMessageEvents` are the same as normal `MessageEvents` but with BDI extensions. They allow the agent more scope when reasoning about how it should respond to a message received from another agent or process.

Like `BDIFactEvents`, `BDIMessageEvents` allow the agent to perform meta-level reasoning when more than one plan instance is applicable. This lets the agent choose which of its available set of responses is most appropriate according to given criteria.

For example, suppose an agent programmed to play as a soccer goal keeper in a soccer simulation has three plans that are applicable when defending a penalty – jump to the left, jump to the right and stay in the centre. When a penalty is called (arriving as a message event) all three of these plans would be applicable.

If the message event is of class `MessageEvent`, the agent can only select the first plan declared in the agent or capability. However, if the message event is a `BDIMessageEvent`, the agent has the option of performing meta-level reasoning to determine which plan is best. This meta-level reasoning may take into account various statistics from the game and past successes of each approach, then make the selection accordingly.

Once the agent has chosen a plan instance to handle a `BDIMessageEvent`, it will commit to this plan. If the plan instance succeeds, the `BDIMessageEvent` succeeds and returns to the plan that generated it.

If the plan instance fails, the default behaviour is that the `BDIMessageEvent` fails and returns control to the plan that generated the `BDIMessageEvent`.

This default behaviour can be overridden by adding either of the following to a `BDIMessageEvent` subclass:

```
    #set behavior Recover repost;
or
    #use behavior BDIGoalBehavior();
```

`BDIMessageEvents` can be sent by agents or by other Java processes running within a JACK application. Like `MessageEvents`, they are sent under the following circumstances:

- when an agent executes a `@send` statement from within a plan; or
- when an agent executes its `send()` base method.

Note: Agents should only execute the `send()` method from code outside a reasoning method. This is because the `@send` statement is meant to be used to post message events to other agents from within a reasoning method.

In order for agents running in different processes to send and receive message events, the JACK runtime environment must provide a communication infrastructure with message routing capabilities. This is done using the JACK DCI network. The JACK DCI network enables the establishment of portals between processes, through which agent messages can be directed as needed. Refer to the *Inter-agent Communications* chapter for further details.

The BDI event processing properties supported by `BDIMessageEvents` by default are summarised in the following table:

BDI Event Property	Supported by Default
Allows meta-level reasoning.	Yes.
Allows reconsideration of alternative plans.	No.
Recalculates the applicable plan set when reconsidering alternatives.	No.

Table 5-4: Default BDI event processing properties, supported by `BDIMessageEvents`

Like `MessageEvents`, `BDIMessageEvents` have a `message` member and can be displayed on an Agent Interaction Diagram. Refer to the *Tracing and Logging Manual* for more details on the Agent Interaction Diagram.

Note: All data fields within a `BDIMessageEvent` must be serializable.

5.3.2.3 The `BDITracedMessageEvent` Class

The `BDITracedMessageEvent` class contains information which enables inter-agent communication to be effectively displayed via an Agent Interaction Diagram (refer to the *Tracing and Logging Manual* for details on the Agent Interaction Diagram. This information is now available within the message event classes and consequently the `BDITracedMessageEvent` was deprecated in JACK version 3.5.

5.3.2.4 The `BDIGoalEvent` Class

`BDIGoalEvents` are unlike the other BDI events described so far. The `BDIFactEvent`, and `BDIMessageEvent` extended existing normal events with some of the BDI reasoning capabilities. The `BDIGoalEvent`, however, is exclusive to the BDI model of agent reasoning. Unlike the other BDI events, it offers all the BDI features by default.

The `BDIGoalEvent` represents a *goal* or *objective* that an agent wishes to achieve. Therefore, instead of representing reactive behaviour, `BDIGoalEvents` represent pro-active behaviour in an agent. When an agent posts and handles a `BDIGoalEvent`, it is adopting a goal. It will then use all the reasoning powers at its disposal – the ability to discard and reconsider plans, the ability to reassess which plans are applicable and the ability to choose between the plans that are applicable, in order to satisfy that goal.

An example of this might be a pilot agent adopting a goal to land an aeroplane. This would be posted internally as a `BDIGoalEvent`. This then becomes an objective that the agent commits to achieving, and the agent should try every applicable plan until one succeeds, and only give up and fail when no more applicable plan instances remain to be tried.

BDIGoalEvents usually arise within an agent as a result of executing specific JACK Agent Language statements in a reasoning method. These statements represent the agent posting the goal for itself, or choosing to undertake a task to satisfy the goal. When an agent adopts a goal, it can do so with the following objectives in mind:

- *achieve* the goal,
- *insist* that the goal is achieved, by double-checking that the goal has been met when the agent finished the task,
- *test* whether the goal can be achieved, or
- *determine* a situation in which the goal can be achieved.

Each of these objectives is specified by a separate reasoning method statement (@achieve, @insist, @test and @determine). These reasoning methods are described fully in the *Plans* section.

BDIGoalEvents can also be posted and sub-tasked – they do not arise solely from the reasoning method statements @achieve, @insist, @test and @determine. However, if they do not arise from one of these statements, the mode (discussed in the section on *Event members*) will be set to null.

The BDI event processing properties supported by BDIGoalEvents by default are summarised in the following table:

BDI Event Property	Supported by Default
Allows meta-level reasoning.	Yes.
Allows re-consideration of alternative plans.	Yes.
Re-calculates the applicable plan set when reconsidering alternatives.	Yes.

Table 5-5: Default BDI event processing properties, supported by BDIGoalEvents

5.3.2.5 The InferenceGoalEvent Class

An InferenceGoalEvent uses RuleBehavior, which extends BDIBehavior by processing all applicable plans (rather than only one of them). The behaviour is the same as BDIBehavior, except for the following:

- When a plan succeeds, instead of completing the event, the plan is added to the 'tried set' (called 'failed set' for BDIBehavior), and the next applicable plan is processed.
- When a plan fails, the event ignores this and continues with the next plan. This aspect is tunable by means of a behaviour attribute as follows:

– to fail the event with the first failing plan, use

```
#set behavior RuleFailure fail;
```

– to fail the event if any of the plans fail, but only after all the applicable plans have been processed, use

```
#set behavior RuleFailure delay;
```

– to ignore plan failure (i.e. the event always succeeds) use

```
#set behavior RuleFailure ignore;
```

The default behaviour is to ignore plan failure (i.e. the event always succeeds). The `RuleFailure` attribute only applies to events that use `RuleBehavior` (e.g. `InferenceGoalEvents`).

The behaviour of an `InferenceGoalEvent` is the same as `BDIBehavior` in terms of forming the applicable set and in terms of meta-level plan choice (although in this case, all plans will be processed). An `InferenceGoalEvent` can be fine tuned in the same way as other BDI events, using the behaviour attributes. Behaviour attributes are discussed in detail later in this chapter. The default `BDIBehavior` attribute settings are as follows:

```
Recover = repost ; //this attribute cannot be changed
ApplicableSet = once ;
ApplicableChoice = first ;
ApplicableExclusion = failed ;
PlanBindings = all ;
OnError = propagate ;
PostPlanChoice = never ;
```

The default `RuleBehavior` attribute setting is:

```
RuleFailure = ignore ;
```

Note that an `InferenceGoalEvent` can only be posted. To achieve the same behaviour when sending an event between agents, the `#use behavior RuleBehavior;` declaration is added to an event which has extended a normal message event. For example:

```
public event ExampleEvent extends MessageEvent
{
    #use behavior RuleBehavior();

    #posted as postingMethod()
    {
    }
}
```

Note: The applicable plan set only includes the highest ranking plans – not all the plans that are relevant and within context. As `InferenceGoalEvents` (and events that use `RuleBehavior`) only calculate the applicable set once, they will only activate the highest ranking plans.

5.3.2.6 The `PlanChoice` Event Class

`PlanChoice` events are a separate class of event altogether, but they are included with BDI events because they represent the mechanism whereby an agent performs meta-level reasoning about the handling of BDI events.

Unlike the other event classes in the JACK Agent Language, a user would not explicitly post a `PlanChoice` event by executing a method or a reasoning method statement in a plan. Instead, `PlanChoice` events are posted internally by an agent whenever:

- an instance of a BDI event arises within the agent, and
- there is more than one applicable plan instance available to the agent to handle this event.

When this happens, the agent is given an opportunity to choose which of the applicable plan instances it will execute. The task to handle this `PlanChoice` event proceeds like any other task execution in JACK. The agent looks for plan instances that are relevant to this `PlanChoice` event and applicable under the current circumstances, selects one and then executes it. The plan should examine the set of applicable plans and select one for the agent to try.

Because an agent handles a `PlanChoice` event in the same way that it handles a normal event, the processing of `PlanChoice` events is referred to as *meta-level reasoning*. It represents reasoning (task execution) that an agent undertakes not to do work, but to decide what is the best approach to doing work.

Plans that are used for meta-level reasoning, as opposed to handling normal or BDI events, are distinguished by the following JACK Agent Language declarations:

```
#handles event PlanChoice event_handle;
#chooses for event event1 event2 ... ;
#chooses for event ... ;
```

The plan's `#handles event` declaration identifies it as one that handles `PlanChoice` events. Since `PlanChoice` events cannot be declared, the *`event_handle`* component will be assigned a reference to any `PlanChoice` event that arises. Therefore, plans that include a `#handles event PlanChoice event_handle;` declaration will handle *all* `PlanChoice` events that arise within the agent unless further methods for discrimination are supplied.

This discrimination is provided by the `#chooses for` event declaration. The `#chooses for` event declaration identifies the events for which the plan's meta-level reasoning is relevant. Therefore, if a meta-level reasoning plan includes one or more `#chooses for` event declarations, the plan will only be relevant for a given `PlanChoice` event if that `PlanChoice` event was caused by one of the listed events. That is, the agent will only use this meta-level plan to choose between multiple plan instances if those plan instances are all applicable to the listed event.

More information on these declarations is supplied in the *Plans* section.

5.3.3 Customising BDI Behaviour with Behaviour Attributes

Each BDI event type has a default behaviour that determines what happens with respect to plan reconsideration, plan set recalculation and meta-level reasoning. This default behaviour can be modified by setting the *behaviour attributes* of the event.

The `#set behavior` declaration is used in BDI event definitions to define how an agent processes an instance of this event when it arises. Options that can be configured include things such as whether the agent will retry the event if an attempted plan fails and whether or not the agent can reason about which plan it executes in response to the event.

Not all BDI events offer all these properties by default. However, each property is present with one or more BDI event. Furthermore, the `#set behavior` declaration can be used to customise the properties of BDI events in each particular application.

Changes to behaviour attributes for BDI Events take two basic forms, as follows;

```
#set behavior attribute value;
#set behavior attribute Type (arg_list);
```

The first form is the most common and is used to set the behaviour of existing BDI events. The second form is used to define custom plan choice events to use for meta-reasoning with the event being defined. Both types are described in the sections that follow.

Note that the exact behaviour generated by setting a particular behaviour attribute can depend on the setting of another behaviour attribute. For example, the behaviour generated by the `Recover` attribute is linked to the setting of the `ApplicableSet` attribute.

With their default values of:

```
#set behavior Recover repost;
#set behavior ApplicableSet new;
```

an event will be reposted on plan failure and the applicable set will be recomputed anew (i.e. possibly with different bindings). Another applicable plan will then be tried (any plan instances already tried and failed will be discarded). However, if the `ApplicableSet` attribute had a value of `once`, the applicable set would not be recomputed before another plan was selected.

#set behavior Recover <value>;

This attribute can be added to a BDI event to define how plan processing should act upon plan failure. Possible values are listed in the following table:

Value	Effect
repost	The event should be reposted on failure, or more precisely, another applicable plan should be tried. If no new applicable plan is found, the event processing fails. [DEFAULT]
never	Event processing should fail rather than recover from plan failure.
always	The event should always be reposted, even when no new applicable plan is found.

Table 5-6: Possible values for the Recover attribute

Note: repost is the default Recover value used by BDIGoalEvent and InferenceGoalEvent, while never is the default for BDIFactEvent, BDIMessageEvent and BDITracedMessageEvent.

#set behavior ApplicableSet <value>;

This attribute can be added to a BDI event to define how to form the applicable set during plan processing, and in particular with respect to recovering:

Value	Effect
new	The applicable set should be computed anew after each plan failure. [DEFAULT]
once	The applicable set is to be computed only once. On event failure, the next applicable plan is selected from the set computed initially for the event.
repeat	The applicable set is computed initially, and then reinstated when exhausted. This is designed to work with the Recover always attribute and allows persistent checking of plans in a computed set until one succeeds.

Table 5-7: Possible values for the ApplicableSet attribute

`#set behavior ApplicableChoice <value>;`

This attribute can be added to a BDI event to define how an applicable plan should be chosen. This setting is overridden by any meta-level reasoning that is in place:

Value	Effect
`first`	The first plan instance generated by the `#uses plan` declaration in the body of the agent or capability is chosen. [DEFAULT]
`random`	A plan is chosen from the applicable set at random.

Table 5-8: Possible values for the `ApplicableChoice` attribute

`#set behavior ApplicableExclusion <value>;`

This attribute can be added to a BDI event to define how plans are excluded from the applicable set:

Value	Effect
`failed`	Plans that have failed are excluded from the applicable plan set. [DEFAULT]
`none`	Plan failure is forgotten immediately, so that plans that have failed previously can be added to the applicable plan set (if they are still applicable).
`rank`	All Plan instances with lesser `PlanInstanceInfo.getRank()` values are excluded from the applicable plan set. [DEFAULT]

Table 5-9: Possible values for the `ApplicableExclusion` attribute

The **two** default values are;

```
#set behavior ApplicableExclusion failed;
#set behavior ApplicableExclusion rank;
```

Note: If the `ApplicableExclusion` attributes are set to `none`, the *reset* plan exclusion properties are effectively reset for this event. After resetting this property, either the `failed` or the `rank` property can be selectively added to further refine plan set exclusion characteristics.

`#set behavior PlanBindings <value>;`

This attribute can be added to a BDI event to define how to process plan context for generating multiple plans instances:

Value	Effect
`all`	Generate all applicable plan instances before plan choice. [DEFAULT]
`single`	Generate one applicable and not failed instance (if possible).
`first`	Try the *context* once for a *relevant* plan type. If that fails, or if the first plan instance has been tried before and failed, no new plan instance is generated.

Table 5-10: Possible values for the `PlanBindings` attribute

`#set behavior OnError <value>;`

This attribute can be added to a BDI event to define how to deal with exceptions from plans:

Value	Effect
`propagate`	Propagate the error up the task stack to where the event was subtasked. [DEFAULT]
`fail`	Capture the error but fail the event processing.
`repost`	Treat the error as plan failure.

Table 5-11: Possible values for the `OnError` attribute

#set behavior PostPlanChoice <*value*>;

This attribute can be added to a BDI event to define under which circumstances the `PlanChoice` event should be posted:

Value	Effect
`never`	Never post a `PlanChoice` event for the event being defined.
`always`	Always post a `PlanChoice` event.
`one_applicable`	Post a `PlanChoice` event if there is only one plan instance in the applicable plan set.
`multiple_applicable`	Post a `PlanChoice` event if there are two or more plan instances in the applicable plan set. [DEFAULT]
`no_applicable`	Post a `PlanChoice` event if there are no plan instances in the applicable plan set i.e. it's empty.
`one_plan`	Post a `PlanChoice` event if there is one relevant plan.
`multiple_plan`	Post a `PlanChoice` event if there are two or more relevant plans.
`no_plan`	Post a `PlanChoice` event if there are no relevant plans.

Table 5-12: Possible values for the `PostPlanChoice` attribute

Setting `PostPlanChoice` to `never` resets all settings for this attribute. All required attributes can then be added.

Note: Several of the above values may be needed to select the desired nuance. For example, if you like a plan choice for one or more applicable plan instances, the declaration would be as follows:

```
#set behavior PostPlanChoice one_applicable;
#set behavior PostPlanChoice multiple_applicable;
```

#set behavior PlanChoiceEvent MyPlanChoice();

This attribute can be added to a BDI event to declare that the event type being defined should use *`MyPlanChoice`* which is constructed inline. The class *`MyPlanChoice`* needs to be defined and it needs to extend class `PlanChoice`.

5.4 Automatic Events

The objective of *automatic events* is to provide a mechanism for an agent to automatically post particular events when certain belief states arise. Automatic events use:

1. A `#uses data` statement. This statement is added to an event definition to provide access to the agent's belief structures.
2. A `#posted when` definition. This definition is added to an event definition to specify the condition which must arise for the event to be automatically posted and optionally, a body which will be used as the posting method if and when the condition should arise.

When an agent or capability declares that it handles an event which has a `#posted when` definition, it will automatically have the necessary belief monitoring activated. The details are described in the section on the `#posted when` declaration.

The use of the `#uses data` and `#posted when` statements to achieve automatic belief monitoring is illustrated in the following example:

```
event Example extends Event
{
     #uses data ReactorVessels rv;
     logical String $vessel;
     logical int $x;
     #posted when ( rv.get($vessel,$x) && $x.as_int() > 300 ) {
         // Any event initialisation goes here
     }
}
```

The block after the `#posted when` condition is optional. If there are no event fields that need to be initialized when the event is actually posted, it can be omitted as shown below:

```
 #posted when ( rv.get($vessel,$x) && $x.as_int() > 300 );
```

The condition is evaluated once initially, and subsequently whenever a change occurs that might affect the condition. If the condition evaluates to true, the event is posted with the logical variables bound as they were in the condition. If there are multiple bindings, a separately bound event will normally be posted for each possible binding. The exception occurs when an event with that particular binding has already been posted to signify the condition becoming true and the condition with that binding has not become false in the meantime.

Suppose that in the above example, `rv` initially contained the following tuples:

```
("vessel1",400) and
("vessel2",500)
```

then two separate events would be posted; one with `$x` = 400 and one with `$x` = 500.

However, if `rv` initially contained no facts and the fact

```
("vessel1",400)
```

was asserted, an event would be posted with `$vessel` = "vessel1" and `$x` = 400. If

```
("vessel2",500)
```

was then asserted, another event would be posted with `$vessel` = "vessel2" and `$x` = 500. However, note that another event with `$vessel` = "vessel1" and `$x` = 400 would **not** be posted because the condition (with `$x` = 400) is still true and had never been false in the intervening period.

If the condition was more complicated (e.g. involved more than one beliefset) and the addition of one fact meant that multiple bindings became true, multiple events would be posted.

5.5 Event Definition

JACK Agent Language events not only support the concepts described in the last section, but also provide type safety within agent definitions. This is because both agents and plans need to declare the events that they handle and the message events that they send. As with function prototypes, this ensures that any type mismatches in an event's parameters can be detected and flagged during compilation.

Event definitions follow one of the formats given below:

```
event EventType extends Event
{
   // JACK Agent Language statements specifying
   // the event's structure and how the event is
   // posted within the agent.
}

event EventType extends MessageEvent
{
   // JACK Agent Language statements specifying
   // the event's structure and how the event is
   // posted to other agents.
}

event EventType extends BDIFactEvent
{
   // JACK Agent Language statements specifying
   // the event's structure and how the event is
   // posted within the agent.
}

event EventType extends BDIMessageEvent
{
   // JACK Agent Language statements specifying
   // the event's structure and how the event is
   // posted to other agents.
}
```

```
event EventType extends BDIGoalEvent
{
   // JACK Agent Language statements specifying
   // the goal's structure and how the goal is
   // posted within the agent.
}


event EventType extends InferenceGoalEvent
{
   // JACK Agent Language statements specifying
   // the goal's structure and how the goal is
   // posted within the agent.
}

event EventType extends PlanChoice
{
   // JACK Agent Language statements specifying the
   // PlanChoice event structure.
   // Note: you are unlikely to want to declare your own
   // PlanChoice events.  You would only do this if you
   // want to extend the meta-level reasoning capabilities
   // supplied with JACK.
}
```

Each part of these definition statements is explained in the following table:

Syntax Term	Description
`event`	JACK Agent Language keyword that identifies the statement as an event definition. It must always be used to define events.
`EventType`	The name that the event will have. The name of an event is analogous to the name of a class in Java.

`extends Event`
`extends MessageEvent`
`extends BDIFactEvent`
`extends BDIMessageEvent`
`extends BDIGoalEvent`
`extends InferenceGoalEvent`
	The same as in normal Java. Each event class defined inherits its underlying properties from a base class. This base class may be: `Event`, `MessageEvent`, `BDIGoalEvent` or `InferenceGoalEvent`.

Table 5-13: Event component definitions

5.6 Event Members and Methods

The event classes described earlier in this chapter include a number of base members and methods. These members and methods are described in the following subsections. In each case, the event classes that contain these members and methods are identified.

public Agent getAgent()

The `getAgent()` method returns the `Agent` that instantiated the event or in other words, the agent which is processing the event.

public String from

The `from` member describes the location of the agent that originally posted the event. This will refer to a different agent than the one returned by `getAgent()` unless the agent posted a message to itself. This member is only available in message events used for inter-agent communication, according to the following table:

Event Type	Member Available
`Event`	No.
`MessageEvent`	Yes.
`BDIFactEvent`	No.
`BDIMessageEvent`	Yes.
`BDIGoalEvent`	No.
`InferenceGoalEvent`	No.

Table 5-14: Availability of the `from` member

Whenever a message event is sent from one agent to another, the name and address of the agent that sent it is automatically assigned to this member. This address is a string that represents both the agent's name and the *portal* at which it is running on the DCI network. Refer to the *Inter-agent Communications* chapter for further details.

Agents that receive a message event can query the `from` member to determine where the message came from. If they want to send a response, they can do so using the `@reply` statement in a reasoning method, or the `reply()` method in ordinary code. Both of these constructs automatically extract the `from` member from the supplied event that originated the interaction.

Because all message events extend from the `MessageEvent` event class, every defined message event class will include this member.

public String message

This member is available only in message events, as shown in the following table:

Event Type	Member Available
`Event`	No.
`MessageEvent`	Yes.
`BDIFactEvent`	No.
`BDIMessageEvent`	Yes.
`BDIGoalEvent`	No.
`InferenceGoalEvent`	No.

Table 5-15: Availability of the `message` member

The `message` member is provided for message events so that they can be traced using the Agent Interaction Diagram (if this JACK option is enabled). Whenever writing a posting method for a message event, a `String` may be assigned to this member. If so, this text will be propagated to the Agent Interaction Diagram, allowing easy identification of the instance of this message event. Refer to the *Tracing and Logging Manual* for details on the Agent Interaction Diagram.

public String mode

This member is available in `BDIGoalEvents` only, as illustrated in the following table:

Event Type	Member Available
`Event`	No.
`MessageEvent`	No.
`BDIFactEvent`	No.
`BDIMessageEvent`	No.
`BDIGoalEvent`	Yes.
`InferenceGoalEvent`	No.

Table 5-16: Availability of the `mode` member

This member holds a `String` that represents the statement used to post a given instance of a `BDIGoalEvent`. By accessing this method, an agent can determine why it is handling a given `BDIGoalEvent` instance. This information is stored in the `mode` member only when `@achieve`, `@determine`, `@insist` or `@test` are used to post the `BDIGoalEvent`. If it did not arise from one of these statements, the mode will be set to `null`.

For example, suppose an agent that drives a networking system needs to derive information about the state of the network before attempting to establish a new connection. When it wants to connect to a new system, it first posts a `BDIGoalEvent` using a `@test` statement. Assuming the connection is successful, the agent would post a `BDIGoalEvent` using the `@achieve` statement.

The plan that handles this event can query the event's `mode` member to determine if the agent wants the connection tested or whether a connection should be established. In each instance the strings "test" and "achieve" would be stored in the `mode` member respectively. In each case, the plan's processing would need to be different.

The strings for each `BDIGoalEvent` posting mode are given in the following table:

BDIGoalEvent Posted by	mode
`@achieve`	`achieve`
`@insist`	`insist`
`@test`	`test`
`@determine`	`determine`

Table 5-17: Posting modes

`Cursor replied()`

This method indicates whether the agent has received any replies to a given message event. It returns a triggered cursor, which will test whether the given message event's reply queue is empty.

A message event can have any number of pending replies. A reply is also a message event, but it must have been sent using a `reply()` method or an `@reply` statement.

If the agent has received at least one reply to the `MessageEvent` that you call this method on, the cursor statement will return `true` when tested.

If the agent has yet to receive a reply to the `MessageEvent` that this method was called on, the cursor statement will return `false` when tested.

Because the cursor is triggered, it can be used in a `@wait_for` statement to make the agent wait for replies before continuing with a given task. Doing this provides the option of implementing synchronous inter-agent messaging, rather than the default asynchronous method. After sending a message event to another agent, a reasoning method can make the task wait on the returned triggered cursor, effectively blocking it until such a reply arrives.

This method is only provided on message events as shown in the following table:

Event Type	Method Available
`Event`	No.
`MessageEvent`	Yes.
`BDIFactEvent`	No.
`BDIMessageEvent`	Yes.
`BDIGoalEvent`	No.
`InferenceGoalEvent`	No.

Table 5-18: Availability of `replied` method

`MessageEvent getReply()`

This method complements the `replied()` method described above. It allows retrieval of a reference to a `MessageEvent` that has been sent as a reply to the `MessageEvent` that it is called on.

When this method is called, it returns the first reply in the reply queue, removing it from the queue at the same time. If there are no message events in the reply queue, it throws an `Error`. This method is only provided on message events as per the following table:

Event Type	Method Available
`Event`	No.
`MessageEvent`	Yes.
`BDIFactEvent`	No.
`BDIMessageEvent`	Yes.
`BDIGoalEvent`	No.
`InferenceGoal`	No.

Table 5-19: Availability of the `getReply` method

5.7 Event Declarations

Events have a small number of JACK Agent Language specific constructs at the field or member level. These constructs are described in the following subsections.

`#posted as` *`methodName`*`(`*`parameters`*`)`

This is one of the event's *posting methods*. A posting method describes how the event can be constructed and posted or sent.

An event's posting method must be used whenever an instance of the event needs to be created. This is normally when agent or beliefset code executes the `post()` method, or a plan uses the `@post` or `@subtask` statements. The posting method describes everything that the agent needs to do to construct an instance of the event.

A posting method declaration is as follows:

```
#posted as methodName (parameters)
{
   // Method Body
}
```

Each part of this declaration is described in the following table:

Component	Meaning
`#posted as`	Indicates that an event's posting method is being defined. All posting method definitions must be identified in this way.
`MethodName`	The name by which the posting method is identified. When the agent posts an event, it uses this name to identify the posting method to be used.
`(`*`parameters`*`)`	Identifies the number and type of parameters that this posting method requires in order to construct and post the event.
`Method Body`	Java code that is used to construct the event.

Table 5-20: Posting method declaration components

An example event definition including a posting method is given below:

```
// An example event definition to be used with a
// prospecting agent. The agent models a prospector who
// travels around looking for treasure. This event is
// posted whenever some treasure is found (presumably so
// that the agent will execute a plan to retrieve it).

event FoundTreasureEvent extends Event
{
   int latitude;
   int longitude;
   int value;

   #posted as foundTreasure (int lat, int ltd, int val)
   {
        latitude = lat;
        longitude = ltd;
        value = value;
   }
}
```

In this example, the posting method's name is `foundTreasure`. Whenever the agent believes it has found some treasure while executing its own methods, it can use the `foundTreasure` posting method to create a `FoundTreasureEvent` which it can post or send to initiate a response.

The `foundTreasure` posting method requires three arguments if the event is to be posted successfully. These are the treasure's latitude, longitude and value. All that the posting method's body does is populate the event's data members with this information so that it will be included in the posted event.

An event may have as many posting methods as you want. You may use different posting methods when instances of the event should take a different form under different circumstances. In particular, you may want to implement multiple posting methods using the idea of a *polymorphic class* which allows you to use the same posting method name while defining different versions of the posting method for different arguments.

For example, suppose an agent that keeps track of a racing leaderboard gets messages sent to it by another agent whenever each racer completes a lap. This message may take two forms: one when the racers have completed a lap during the race, and another when the racers have completed the final lap. In each case, the message may convey different information about the racer. The agents can use the same message event for each situation, but a different posting method to propagate the event instance with the required information in each case.

Just as you can extend the base `Event` classes you can also extend your own event definitions. You may choose to define generic event types and then extend them to more specific cases. For example, a generic `FoundTreasureEvent` might be used as the base class for a `FoundDiamondEvent` and a `FoundGoldEvent`). When you choose to do this, you should only define posting methods for the outermost (leaf) event classes.

You may wish to extend your own event classes if, for example, you require:

- improved efficiency in terms of plan selection (the events are more specific),
- different behaviours (using `#set behavior`),
- additional members.

`#uses data` *DataType data_name*

The `#uses data` statement in an event definition has the following form:

```
#uses data DataType data_name;
```

where the *DataType* and *data_name* match the data or beliefset declaration in the enclosing capability or agent. This statement is added to an event definition to provide access to the enclosing capability or agent's belief structures.

`#posted when (` *condition* `)` *optional_method_body*

The `#posted when` statement has the following form:

```
#posted when ( condition ) { body }
```

where the `condition` is a logical condition of the same form as a condition for a `@wait_for statement`. In particular, it must be a triggered condition, i.e. include conjuncts that are triggered, such as beliefset cursors etc.

Note that the `elapsed()` and `after()` methods are plan methods and thus not available for events – if there is a requirement for issuing a regular recurrent event stream, the use of a plan is recommended.

The body after the condition is optional and consists of any initialisation that should be done if and when the event is posted. The body is the same as that of a normal posting method declared using `#posted as`. If no initialisation is needed, the entire body (including the braces) may be omitted.

If an event definition includes a `#posted when` statement, the compiler deals with logical variables in the event by providing a logical environment in which to evaluate the condition, and later providing new logical environments and bound variables for the automatic events being posted.

An event definition may include several `#posted when` and any number of `#posted as` statements just like any other event. The former will then issue automatic events according to their conditions, and the latter would be used exactly as before.

`#set transport` *format*

The `#set transport` statement is used to signal which transport format is to be used for message events. It has the following form:

```
#set transport format
```

where `format` is either `jacob` or `java`. Since release 4.1, the default format is `jacob`. Although `jacob` is generally more efficient for transport, there are some things that `jacob` does not support (e.g. arrays). In such cases, `java` format should be specified using the `#set transport` declaration,

If the `#set transport` statement is used to specify `java` format, the event is compiled to use Java serialization.

`jacob` format provides support for transport in binary, text or XML style. The default style with `jacob` format is binary. It is possible to specify the style processwide by setting the property `JACOB.OutputType` to one of `(ascii,binary,xml)`.

5.8 Posting and Sending Events

In JACK, posting methods create instances of the events to be posted or sent. These instances are then provided as arguments to the reasoning statements or methods which are responsible for their posting or sending (e.g. `post()`, `send()`, `@post` ...). The event's posting methods are accessed through the event reference provided in the `#sends` event or `#posts` event declaration. Note that there is only one `#sends` or `#posts` event declaration for a given event type in any agent or plan. Event instances are created in an agent or plan by using the event's posting method(s). The process is illustrated below.

```
#sends event MessageEventType mef;
    :
    :
// create event
MessageEventType me = mef.postingMethod1(arg1, arg2);

// send the event
send(destination1, me);

// create and send another event
send(destination2, mef.postingMethod2(arg3, arg4, arg5));
    :
    :
```

6 Inter-agent Communications

6.1 Introduction

JACK provides a runtime networking environment on which different agent processes can operate. Agents can address messages (`MessageEvents`, and `BDIMessageEvents`) to one another by specifying the name of the destination agent and, if applicable, its portal and host. The JACK runtime network then takes care of routing this message to its desired destination.

6.2 Local Communication

Not all agents need to run in independent JACK processes. Often more than one agent will share the same process. When this is the case, the routing of messages between them is trivial. All that the source agent needs to know is the destination agent's name to send the message accordingly.

For agents to communicate with one another, the following requirements must be met:

- The agents must know one another's name.

  Strictly, this is only required by the agent that sends the message event, because when message events arrive they contain the name of the sending agent in the `from` data member.

- The source agent needs to be able to send a message event.

  This is achieved by including a `#sends event` declaration in the agent's definition for the required class of message event. Note that the message event must also have been defined.

  The source agent must then include code to send the message event. It can do this by calling the `send` method from code outside a reasoning method, or the `@send` statement from within a reasoning method.

- The destination agent needs to be able to handle this message event.

  This is achieved by including a `#handles event` declaration for this message event in the destination agent's agent definition. To handle this event correctly, the destination agent must also include at least one plan with the same `#handles event` declaration, and declare that it uses this plan via a `#uses plan` declaration.

6.3 Remote Communication

When agents are running in separate processes, The JACK communications layer needs to be used to allow these agents to communicate. This communications layer is known as the DCI network.

The DCI network is layered in such a way that different underlying transport mechanisms can be accommodated. By default, JACK uses the UDP transport protocol which is available on existing TCP/IP networks. This protocol is fast but connectionless and it does not guarantee delivery, so JACK provides a thin layer over UDP which provides reliable peer-to-peer communications.

Note: No call to any networking code is made unless a connection is actually requested or made possible (by specifying an explicit port number for the local portal). This means that machines with no networking installed will still be able to use JACK locally.

When agents in remote processes need to be able to communicate, the processes need to be told how to communicate with one another. Each process has its own *portal* and these connect to each other to provide a communications link for the agents in each process. A pair of portals can be connected explicitly, or a name-server can be used so that portals can connect on the fly as needed.

The full name of an agent takes the form *agent*`@`*portal*, where `agent` refers to the name given to an agent on creation and `portal` refers to the named portal assigned to the process in which the agent is running. The process may be running on a remote host or the local machine. To avoid ambiguity, it is recommended that the full name of an agent be used at all times when sending messages.

The easiest way to set up a DCI network is to designate one of the processes as a *name-server.* This simply means that whenever an agent tries to send a message to an agent at an unknown *portal*, the agent will query the name-server to try to locate the portal. If the name-server knows about the unknown portal, a connection to it is silently established. Otherwise, the message will be undeliverable and the send request will fail.

If there is no designated name-server, all connections must be explicitly requested. Any explicit connection request must be made by one (and only one) of the two processes comprising each connection.

Processes automatically keep their name-server(s) informed of new connections that they make. When a process makes an initial connection to a name-server, the name-server will update its list of known addresses. This can be used to implement a robust network containing two or more name-servers so that no information is lost if one name-server goes down and subsequently comes up again.

If a process exits for any reason, it can be restarted and communication will resume from the restart time. Any messages sent while the process was not running will be lost.

6.3.1 DCI from the Command-line

Portals, name-servers and connections can be established on the command line, provided that a call has been made to the JACK initialisation method:

```
aos.jack.Kernel.init(String[] args)
```

The simplest example is connecting two processes so that agents in each of them can communicate with each other. Suppose one of them is called `Speaker` and the other is called `Listener`. To connect them using a name-server, the two processes would be run as follows:

```
java Listener "-dci.new:listener=xxxx"
java Speaker  "-dci.new:speaker" "-dci.ns:xxxx"
```

The first process creates a portal called *`listener`* bound to UDP port *`xxxx`*.

The second process creates a portal called *`speaker`*. It did not specify a particular port so a random port will be allocated for it. It also specifies a name-server, so it will connect to the name-server portal immediately.

Note: If the processes were running on different machines, the UDP port could have been specified as *`host`*:*`xxxx`* instead of just *`xxxx`*. Whenever the host is omitted in this way, the local host is assumed.

To connect the two processes using an explicit connection, the two processes would be run as follows:

```
java Listener "-dci.new:listener=xxxx"
java Speaker "-dci.new:speaker" "-dci.con:speaker->listener=xxxx"
```

As in the name-server example, the *`listener`* portal did not bother to choose a specific UDP port because there was no need. However, the *`listener`* portal had to specify a particular port so that *`speaker`* would know where to connect to.

Note: When connecting explicitly in this way, only one of the processes initiates the connection. Once connected, the communications path is bidirectional and completely symmetrical. Any unsuccessful connection attempts will timeout in 30 seconds by default.

An application can create a portal without the use of the `-dci.new` command line option. By default the portal associated with an application has name `%portal`, host `localhost`, and a port number that is the next available port. However, if the Java properties `jack.portal.name`, `jack.portal.host` or `jack.portal.port` are set, then these values will be used instead. This means that if the portal is being created solely for tracing purposes, (refer to the *Tracing and Logging Manual*), then there is no need to include an `init(args)` call in the `main()` method of the application.

6.3.2 DCI Command Line Summary

Argument	Purpose
`-dci.new:`*`A`*`=`*`host`*`:`*`port`*	Creates a new portal called *`A`* for this process. The *`host`* and *`port`* are optional.
`-dci.ns:`*`host`*`:`*`port`*	The given location (*`host`* and *`port`*) is designated as a name-server. The *`host`* part is optional.
`-dci.con:`*`A`*`->`*`B`*`=`*`host`*`:`*`port`*	The local portal named *`A`* requests a connection to the portal named *`B`* which can be found at the given location (*`host`* and *`port`*). The *`host`* part is optional.
`-dci.timeout=`*`seconds`*	This changes the default connection timeout from 15 seconds to the specified value. Use a timeout of 0 to wait indefinitely.

Table 6-1: DCI command line summary

6.3.3 DCI in Code

The above DCI command line functionality can also be achieved in the JACK code through the following static methods in the class `aos.jack.jak.core.Dci`.

`void create(String` *`name`*`, String` *`desc`*`)`

The `create` method corresponds to the `-dci.new` command line argument. This method needs to be called before the first agent is created. *`name`* is the name of the portal, and *`desc`* is the description string that follows the '=' sign on a command line.

`void connect(String` *`lname`*`, String` *`rname`*`, String` *`rdesc`*`)`

The `connect` method corresponds to the `-dci.con:P1->P2=host:port` command line argument. This method can be called at any time to establish a connection. *`lname`* is the local portal's name, *`rname`* is the remote portal's name, and *`rdesc`* is the description string that follows the '=' sign on the command line.

`void nameserver(String` *`rdesc`*`)`

The `nameserver` method corresponds to the `-dci.ns:host:port` command line argument. This method can be called at any time to establish a new, additional name server to use. *`rdesc`* is the name server specification string that follows the "`-dci.ns:`" on the command line.

`void setTimeout (int seconds)`

The `setTimeout` method corresponds to the `-dci.timeout` command line argument. This method can be called at any time to change the timeout period for the next connection request.

Note that all of the above methods except `setTimeout()` throw a `DciException`.

In addition, the following static methods are available to determine whether a particular agent is running and accessible on the DCI network.

`boolean pingOk (String agent)`

This form of the ping method performs one ping only and does not attempt any connection. It does not distinguish between different types of failure and simply returns 'true' on success and 'false' on any kind of failure.

`int ping (String agent)`

This form of the ping method performs one ping only and does not attempt any connection. This method times-out after 30 seconds. The possible return values are described in the table at the end of this section.

`boolean multiPingOk (String agent)`

With this version of `multiPingOk`, if the given portal is not connected but a nameserver is present, then a connection attempt is made before the ping is attempted. One ping is attempted and if it is unsuccessful, more pings are attempted at intervals of 1 second for a total waiting period of 30 seconds. It returns 'true' or 'false' for success or failure.

`boolean multiPingOk (String agent, int timeout, int interval)`

This version of `multiPingOk` is like `multiPingOk(String)` but it allows the timeout and interval to be specified.

`int multiPing (String agent, int timeout, int interval)`

This version of `multiPingOk` is like `multiPingOk(String)` but it allows the timeout and interval to be specified plus it returns the same values as `ping(String)` which are described in the table at the end of this section.

The possible return values from the methods returning `int` are described in the following table:

Return value	Meaning
`Dci.PING_NOT_READY`	There is no portal present in the current process.
`Dci.PING_UNKNOWN_PORTAL`	The given portal is not connected.
`Dci.PING_UNKNOWN_NAME`	The given name is not known at the given portal.
`Dci.PING_OK`	The given name at the given portal responded.

Table 6-2: Return values from `ping` and `multiPing`

These values are defined in `aos.dci.Portal`.

7 Plans

7.1 What is a Plan?

The `Plan` class describes a sequence of actions that an agent can take when an event occurs. Whenever an event is posted and an agent adopts a task to handle it, the first thing the agent does is try to find a plan to handle the event.

Plans can be thought of as pages from a procedures manual, or even as being like methods and functions from more conventional programming languages. They describe, in explicit detail, exactly what an agent should do when a given event occurs. Equipped with a set of plans, an agent has a set of skills and procedural knowledge that it can draw upon as required. When the event that a plan addresses occurs, the agent can execute this plan to handle it.

Each plan is capable of handling a single event. The event it can handle is identified by the plan's `#handles event` declaration. When an instance of a given event arises, the agent may execute one of the plans that declare they handle this event.

An agent may further discriminate between plans that declare they handle an event by determining whether a plan is *relevant*. It does this by executing the `relevant()` method of each plan. If plans do not specify a `relevant()` method, they are relevant for all instances of that event. When present, the `relevant()` method lets a plan specify exactly which instance of a given event it is relevant for.

Once the relevant plan(s) have been identified, the agent determines which of these are *applicable*. It does this by executing the plans' `context()` method. The context method is a JACK Agent Language *logical expression* that can bind the values of the plan's *logical members*. For every possible set of bindings, a separate applicable instance of the plan is generated.

When the agent has found all applicable instances of each relevant plan, it selects one of these to execute. If the event is a BDI event, this may cause a `PlanChoice` event to be posted, which may initiate some meta-level reasoning within the agent to select the most appropriate plan instance. Refer to the *Meta-Level Reasoning* chapter for more details on meta-level reasoning.

When the agent executes a plan, it starts by executing the plan's `body()` method. The body method is a special kind of method in the JACK Agent Language called a *reasoning method*. Reasoning methods are quite different from ordinary methods in Java, as they are bound by extra logical rules and conditions. Each statement in a reasoning method is treated as a logical statement that can either succeed or fail.

Failure of a plan statement will cause the `body()` method to fail unless the plan specifically allows for this possibility. If execution proceeds to the end of the `body()` method, the `body()` method succeeds.

The `body()` method can call other reasoning methods as it executes. These reasoning methods must be declared in the plan using the `#reasoning method` declaration. They help break complex processing down into smaller components and make plans both simpler to understand and more scalable.

All reasoning methods can include JACK Agent Language *reasoning method statements*. These statements are identified by a preceding `@` symbol, and describe logical behaviour that the reasoning method should adhere to. If a reasoning method statement is violated the reasoning method also fails. Reasoning methods execute as *Finite State Machines* (FSMs).

7.2 Finite State Machines

Finite State Machines (**FSM**s) are a standard concept in computer science. They describe execution engines that represent calculations and state changes in atomic, indivisible steps. In a finite state machine, execution may be suspended or switched, but only at designated points. Each step in the execution is guaranteed to be atomic, and once started will not be interrupted until it is complete. These atomic steps are normally quite small, but nonetheless the atomic nature of each step is assured.

Finite State Machine statements (or FSM statements) are significant in the JACK Agent Language because Java is a multi-threaded language where switching between execution threads is normally the responsibility of the programmer. The Java execution engine does not guarantee safe points at which it will switch execution threads, so it is up to the Java programmer to implement object locking and any other concurrency controls which may be required.

With FSM statements, however, multi-threading is taken care of by the JACK kernel. Each statement is executed in a series of atomic steps, between which the agent can switch to other execution threads safely. All reasoning methods and task executions in JACK are FSMs, as are statements that execute subtasks synchronously with the parent task (such as `@subtask`, `@achieve`, `@insist`, `@test` and `@determine`). This ensures that the statements they contain execute in safe, atomic steps. These steps are not always whole statements: sometimes a single statement may consist of multiple component steps.

Note: Because threading in FSMs is handled by the JACK kernel, no programmer-level multi-threading should be performed in an FSM statement. This would not only be superfluous, but may also hamper the efficiency of the underlying kernel multi-threading.

7.3 Plan Definition

The minimal format for the definition of a plan is given below:

```
plan PlanName extends Plan
{
   #handles event EventType event_ref

   // Plan method definitions and JACK Agent Language
   // #-statements describing relationships to
   // other components, reasoning methods, etc.

   body()
   {
      // The plan body. This describes the actual steps
      // an agent performs when it executes this plan.
   }
}
```

Each component of this definition is explained in the following table:

Component	Description
`plan`	A JACK Agent Language keyword used to introduce a plan definition.
`PlanType`	The plan's name.
`extends Plan`	Plays the same role as in Java – it indicates that the plan being defined inherits from a JACK Agent Language base class called `Plan`. The `Plan` base class implements the plan's base methods and the underlying functionality to support the plan's core behaviour, such as reasoning methods and reasoning method statements.
`#handles event()`	Specifies the event type that this plan handles. The plan may place further constraints on its applicability via `relevant()` and `context()` methods.
`body()`	Describes the actual work done by an agent when the plan is executed. It is the plan's top-level reasoning method: if it succeeds, the plan succeeds and if it fails, the plan fails.

Table 7-1: Components of a Plan definition

7.4 Plan Members and Methods

Like agents and events, plans can include normal Java members and methods. The user may choose to add these to implement certain low-level functionalities.

Some members and methods are supplied automatically in JACK. These members and methods enable the user to:

- identify which agent a plan belongs to,
- determine when a plan is *relevant*,
- determine when a plan is *applicable*;
- implement the core functionality of the plan, and
- define properties that an instance of a plan has to assist in meta-level reasoning.

Each of these members and methods are summarised in the following table and are described in more detail below:

Method	Purpose
`Agent getAgent()`	Identifies the agent which instantiated the plan. Using the `#uses agent implementing` declaration and the `#uses interface` declarations enables a `Plan` class to be shared between agents.
`relevant()`	Determines whether the plan is relevant to a particular kind of event.
`context()`	Determines whether the plan is applicable to a particular event occurrence.
`body()`	Describes what the agent must do when it executes the plan. It is the plan's top-level reasoning method and must always be present.
`getInstanceInfo()`	Allows details to be extracted on any properties that have been defined for a plan instance. These property details can be compared in meta-level reasoning plans to determine which instance to execute.
`after()` `afterMillis()`	These methods create cursors that become true immediately after a specified time.
`elapsed()` `elapsedMillis()`	These methods create cursors that become true immediately after a specified time period has elapsed.

Table 7-2: Plan methods

`Agent getAgent()`

`getAgent()` can be used anywhere within a plan to access the instantiating agent. It can be used within a plan to access the agent's `timer` member and the `name()` method. It can also be used to access user-defined agent methods and members.

Note that `getAgent()` has a return type of `Agent`. An explicit cast is then required to provide access to user-defined agent methods and members. A preferred alternative to `getAgent()` is to use the `#uses interface Interface reference` declaration which provides access to the enclosing agent with the correct type cast. This is described in more detail in the section on plan declarations in this chapter.

`relevant(`*`EventType`*`)`

Relevance is the first filter that the agent applies when determining which plan to execute when a given event occurs. To be relevant, the plan must declare that it is capable of handling the kind of event that has arisen (via the `#handles event` declaration) and that it is relevant to the event instance (via the `relevant()` method).

Not all plans have a `relevant()` method. When present, it provides the agent with a filter to exclude plans that will definitely not be able to handle the event. For example, if the plan is only relevant when certain parameters are passed, or when some parameters have specific values, this can be tested by the `relevant()` method. If the event is not one that the plan can handle, the agent can put it aside immediately. The `relevant()` method provides a finer granularity for determining relevance than just the distinction between events. It can actually allow the agent to have plans to discriminate between instances of the same event when that event has different parameters types, or different values for a given parameter.

The `relevant()` method always takes the following form:

```
static boolean relevant (EventType event_ref)
{
   // Code to determine if the plan is
   // relevant to this event.
}
```

That is, it must always be a static boolean method. If this method returns true, the plan is relevant to the event. If not, the plan is not relevant to the event.

The body of the `relevant()` method can perform any Java processing you want. Usually, however, it will test one or more of the event's members to make a decision.

For example, suppose a prospecting agent that travels around looking for treasure has a plan that describes how to call in a helicopter to pick up some treasure it has found. Suppose also that the agent includes a `FoundTreasureEvent` which is generated when treasure is discovered. If the helicopter costs $500 to call in, the plan would only be relevant if the treasure found is worth more than $500. The `relevant()` method can check the value of the `FoundTreasureEvent`'s `val` field and if it is greater than $500, return `TRUE`.

Using a `relevant()` method in this way, the agent can filter out those plans it knows it won't want to execute before proceeding with the applicability calculations described in the following sub-section. Typically, these applicability calculations are more resource-intensive, so well-thought out `relevant()` methods can improve the computational efficiency of the application.

The `relevant()` method is especially useful if an event has more than one posting method. If different posting methods set different properties and provide different number of parameters, this method can be used to catch instances of the event posted by one posting method and not the others.

For example, if an agent keeps track of racers, and this agent posts an event whenever a racer completes a lap, the event may take two forms. One form may represent the completion of a lap during the race, whereas the other may represent the completion of the final lap at the end of the race. If the plan is to compose a final leaderboard, this plan will only be relevant when the final lap is complete, not for the intermediate laps. By testing the parameters supplied with each lap instance event, the plan can identify the event for which it is relevant.

context()

The context method is the next filter that the agent applies when determining which plan to execute when an event occurs. The context method usually contains a logical condition that tests one or more of the agent's beliefset relations and/or data members. In many respects, it represents the core of the agent's simulated rational behaviour, because it takes into account the *current circumstances* (identified by the current values of respective members and data structures), and the agent *current beliefs* (represented by its beliefset relations), and uses this information to select a plan instance that is *applicable* to the current conditions.

The plan instance is a copy of the plan with particular values for its members. These members may include ordinary Java members and special *logical members*. Instead of being assigned values like normal Java members, logical members are bound to particular values by a process of unification (attempting to match them with other known or partially known values). The semantic behaviour of logical methods is quite different to that of Java methods; therefore, logical methods include specific JACK Agent Language statements and expressions to manipulate them and their values.

The `context()` method always takes the following form:

```
context()
{
   // Logical condition to determine which plan instances
   // are applicable.
}
```

That is, it does not take any arguments and its body is always a single JACK Agent Language *logical expression*. Logical expressions are statements consisting of boolean members, logical members and beliefset cursor expressions.

When evaluating the `context()` method, the agent will consider all possible alternatives. For every set of values that can satisfy the `context()` method, a separate instance of the plan will be generated and available for execution. For example, a context method that has 5 possible bindings will cause 5 separate plan instances to be produced.

This is like a shopping agent being told to buy a piece of fruit. If a shop sells apples, oranges and bananas, buying any one of these will handle this event. The agent would have three instances of the fruit-buying plan available: one to buy an apple, one to buy an orange and one to buy a banana. The plan instance chosen depends on the event model being used (refer to the *Events* section for details). The agent will have achieved its objective if the chosen plan instance succeeds.

body()

The `body()` method is the main reasoning method in a plan and is executed whenever a plan is executed. The `body()` method is analogous to the `main()` method in Java in that it represents the starting point in a plan's execution.

Because a plan's body is a reasoning method, its execution structure is not the same as for an ordinary Java method. Firstly, it can contain a number of JACK Agent Language statements that can only appear in reasoning methods. These statements are all preceded by the `@` symbol and are described in the *Reasoning Method Statements* section below.

Secondly, reasoning methods are like 'big logical expressions' in that when the agent executes them, it is constantly thinking 'is this true?'. Each statement in the `body()` method is, therefore, treated as a boolean expression, and each semicolon between statements as an AND connector. The `body()` method is true (succeeds) if, and only if, execution reaches the end. If any statement fails, the `body()` method terminates immediately and fails.

For example, if a plan's `body()` method can be broken down as follows:

```
body()
{
   statement A;
   statement B;
   statement C;
   statement D;
}
```

This method will only succeed if *`statement A`* **and***`statement B`* **and** *`statement C`* **and** *`statement D`* succeed. If *`statement B`* was to fail, for example, the agent would stop executing the `body()` method at this point and the plan instance would fail (meaning that, in this case, *`statement C`* and *`statement D`* are never attempted).

It is not necessary for every statement in the reasoning method to succeed for the method to succeed. Only the entire boolean expression represented by each statement needs to be true. For example, if the reasoning method contains the block:

```
if (condition A)
   statement B;
else
   statement C;
```

If *condition A* fails, the `body()` method will not fail because there is an alternative execution path to follow – the `else` clause. However, if *B* or *C* fail, the plan still fails.

PlanInstanceInfo getInstanceInfo()

This callback is used to retrieve information about the instance of a plan that it is called on. It has been provided for use in meta-level reasoning plans, so that when the agent is faced with multiple applicable plan instances, it can use the information returned to help choose between them.

By default, this callback is undefined. When a plan is written to handle a BDI event for the agent to be able to perform meta-level reasoning with, this method should be re-implemented to provide the meta-level reasoning plan with information that it can use to make the decision.

For example, consider a soccer playing agent that has a number of plans which describe field tactics it can use. Each of these plans might have an aggression property added to them for meta-level reasoning purposes. That way, when more than one of these field tactic plans are applicable, the agent's meta-level plan to choose between them can select the plan instance with the highest aggression rating.

Cursor after(double *t*),afterMillis(long *t*)

These methods create cursors that become true immediately after a specified time. *t* specifies the time as measured by `agent.timer`. The time is specified in seconds for `after()` and in milliseconds for `afterMillis()`. Note that their argument types are not the same. If the user needs to use a timer other than `agent.timer`, variants of both methods exist which accept a second argument of type `aos.util.timer.Timer`.

These methods are commonly used within `@wait_for` statements, as illustrated by the following plan fragment:

```
//wait until half past the hour

long now = agent.timer.getTime();
long t = now / 3600000; // convert to hours (truncated)
t *= 3600000;           // convert back to millis (on the
                        // last hour)
t += 30*60000;          // add the after-the-hour offset
if ( t < now )
   t += 3600000;        // Adjust to the next hour if necessary
@wait_for(afterMillis(t));
```

`Cursor elapsed(double t),elapsedMillis(long t)`

These methods create cursors that become true immediately after a specified time period has elapsed. `t` specifies the time period as measured by `agent.timer`. The time period is specified in seconds for `elapsed()` and in milliseconds for `elapsedMillis()` – note that their argument types are not the same. If the user needs to use a timer other than `agent.timer`, variants of both these methods exist which accept a second argument of type `aos.util.timer.Timer`.

These methods are commonly used within `@wait_for` statements, as illustrated by the following plan fragment:

```
//wait for 10 minutes to pass
@wait_for(elapsed(10.0*60));
```

7.5 Plan Declarations

`#chooses for event Event1 Event2 ...`

When a plan contains one or more `#chooses for event` declarations, this means that the plan is used for **meta-level reasoning**. Instead of describing what the agent should do to achieve an end, it describes how the agent should choose between a number of applicable plan instances.

Meta-level reasoning describes the ability that an agent has to choose between applicable plans for a given BDI event. Normally when more than one plan instance is applicable for a given event, the agent just chooses one at random. However, with meta-level reasoning the agent can be more discerning. Instead of choosing an applicable plan instance to try at random, the agent can invoke a plan to make this decision in a specific, deterministic way. Plans that do this are known as *meta-level plans*.

Each component of the `#chooses for event` declaration is described in the following table:

Component	Meaning
`#chooses for event`	Identifies the declaration as a `#chooses for` event declaration.
`event1 event2 ...`	Lists the BDI events that the plan can choose for between multiple applicable plan instances.

Table 7-3: Components of the `#chooses for event` declaration

The `#chooses for event` declaration applies only to meta-level plans. When present, it identifies the BDI events that the meta-level plan is able to choose between. This is necessary because meta-level plans must all handle the internal `PlanChoice` event. They all contain the following `#handles event` declaration:

```
#handles event PlanChoice event_handle;
```

This declares that the agent handles any `PlanChoice` event. To narrow down the range of PlanChoice events that the plan can handle, use one or more `#chooses for event` declaration. Only those `PlanChoice` events that apply to one of the listed BDI events will be handled by the meta-level plan.

For example, suppose a plan definition contains the following declarations:

```
plan ChooseFieldStrategy extends Plan
{
   #handles event PlanChoice ev;
   #chooses for BDIMidfieldPlay BDICornerPlay BDIKickoff;

   // Other declarations and reasoning method definitions
}
```

This identifies the `ChooseFieldStrategy` plan as a meta-level reasoning plan for the BDI events: `BDIMidFieldPlay`, `BDICornerPlay` and `BDIKickoff`.

#handles event *EventType reference*

Most #-declarations are optional in a plan definition, but the `#handles event` declaration is mandatory. It specifies the event that the plan handles. Whenever an instance of this event occurs, the agent will consider this plan as a candidate response. Unless the plan's `relevant`() method says otherwise, the agent will assume that this plan is relevant to the event.

Hence, without a `#handles event` declaration, a plan would never be executed by an agent. Regardless of the event that arose, the agent would never deem the plan to be relevant.

Each plan definition must have exactly one `#handles event` declaration. It is not possible for a single plan to be able to handle more than one kind of event. If this situation can arise in the model, multiple copies of the plan will need to be defined, each specifying one of the events as its relevant event.

The `#handles event` declaration takes the following form:

```
#handles event EventType reference;
```

Each component in this declaration is described in the following table:

Component	Meaning
`#handles event`	Identifies the declaration as being of an event that the plan can handle.
`EventType`	The type of event that the plan can handle. The JACK runtime ensures that any agent which uses this plan also includes a `#handles event` declaration identifying the same event type. An agent can only handle an event if it has a plan to do so. Equally, a plan is only ever of use to an agent if the agent declares that it handles the event handled by the plan.
`reference`	The name, or handle, that will be used to identify the event in the plan. When any of the plan's reasoning methods wants to access one of the event's members or methods, it uses this event reference.

Table 7-4: Components of the `#handles event` declaration

`#posts event` *EventType reference*

The `#posts event` statement declares that the plan is able to post events of this type when executed. This may be in the `body()` method, or one of the other reasoning methods. Note that the plan does not have to post this event every time it is executed, it merely has the capacity to do so.

Not all events must be declared with the `#posts event` statement – only those that the agent posts **internally**. Events that the agent posts externally must be explicitly declared too, but this is done using the `#sends event` statement instead.

A `#posts event` declaration takes the following form:

```
#posts event EventType reference;
```

Each component is explained in the following table:

Component	Meaning
`#posts event`	Identifies a (goal) event declaration. When an agent executes this plan, it may cause events to be posted which it will have to handle.
`EventType`	Identifies the type of event that can be posted. *`EventType`* must be included in the agent's event definitions.
`reference`	The name, or handle, used to identify the event in the plan. The plan can use this handle to access the event's members and methods.

Table 7-5: Components in the `#posts event` declaration

The `#posts event` declaration is similar to a function prototype in C. Technically, it is redundant since the agent that the plan belongs to has already declared all the events it can handle using its own `#handles event` statements. However, explicitly declaring the events that a plan posts ensures a correspondence between the events that an agent's plans can post and the events that it can handle. There should be no events posted by the plan that the agent cannot handle.

By explicitly declaring handled events at both the agent and the plan level, the compiler can check that there are no 'dangling events' that the agent will ignore. This is in line with the explicit definitions in modern programming languages and helps protect programmers from runtime problems that are hard to locate.

`#posts event` declarations identify the normal events and goal events that an agent posts. Message events that the plan can post are declared using the `#sends event` construct described below.

`#sends event` *MessageEventType reference*

The `#sends event` declaration is just like the `#posts event` declaration, except that it identifies message events that the plan can send to other agents. A plan can only execute statements that send these events to other agents if the plan's definition includes a `#sends event` declaration for that kind of event.

A `#sends event` declaration takes the following form:

```
#sends event MessageEventType reference;
```

Each component is explained in the following table:

Component	Meaning
`#sends event`	Identifies a message event declaration. When an agent executes this plan, it may cause a message event to be posted.
`MessageEventType`	Identifies the type of message event that can be posted. *`MessageEventType`* event must be included in the agent's event definitions.
`reference`	The name, or handle, used to identify the event in the plan. The plan can use this handle to access the event's members and methods.

Table 7-6: Components in the `#sends event` declaration

The `#sends event` declaration is similar to a function prototype in C. It specifies at the outset the range of message events that the plan can post. Only those events specified in `#sends event` declarations can be posted by statements in the plan.

The advantages of the `#sends event` declaration are the same as those of function prototypes in C: they explicitly identify the events that the plan can post and hence make the plan easier to port to other agents and allow for compiler checking that can reduce programmer error.

`#uses data` *DataType reference*

The `#uses data` statement identifies those objects or beliefset relations that a plan uses (either to read or to modify). To be accessed this way from within a plan, the Java object or beliefset relation must have been declared in the enclosing agent or capability.

A `#uses data` statement takes the following form:

```
#uses data DataType reference;
```

Each item in the definition is described in the following table:

Component	Meaning
`#uses data`	Identifies a data object or beliefset relation that the plan can use.
`DataType`	Identifies the type of the user-defined data structure or beliefset relation that the plan can access.
`reference`	The name that the data object or relation will be known by in the plan.

Table 7-7: Components in the `#uses data` declaration

For more information on beliefset relations and how to use them, refer to the *Beliefset Relations* chapter.

`#reads data` *DataType reference*

The `#reads data` declaration indicates to the user that the Java object or JACK beliefset of type *`DataType`* is to be accessed within the plan. The Java object or beliefset must have been declared in the enclosing agent or capability. Note that the read only constraint is not enforced by JACK.

A `#reads data` declaration takes the following form:

```
#reads data DataType reference;
```

Each item in the definition is described in the following table:

Component	Meaning
`#reads data`	Identifies a JACK beliefset or user-defined data structure that the plan can access.
`DataType`	Identifies the type of the beliefset relation or the user-defined data structure that the plan can access. The relation or object of this type must have been declared in the enclosing agent or capability.
`reference`	The name that the data object or beliefset relation will be known by in the plan.

Table 7-8: Components in the `#reads data` declaration

For more information on beliefset relations and how to use them, refer to the *Beliefset Relations* chapter.

`#modifies data` *`DataType reference`*

The `#modifies data` statement indicates to the user that a Java object or JACK beliefset of type *DataType* is to be modified within the plan. The object or beliefset must have been declared in the enclosing agent or capability.

A `#modifies data` statement takes the following form:

```
#modifies data DataType reference;
```

Each item in the definition is described in the following table:

Component	Meaning
`#modifies data`	Identifies a data object or beliefset relation that the plan can modify.
`DataType`	Identifies the type of the user-defined data structure or beliefset relation that the plan can access.
`reference`	The name that the data object or relation will be known by in the plan.

Table 7-9: Components in the `#modifies data` declaration

For more information on beliefset relations and how to use them, refer to the *Beliefset Relations* chapter.

#uses agent implementing *Interface reference*

This declaration identifies a particular Java interface that the plan requires when it is executed. Some plans rely on certain methods to be provided by the agent in which they are used. For this reason, JACK normally only allows plans to be used in the agent for which they have been defined.

The `#uses agent implementing` declaration gets around this by allowing the plan's required methods to be described by a Java interface. Any agent that implements this interface must provide implementations for these methods and as a result will be able to adequately support the plan.

Note: `Agent` sub-classes can be defined by extending from other `Agent` classes that have been defined. It is not necessary to extend only from the base `Agent` class.

This declaration takes the following form:

```
#uses agent implementing Interface reference;
```

where *`Interface`* is the name of the Java interface that contains the methods this plan requires, and *`reference`* is the plan's handle on this interface. The *`reference`* handle allows the plan to invoke methods through the interface. The *`reference`* becomes a member of the plan referring to the agent, but only of the given interface type.

When a plan definition includes a `#uses agent implementing` declaration, it can be used in any agent that claims to implement this interface. This claim is made in the agent definition header. For example, a plan that includes the following declaration:

```
#uses agent implementing WorldInterface world;
```

can be included in an agent whose definition is shown below:

```
agent World extends Agent implements WorldInterface
{
   // Agent definition
}
```

#uses interface *Interface reference*

This declares that one of the enclosing capabilities (or the enclosing agent) implements *`Interface`*. Note that this declaration supersedes the `#uses agent implementing` declaration for most purposes.

The declaration takes the following form:

```
#uses interface Interface reference;
```

where *Interface* is the name of the Java interface that contains the methods this plan requires, and *reference* is the plan's handle on this interface. Note that a class can also be considered to be an interface, so that *Interface* can be the actual name of a Capability class or an Agent class.

For example, given an agent:

```
agent HelloWorld extends Agent
{
      #handles event SayHello;
      #uses plan HelloPlan;
           :
       int counter;
           :
       public void hello()
       {
           //Java code
       }
 }
```

the HelloPlan can include the declaration

```
 #uses interface HelloWorld self;
```

This gives the plan access to the HelloWorld members and methods as illustrated in the following code fragment:

```
plan HelloPlan extends Plan
{
   #handles event SayHello sh;

           :
    #uses interface HelloWorld self;

    body()
    {
           :
        System.out.println(self.count);
           :
        self.hello();
     }
 }
```

#reasoning method *name(parameters) <body>*

The plan's body() method is the main method in a plan. It describes all the actions that the agent performs when this plan is executed. A plan's body() method is unlike normal Java methods in that it is a *reasoning method*. However, the body() method is not the only reasoning method that a plan may contain. As with other structured programming languages, it is sometimes easier to split functionality into separate methods. Similarly, it is possible for a plan to include other reasoning methods, which the main body() method can call as needed.

Each reasoning method that a plan contains beyond the `body()` reasoning method must be defined using the `#reasoning method` statement. The syntax for these definitions is shown in the following code:

```
#reasoning method name(parameters)
{
   // body of reasoning method. This can contain a mixture
   // of Java code and JACK Agent Language @-statements
}
```

Each of the components of this definition are explained below.

Component	Meaning
`#reasoning method`	Introduces the definition of a plan's reasoning method.
name	The name used to identify the method. Other reasoning methods in the plan can invoke this method by name.
parameters	A list of the reasoning method's parameters. The syntax and semantics are the same as for parameter lists in normal Java.

Table 7-10: Components of a reasoning method definition

Reasoning methods do not have a return type like normal Java methods. Instead, they either succeed or fail. This is determined by the execution path through the reasoning method. When the agent executes a reasoning method, it treats every statement or block as a separate step that must be completed successfully if the reasoning method itself is to be completed successfully.

Therefore, if the agent succeeds in executing a statement or block, it will proceed to the next statement or block in the reasoning method. However, if it fails, it will assume that the reasoning method can no longer be completed successfully and therefore will fail.

Of course, this does not mean that every single conditional test must succeed for the agent to continue executing the reasoning method. For example, if the reasoning method contains an `if ... then ... else` construct and the `if` condition fails, it can still execute the `else` clause and continue. However, if the `if` condition succeeded and then one of the statements in the `then` block failed, the agent would still abandon the reasoning method as unsuccessful.

If the agent reaches the end of a reasoning method without executing any statements that fail, the reasoning method succeeds. Otherwise, it fails. This is analogous to a human performing a task that requires the completion of several steps. If any step fails, the task can no longer be completed, but if all steps succeeds, the task succeeds.

Reasoning methods can include special JACK Agent Language constructs known as *reasoning method statements*. Reasoning method statements control aspects of how the reasoning

method runs, and execute any agent-specific activities that the plan has to perform (such as sending messages to other agents, posting events, or ensuring that the plan is executed only while a given maintenance condition continues to be true). JACK Agent Language reasoning method statements begin with an `@` character.

Reasoning methods execute as *finite state machines*. Finite state machines define an execution model where each step is atomic and the execution can be paused only between these steps.

`#reasoning method pass()` <*body*>

The `pass()` reasoning method is a special reasoning method that may be included in a plan. Unlike other reasoning methods, it should not be called explicitly by other reasoning methods in the plan. Instead, it is executed after the plan's `body()` method has succeeded, but before this success is reported back to the task in which the plan is being executed. Therefore, it can be used to describe any post-processing or 'cleaning up' that may be required after the plan has succeeded.

The `pass()` reasoning method is defined using the following format:

```
#reasoning method pass ()
{
   // Post-processing and cleanup steps for when the
   // plan has succeeded.
}
```

The `pass()` reasoning method can only be called by the task-execution infrastructure. Furthermore, its success or failure has no effect on the plans outcome. For example, if the plan succeeds but the `pass()` method fails, the plan still succeeds.

The `pass()` reasoning method is executed if and only if the plan succeeds. If the user wants to perform post-processing for a plan that fails, they should use the `fail()` reasoning method described below.

`#reasoning method fail()` <*body*>

This reasoning method complements the `pass()` reasoning method. Like the `pass()` reasoning method, it should not be called from other reasoning methods, but instead must be called by the underlying JACK task execution engine. The `fail()` reasoning method is called if, and only if, the plan it appears in fails. Like the `pass()` reasoning method, it can be used to perform any post-processing or cleanup activities that might be required.

The `fail()` reasoning method is defined using the following format:

```
#reasoning method fail ()
{
   // Post-processing and cleanup steps to be performed
   // if the plan fails.
}
```

7.6 Reasoning Method Statements (@-Statements)

As has been described earlier, reasoning methods are different from normal Java methods in a number of ways. One of these is that they can include a number of special JACK Agent Language statements. These statements are all preceded by an `@` symbol and describe specific logical guards or logical actions that the agent should perform within the reasoning method. The reasoning method statements are:

- `@wait_for(parameters)`
- `@action(parameters) <body>`
- `@maintain(parameters)`
- `@post(parameters)`
- `@reply(parameters)`
- `@send(parameters)`
- `@subtask(parameters)`
- `@sleep(parameters)`
- `@achieve(parameters)`
- `@insist(parameters)`
- `@test(parameters)`
- `@determine(parameters)`
- `@parallel(parameters) <body>`

Each of these reasoning method statements is described below.

@wait_for(*parameters*)

The `@wait_for` expression controls the temporal flow through a reasoning method. When the agent executes a `@wait_for` expression, it causes the plan to wait until a given logical condition becomes true before continuing. That is, the agent waits for the condition to be satisfied and then continues.

The `@wait_for` expression is a Finite State Machine (FSM).

An `@wait_for` expression takes one of the following three forms:

```
@wait_for (Cursor wait_condition);
@wait_for (Cursor wait_condition, Cursor sentinel_condition);
@wait_for (Cursor wait_condition, double timeout);
```

Each component of this statement is described in the following table:

Component	Meaning
`@wait_for`	Introduces a `@wait_for` statement.
`wait_condition`	The wait condition specifies the logical condition that the agent must wait for before continuing the plan. The logical condition can be any JACK Agent Language cursor statement. A cursor is any logical expression whose truth value can change over time. An example is a statement that performs a beliefset query and then tests the result – as the tuples in the agent's beliefset can change while other tasks are performed. Although the query can be satisfied or not at one moment, this doesn't necessarily mean that it will be the next.
`sentinel_condition`	The sentinel condition specifies a guard on the wait – a condition that represents a timeout for the agent's waiting opportunity. If the sentinel condition is satisfied before the wait condition, the agent has run out of waiting time and therefore wait statement has failed. If the wait condition is satisfied before the sentinel condition, however, the `@wait_for` statement succeeds. The sentinel condition is optional. When absent, the agent is free to wait for as long as necessary for the wait condition to be satisfied.
`timeout`	The timeout specifies a guard on the wait – a deadline that marks when the agent can no longer wait until the *`wait_condition`* is true. The timeout is a double precision real number that represents the wait period in seconds. The timeout is checked like a stop-watch from the moment the `@wait_for` condition is first tested. If this time period elapses without the wait condition having become true, the `@wait_for` statement fails. Otherwise, it succeeds. The timeout is optional. When absent, the agent has no waiting deadline and can wait as long as is necessary.

Table 7-11: Components of the `@wait_for` statement

Each form of the `@wait_for` statement, and how it is executed, is described in the following sub-sections.

@wait_for(*wait_condition*)

This is an unguarded wait. The agent will stop executing the current task and wait for as long as is necessary for the *wait_condition* to be satisfied. If the condition is already true the agent continues, unhindered, through the reasoning method. This expression always succeeds – it succeeds when the *wait_condition* is satisfied.

Note: Because this form of the @wait_for statement is unguarded, a poorly formed *wait_condition* could infinitely suspend the execution of the plan.

This will not hamper the progress of the agent because the wait condition must be a triggered cursor. Triggered cursors are not checked using a busy-wait loop. Instead, they are only tested when the agent performs a modification action on one of the cursor's relations.

This expression never fails. If the *wait_condition* is never met, the agent just waits for it indefinitely (or until one of the circumstances described below arises). Note that this does not mean that the agent as a whole comes to a standstill. Since the waiting task is suspended, the agent will be able to proceed with any other tasks that are active, and undertake new tasks as new asynchronous events arrive.

In the JACK runtime environment, satisfying the wait condition is not the only way to 'break' a @wait_for statement. Although satisfying the wait condition is the way a @wait_for statement is meant to be broken, it will also be broken under the following circumstances.

- An exception is raised by the wait condition.

  This may happen if a change is made that triggers the retesting of a cursor expression in the wait condition. If the retest raises an exception, that exception will break the @wait_for statement.

- There is a maintenance condition violation in a parent task.

  This may happen if the agent is executing the @wait_for statement in a plan that is running in a @maintain statement's subtask.

  If the @maintain statement's maintenance condition is violated while the agent is waiting, the entire subtask will be aborted, and the wait will terminate.

@wait_for(*wait_condition, sentinel_condition*)

This form of the wait condition tells the agent to do the following: "wait for *wait_condition* to become true until *sentinel_condition* becomes true". Both the *wait_condition* and the *sentinel_condition* can be any form of triggered cursor.

This expression succeeds when the *wait_condition* is satisfied before the *sentinel_condition*. In fact, it succeeds the moment that *wait_condition* becomes true.

Note: This form of the `@wait_for` expression may still be vulnerable to indefinite waits if neither the *`wait_condition`* nor *`sentinel_condition`* are ever satisfied.

This expression fails when *`sentinel_condition`* is satisfied before *`wait_condition`*. In fact, it fails the moment that *`sentinel_condition`* becomes true.

`@wait_for(`*`wait_condition, timeout`*`)`

This form of `@wait_for` expression tells the agent to do the following: "wait for *`wait_condition`* to become true for *`timeout`* seconds". This is as though the agent is waiting while the stopwatch runs. The timeout period is a *double precision real* number that specifies the timeout period in *seconds*. The granularity of this time period, however, is to the nearest *millisecond*. When the agent executes this statement, the stopwatch starts ticking. If the *`wait_condition`* is satisfied before the timeout is reached, the statement succeeds; otherwise, it fails.

This expressions succeeds when *`wait_condition`* is satisfied before the *`timeout`* period has passed. It succeeds the moment that *`wait_condition`* becomes true. It fails when the *`timeout`* period has passed without *`wait_condition`* having been satisfied. It fails the moment the *`timeout`* period expires.

Unlike other forms of the `@wait_for` expression, this expression will never put the agent in an indefinite wait because the timeout can always expire.

`@action(`*`parameters`*`)` *<body>*

The `@action` statement is a compact way of writing in-line action cursors – these are discussed in the *Cursors* section in this chapter. The `@action` statement has two forms:

```
@action() { <body> } ;
```

and

```
@action(thread_pool) { <body> } ;
```

where *`body`* is the set of statements for the 'action' method of an inline anonymous extension of `aos.jack.util.cursor.Action`, and the optional *`thread_pool`* argument specifies a thread pool to be used instead of the agent's thread pool. If no thread pool is given, the agent's thread pool is used. `Threadpool` is discussed in *Appendix B: Utilities*.

The `@action` statement is translated into a `@wait_for` statement with an action cursor.

@maintain(*logical_condition, event*)

Like the `@subtask` statement, the `@maintain` statement posts an event to be handled synchronously in the same execution thread. The current plan is suspended while the agent handles this event in a separate subtask. However, as well as specifying an event to be handled, the `@maintain` statement also specifies a logical condition which is to be maintained for the **entire** duration of the subtask execution. If the condition is violated during execution of the subtask the statement fails. As one would expect, if the condition is false when the event is posted, the statement will fail immediately.

A `@maintain` statement takes the following form:

```
@maintain (logical_condition, event);
```

Note: The second argument in an `@maintain` statement actually refers to the subtask which arises from the synchronous handling of *`event`*, and is therefore of type `FSM` (refer to the *FSM Statements* section in this chapter for details about FSMs). Consequently, the second argument can be any expression of type `FSM`, such as a reasoning method.

Each component of this statement is explained in the following table:

Component	Meaning
`@maintain`	Identifies the statement as being a `@maintain` statement.
`Cursor` *`logical_condition`*	Specifies a logical condition that must be continually true while the subtask is being performed. The agent waits until any change to the condition's arguments is made, and on this trigger tests the condition to see whether it is still true. The logical condition is a cursor expression.
`Event` *`event`*	The event that is posted by the `@maintain` statement. It can be any `Event` or `BDIGoalEvent`, but will always be handled synchronously by the agent.

Table 7-12: Components of the `@maintain` statement

Note: The instance of the event to be posted will have been created by invoking a posting method on the reference that is declared in the associated `#posts` or `#sends` statement. This was described in the section on posting/sending events in the *Events* chapter.

Termination of the subtask will occur either because:

- the subtask has completed execution without a maintenance violation having occurred. In this situation, the `@maintain` statement will succeed or fail depending on the success or failure of the subtask.
- a maintenance violation has occurred while the subtask is executing. In this case, the `@maintain` statement will fail and the subtask will fail.

Note: The subtask may instigate further asynchronous and synchronous activity. Any asynchronous activity is treated as independent of the original subtask and is not terminated if a maintenance violation occurs. However further synchronous activity is considered to be part of the original subtask and all subtasks in the processing chain at the time of the violation will be terminated.

A maintenance violation is like an exception and will not invoke any fail() methods. If some cleanup action is required when a maintenance violation occurs, it can be achieved by explicitly catching the `MaintenanceViolation` exception within a plan. This is shown below. Note that if this exception is caught, it **must** be rethrown so that it can eventually reach the `@maintain` exception handler. The statement will then fail (as it should).

```
try
{
    // code
}
catch (MaintenanceViolation e)
{
   // cleanup
   throw e;
}
```

@post(*event*)

The `@post` statement is used to post an event from within a plan. The statement accesses the event *posting method* and uses it to post a new instance of the event for the agent to handle.

The event is posted asynchronously to be handled by a new task that is executed in a new and independent task execution thread. Hence this statement always succeeds, because once the event has been posted, the plan continues processing. Even if the event is not handled (because there are no relevant or applicable plans), or the agent attempts to handle it but fails, the `@post` statement still succeeds.

A `@post` statement takes the following form:

```
@post (event);
```

Each component of this statement is described in the following table:

Component	Meaning
`@post` `Event` *`event`*	Identifies this as a **`@post`** statement. The event to be posted. This event must have: • been defined as described in the section *Event Definition*; • been included in this plan's set of `#posts event` declarations; and • been included in the `#handles event` declarations of any agent that uses this plan (i.e. that claims to use this plan with a `#uses plan` declaration).

Table 7-13: Components of the `@post` statement

The plan must use one of the event's posting methods to create the event to be posted. This was described in the section on posting/sending events in the *Events* chapter.

Provided all these conditions are met, a plan can post an instance of an event using the `@post` statement. The event instance is created using the parameters passed to the event's posting method.

@reply(*original_event, reply_event*)

The `@reply` statement is used by an agent to reply to a message event that it has received from another agent. It replies to the sending agent with a message event (`MessageEvent` or `BDIMessageEvent`) which arrives as a data object on the reply queue of the original message event in the sending agent. This means the message event that is sent back using `@reply` does not trigger a new task or plan.

Messaging is normally initiated with `@send` and thereafter `@send` or `@reply` are used according to protocol.

A `@reply` statement takes the following form:

```
@reply (original_event, reply_event);
```

Each component of this statement is described in the following table:

Component	Meaning
`@reply`	Identifies the statement as a `@reply` statement.
`original_event`	The `MessageEvent` that is being replied to.
`reply_event`	The `MessageEvent` that is being sent as a reply.

Table 7-14: Components of the `@reply` statement

An example involving the updating of a GUI is presented below:

```
// Note the 2 methods from MessageEvent that are used
// in the example:
//
//   replied()  which is a boolean cursor method that
//   becomes true when there is an unread reply to the event
//
//   getReply() which returns the first unread reply to
//   the event

// The sender's plan (GUI.plan)
plan GUI extends Plan
{
  #handles event CommandString cse;
  #sends event UserRequest ure;
  // note the absence of a #handles Response declaration

         ...

  body()
  {
         ...

     // the user has entered a command (available in cse) -
     // send it to the command handler agent for processing.

     UserRequest ur = ure.postingMethod(...);
     @send("CommandHandler",ur);
     @wait_for(ur.replied());
     Response r = (Response) ur.getReply();

    // now update the GUI ...
  }
}

// The receiver's plan (ProcessCommand.plan)
plan ProcessCommand extends Plan
{
  #handles event UserRequest ure;
  #sends event Response re;

  body()
  {
    // process command

    // send response
    @reply(ure,re.response(...));

         ...
  }
}
```

@send(*agent_name, message_event*)

The `@send` statement is used to send a message event to another agent from within a reasoning method. Like the `@post` statement, the `@send` statement uses one of the message event's own posting methods to create the instance of the event to be sent. This was described in the section on posting/sending events in the previous chapter.

A `@send` statement takes the following form:

```
@send (agent_name, message_event);
```

Each component of this statement is described in the following table:

Component	Meaning
`@send`	Identifies the statement as a `@send` statement.
`String agent_name`	The name of the agent to send this message event to. The agent's name is specified as a string. Routing of the event to the agent will be handled by JACK's underlying network layer.
`Event message_event`	The message event to send. This message event must have: • been defined as described in the *Events* section; • been included in this plan's set of `#sends event` declarations; • been included in the `#handles event` declarations of any agent that uses this plan (i.e. that contains a `#uses plan` declaration for this plan).

Table 7-15: Components of the `@send` statement

To be able to send an event, the plan must know the name of the agent to send it to. Like the `@post` statement, the `@send` statement always succeeds. This is because the message event is sent asynchronously and once the agent has sent it off, it immediately continues executing the plan. How the message event is handled by the destination agent is not a factor.

`@subtask(event)`

The `@subtask` statement is similar to the `@post` statement, except that instead of posting an event in the normal (asynchronous) way, it posts the event *synchronously*. Instead of being handled in a separate task and execution thread, an event posted with the `@subtask` statement is handled as a *subtask* of the current task.

The `@subtask` statement does not create a new execution thread within the agent as is the case with normal event posting. Instead, it suspends the current plan and adopts a task to handle the event within the same execution thread. The `@subtask` statement then succeeds or fails depending on whether the posted event succeeds or fails.

A subtask is handled exactly like a normal task. That is, the agent posts the event, finds all relevant plans and all applicable plan instances, then executes these plan instances until one of them succeeds or it runs out of alternatives. The only difference is that the suspended plan waits for this result to be returned, whereupon the `@subtask` either succeeds or fails.

A `@subtask` statement takes the following form:

```
@subtask (event);
```

Each component of this statement is described in the following table:

Component	Meaning
`@subtask`	Identifies this as a `@subtask` statement.
`Event` *`event`*	The name of the event to executed in the subtask. This event must have: • been defined as described in the *Events* section; • been included in this plan's set of `#posts event` declarations; and • been included in the `#handles event` declarations of any agent that uses this plan (i.e. that claims to use this plan with a `#uses plan` declaration).

Table 7-16: Components of the `@subtask` statement

Note that the instance of the event to be executed as a subtask will have been created by invoking a posting method on the reference that is declared in the associated `#posts` or `#sends` statement. This was described in the section on posting/sending events in the previous chapter.

The `@subtask` statement can be used to make plan code more portable and manageable. It allows agents to re-use entire plans (or even plan sets) by calling them from within other plans. The other plan can post the plan's invocation event as a `@subtask`, and take further action based on whether this subtask succeeds or fails. Because the subtask is handled in the same task execution thread, the `@subtask` statement gives JACK the means for functional abstraction in plans.

A `@subtask` statement succeeds when the event that it posts is handled successfully in the subtask. If the agent can execute a plan that handles this event successfully, the subtask will be successful. The statement fails when the event that it posts is not handled successfully by the subtask. This may be because no relevant plans are found, no applicable plan instances are found, or all applicable plan instances that are found fail.

@sleep (*timeout*)

The `@sleep` statement, like the `@wait_for` statement, is used to control the temporal flow of a plan. The `@sleep` statement is far more straightforward, however: it merely tells the agent to wait for a designated period before continuing.

A `@sleep` statement takes the following form:

```
@sleep (double timeout);
```

The *`timeout`* is specified as a double and represents the period of time that the agent must wait (sleep) before continuing with the plan. The *`timeout`* period is specified in ticks on the agent's clock. The actual length of time depends on the `Timer` that the agent is using.

If the timer is a real-time clock, the timeout period represents a sleep period in *seconds*, with *millisecond* granularity. If the timer is a dilated or simulation clock, it represents the number of clock ticks on that timer.

For example, the `@sleep` statement;

```
@sleep (6.789);
```

means that the enclosing plan is suspended for 6789 ticks on the `Timer` assigned to the timer member of the calling agent. For a real-time clock this is 6789 ticks at one millisecond per tick, or 6 seconds, 789 milliseconds.

Note: `@sleep` only causes the current task to sleep. Any other tasks that the agent is currently executing proceed as normal.

A `@sleep` statement succeeds when the time period has expired. In other words, the `@sleep` statement always succeeds – it never fails.

@achieve(*condition, goal_event*)

The `@achieve` statement is one of four posting statements that can be used on *goal events only.* The other three are `@test`, `@insist` and `@determine`, all of which test a logical condition and, depending on its result, post a goal event in a different way.

The `@achieve` statement is used to model the situation where an agent is asked to *achieve a goal*. The agent is given a *goal* to achieve, and a *condition* to determine whether any action needs to be taken (i.e. whether the goal has already been achieved).

When the agent executes an `@achieve` statement, it tests the condition.

- If the condition is *true*, the agent believes that the goal has already been achieved and will do nothing. The `@achieve` statement *succeeds* and the agent goes on with the next statement in the plan.

- If the condition is *false*, the agent believes that it must do something to achieve the goal. It suspends the current plan and starts executing a subtask to achieve it. The goal is supplied as a goal event, and achieving it amounts to handling the goal event in a subtask execution.

Because subtasks are handled synchronously, the parent plan waits until the subtask has either succeeded or failed before continuing. The success or failure of the subtask determines the success or failure of the `@achieve` statement.

An `@achieve` statement takes the following form:

```
@achieve (condition, goal_event);
```

Each component is explained in the following table:

Component	Meaning
`@achieve`	Introduces an `@achieve statement`, which asks the agent to test a condition and if it is not true, to handle a goal event.
`Cursor` *`condition`*	The logical condition that the agent tests before determining whether anything needs to be achieved. If this condition is true, the agent assumes that the goal has already been achieved and does nothing. Otherwise, it handles the goal event in a subtask execution.
`BDIGoalEvent` *`goal_event`*	The goal event describing the goal that the agent must try to achieve.

Table 7-17: Components of the `@achieve` statement

Note that the instance of the event to be posted will have been created by invoking a posting method on the reference that is declared in the associated `#posts` or `#sends` statement. This was described in the section on posting/sending events in the previous chapter.

An `@achieve` statement succeeds when either the *`condition`* is true, or when the *`condition`* is false but the subtask to handle the *`goal_event`* succeeds. It fails when the *`condition`* is false and the subtask to handle *`goal_event`* fails.

`@insist(`*`condition, goal_event`*`)`

The `@insist` statement is similar to the `@achieve` statement, but places a greater emphasis on ensuring that the goal is handled properly. With the `@achieve` statement, the agent assumes it has handled the goal if it successfully performs the subtask. No check is made of the success condition to ensure that this is the case: it is assumed.

This assumption is reasonable and it is a general assumption made for event handling in JACK. In fact, the successful handling of an event is *defined* as the location and successful execution of an applicable plan. All task (and subtask) executions succeed as soon as an applicable plan is completed. However, the `@insist` statement allows the agent to add an extra step – to 'make sure' that the goal has been achieved.

It does this by *re-testing the given condition* after each successful subtask execution. If the condition is true, the `@insist` statement succeeds and the agent will continue through the plan. If it fails, however, the agent will attempt to *re-execute the subtask* (i.e. try to achieve the goal again). If the subtask fails, the `@insist` statement fails. Otherwise, the agent repeats the process by checking the condition again.

Hence, an `@insist` statement may involve multiple subtask executions. Each time the same goal event is posted. That is, the agent is insisting that the goal has been achieved as defined by the success condition given in the `@insist` statement. If this goal has not been achieved, the `@insist` statement fails.

An `@insist` statement takes the following form:

```
@insist (condition, goal_event);
```

Each component is explained in the following table:

Component	Meaning
`@insist`	Introduces an `@insist` statement, which repeatedly tries to achieve a goal until the given success condition holds true.
`Cursor` *`condition`*	The logical condition that the agent uses to define the goal's successful achievement. It is tested before each subtask execution. If true, the goal has been achieved and no more work needs to be done. Otherwise, the agent must try to handle the goal event once more.
`BDIGoalEvent` *`goal_event`*	The goal event describing the goal that the agent must achieve.

Table 7-18: Components of the `@insist` statement

Note that the instance of the event to be posted will have been created by invoking a posting method on the reference that is declared in the associated `#posts` or `#sends` statement. This was described in the section on posting/sending events in the previous chapter.

An `@insist` statement succeeds when the *`condition`* is true. This can be when the `@insist` condition is first executed (i.e. before any subtasks have been performed) or after any successful subtask execution of the *`goal_event`*. It fails when any subtask execution of the *`goal_event`* fails.

`@test(`*`test_condition, goal_event`*`)`

The `@test` statement tells the agent to determine whether a logical condition (*`test_condition`*) is true or false. It is similar to the `@achieve` statement, except that in this case the logical condition is important.

If the *`test_condition`* is true, the `@test` statement succeeds and it performs a unification against the *`test_condition`*.

If the *`test_condition`* is false, the `@test` statement fails and there is no unification.

The goal event comes in to play when the logical condition is neither true nor false, but rather is *unknown*. Not all logical conditions can be unknown in JACK Agent Language – only those that follow Open World semantics. Therefore, if a logical expression consisting of ordinary boolean values and Closed World relations is tested, the `@test` statement will always succeed or fail based on the logical expression's truth valuation and the `@test` statement's goal event will never be posted. Thus, it is only when Open World relations are involved that the possibility of goal handling may arise.

When an Open World relation is involved and the *`test_condition`* is unknown, the *`goal_event`* will be posted. The event is then responsible for doing any unification that is required, as the `@test` statement will do nothing when the goal has finished executing – it will simply succeed or fail based on the success or failure of the goal. The `@test` statement does not check whether or not the *`test_condition`* is true if the goal succeeds.

A `@test` statement takes the following form:

```
@test (test_condition, goal_event);
```

Each component is explained in the following table:

Component	Meaning
`@test`	Introduces a `@test` statement, which tests a condition and succeeds or fails accordingly. If it cannot evaluate the condition, it executes a subtask to try and find the answer.
`Cursor` *`test_condition`*	The logical condition that the agent is being asked to test. If the agent can determine whether it is true or false from its own data and beliefs, no further work needs to be done. Otherwise, the agent must adopt a goal subtask to find the truth valuation.
`BDIGoalEvent` *`goal_event`*	A goal event describing the goal to find out the condition's truth. It will only be executed if the agent cannot determine the condition's truth from its own set of beliefs.

Table 7-19: Components of the `@test` statement

Note that the instance of the *`goal_event`* to be posted will have been created by invoking a posting method on the reference that is declared in the associated `#posts` or `#sends` statement. This was described in the section on posting/sending events in the previous chapter.

The `@test` statement can be thought of as modelling what a human would do when asked to test something. First, they would consult their own knowledge (beliefs) to find the answer. If this uncovers nothing, action would be taken to find out the answer. The `@test` works in the same way – the agent first tests the logical condition (often against its own beliefset). If this does not work, it starts trying to achieve a goal to find the answer.

An `@test` statement succeeds when the *`test_condition`* is true, or the *`test_condition`* is unknown but the goal to find its truth (*`goal_event`*) succeeds (i.e. the subtask that handles this goal succeeds). It fails when the *`test_condition`* is false, or the *`test_condition`* is false and the goal to find out its truth (*`goal_event`*) fails.

`@determine(`*`binding_condition, goal_event`*`)`

The `@determine` statement tells the agent to try and find a logical binding for which a given `BDIGoalEvent` will succeed. It does this by finding all possible sets of values that satisfy the logical condition, then posting the goal event for each in turn until one of them succeeds. When the goal event succeeds for a particular set of bindings, the `@determine` statement succeeds and the values for these bindings are committed. The `@determine` statement will return (bind) the logical members in the logical condition with those values that caused the goal event to be satisfied.

In a way, it is the opposite of the other three `BDIGoalEvent` posting statements: `@achieve`, `@insist` and `@test`. Instead of testing a logical condition and posting a goal event when it fails or is unknown, the `@determine` statement posts a goal event for known values. The agent is not trying to add to its knowledge base. Rather, it is trying to determine which of the possible candidates in its knowledge base allow it to achieve the goal that it wishes to achieve.

The `@determine` statement's logical condition is designed for use with a beliefset cursor. It iterates through the results that the beliefset cursor returns, testing these returned values, until one set of values satisfies the logical condition. In other words, the agent is being asked to "determine a situation under which the given goal can be achieved". The `@determine` statement iterates over the beliefset cursor, testing and discarding possible bindings for its logical members, until all set have been found that satisfy the condition. Once this set of bindings has been found, the agent posts a `BDIGoalEvent`, synchronously, for each binding in turn. This causes a subtask to be executed for each binding, one after the other, until one succeeds.

If a subtask succeeds, the `@determine` statement succeeds and no more bindings are tried. It commits the current set of bindings and discards the rest. Otherwise, the agent takes the next set of bindings that satisfied the logical condition and attempts the goal again. Therefore, the `@determine` statement succeeds if, and only if, the goal can be met for at least one of the sets of bindings returned by the logical condition.

A `@determine` statement takes the following form:

```
@determine (binding_condition, goal_event);
```

Each component is explained in the following table:

Component	Meaning
`@determine`	Introduces a `@determine` statement, which iterates through all possible values that satisfy a logical condition until a goal subtask using these values succeeds.
`Cursor` *`binding_condition`*	The logical condition that the agent uses to find values. For each set of bindings that satisfy this condition, the agent posts a goal event.
`BDIGoalEvent` *`goal_event`*	The goal event that the agent executes for each set of values that satisfy the binding condition. When this goal event succeeds, the `@determine` statement succeeds.

Table 7-20: Components of the `@determine` statement

Note that the instance of the event to be posted will have been created by invoking a posting method on the reference that is declared in the associated `#posts` or `#sends` statement. This was described in the section on posting/sending events in the previous chapter.

The `@determine` statement has been designed specifically to work with beliefset relations in the *`binding_condition`*, iterating over the relation's tuples until it finds ones that might meet the goal. However, it determines all possible bindings before the first goal event is posted. This means that if the subtask changes any tuples that are used in the logical condition while they are processing, this will not affect the `@determine` statement. If the subtask goes on and fails, it will post next goal with the next set of bindings from its list. It will not re-evaluate the logical condition and find that the set of tuples have changed.

An `@determine` statement succeeds when the *`goal_event`* is successfully handled for a set of bindings provided by *`binding_condition`*. The statement succeeds as soon as the first binding is successfully handled – no more bindings are tested. It fails when no bindings are found that satisfy *`binding_condition`*, or the handling of *`goal_event`* fails for all bindings found.

`@parallel(`*`parameters`***`)`** *<body>*

The `@parallel` statement allows concurrent sub-tasking of a set of statements within reasoning methods. The `@parallel` statement suspends execution of the calling plan while all enclosed statements are executed in parallel.

The `@parallel` statement is used in a plan as a program control structure to sub-task goals in parallel, or more precisely, to progress on several branches of activity in the plan in parallel. The success or failure of the statement depends on the successes and failures of the parallel branches involved. The programmer specifies whether all branches need to succeed or whether it is sufficient that at least one branch succeed. The programmer also specifies whether to wait for all branches to complete before the statement completes, or whether to complete the statement as soon as possible (e.g. with the first successful branch, if the success of one branch is sufficient).

The `@parallel` statement provides a very powerful mechanism for expressing plans. The implied task synchronisation reduces the effort of programming coordinated activity, in particular while focusing on the "success paths". Recovery procedures, contingency planning and their effect on coordination, require careful design and use of the task control statements available in JACK.

The form of the `@parallel` statement is as follows:

```
@parallel( arguments ) {
    branch_1;
    branch_2;
     :
     :
    branch_n;
};
```

The `@parallel` statement works like a control structure where the statements, `branch_1`, `branch_2`, etc. are executed as parallel tasks, while the `@parallel` statement itself waits until its termination condition holds. A branch is either a single statement or a compound statement. Branches may be labelled. They have the same form as a Java labelled statement.

The mandatory arguments to the `@parallel` statement specify success condition, termination condition, and how termination is notified. An optional fourth argument is allowed, which is an object through which the execution of the parallel statement can be monitored.

- The **success condition attribute** specifies when the `@parallel` statement as a whole succeeds. The variants are:
  - `ParallelFSM.ALL`. The `@parallel` statement succeeds when all parallel branches have terminated successfully. This can be viewed as a parallel AND statement, because all branches must succeed. Further, the `@parallel` statement fails immediately with the first branch failure, and all ongoing branches are then notified accordingly.
  - `ParallelFSM.LAST`. The `@parallel` statement succeeds when all the parallel branches have terminated successfully. However, failure is postponed until all branches have terminated.
  - `ParallelFSM.FIRST`. The `@parallel` statement succeeds as soon as any one of the parallel branches succeeds, and all ongoing branches are then notified accordingly. This can be viewed as a parallel OR statement with (temporal) short circuiting.
  - `ParallelFSM.ANY`. The `@parallel` statement succeeds if one of the branches succeed, but does not terminate until all branches have terminated.
- The **termination condition attribute** is a triggered condition which will terminate the `@parallel` statement if it becomes true. Ongoing parallel branches are then notified and treated as failed, and the `@parallel` statement fails.

  The termination condition can be any triggered expression in JACK and may in particular be affected by some of the branches. For instance, a branch may make a change to a team belief that could trigger the condition that terminates the `@parallel` statement.

  Note that if `false` is used in place of a termination condition, then this abort mechanism is effectively turned off.
- The **notification exception attribute** provides programming control of how branches are notified about termination. The `notification` exception is a user defined Java exception object. This exception is thrown to active branches that are executing in parallel if they are required to terminate (i.e. if the termination condition is encountered). The termination takes effect immediately for the `@parallel` statement. The branches are thereafter notified by using the given exception. If there is no termination condition, then the value of null is used. If there is a termination condition, then the exception object must be provided.

- The **optional monitor attribute** must, if given, be an instance of class `ParallelMonitor`. Through this object, the parallel execution can be monitored and controlled, as illustrated by the following code outline:

```
ParallelMonitor p = new ParallelMonitor();
@parallel(..., p) {
    reasoning_method1(....);
    xxx: reasoning_method2(....);
    {
        @wait_for(elapsed(1000));
        p.throwTo("xxx", new Exception("Check point"));
    }
};
```

With a `ParallelMonitor` object, the teamplan can inspect the processing of parallel branches, and as in the example, throw exceptions to branches selected by label or by index.

The `ParallelMonitor` object given to a `@parallel` statement becomes a handle for that statement, which can be probed regarding termination and/or success of the individual branches. It can also be used to add branches to a `@parallel` statement dynamically through the `addTask(FSM)` method. The `FSM` argument is an event or a reasoning method in the plan. The following code extract is an illustration of how this may be used.

```
ParallelMonitor p = new ParallelMonitor();
@parallel( ... , p ) {
    // some definite parallel branches
    ...

    // This branch generates more parallel branches
    for (int i=0; i < 42; i++)
            p.addTask( cleverness(i) );
}

#reasoning method cleverness(int i) { ... }
```

The code above would create 42 instances of the `cleverness(int)` reasoning method, with different input argument, and add their executions to the parallel statement monitored by `p`. The `@parallel` statement will then not complete until all the branches have completed.

In addition, parallel branches can be referred to via *labels*. If a branch is a labelled statement, the program can refer to that statement using the label as a string. The earlier code outline is an illustration of this, where the branch labelled `xxx` is thrown an exception after 1000 seconds.

Branches can also be referred to by index, where 0 is the first branch, 1 the second branch, etc. Dynamically added branches are numbered contiguously after the definite branches, and the `addTask(FSM)` method also returns the index for the branch added. The `ParallelMonitor` class is documented in section on the *ParallelMonitor Base Class*.

Exception Handling within the Parallel Execution Model

Exception handling within the parallel execution model is designed to strictly follow the Java model for exceptions. That is, a branch may throw an exception, and if not caught, the exception is propagated upwards in the calling stack. When the exception reaches the `@parallel` statement, it causes a notification to any ongoing branch before the exception propagates out of the `@parallel` statement.

A branch may catch exceptions, as in the following example:

```
@parallel(...) {
    try { reasoning_method1(...) }
    catch (...) {...}
    finally {...}

    try { @test(...) }
    catch (...) {...}
    finally {...}
};
```

In this example, there are two parallel branches that each contain a `try-catch-finally` block. If, for instance, an exception is thrown within the reasoning_method1, the parallel sub-statement may catch that exception and succeed anyhow without propagating the exception.

The `ParallelMonitor` Class

A `ParallelMonitor` object enables the tasks associated with a `@parallel` statement to be explicitly monitored and controlled. The `aos.extension.parallel.ParallelMonitor` class implements the following interface:

```
public int addTask(FSM)
//
// Adds a new branch to the @parallel statement. FSM
// is either a reasoning method or an event. It returns the
// index to the new branch. The first branch in the @parallel
// statement has an index of 0.
//

public Cursor finished()
//
// A triggered Cursor for checking that the @parallel
// statement has finished.
//

public Cursor changed()
//
// A triggered Cursor for reacting to state changes in the execution
// of the @parallel statement, e.g. when branches finish.
//
```

```
public boolean hasFinished()
//
// Tests whether the @parallel statement has finished or not.
//

public int getStatus()
//
// Returns the current execution status of the @parallel
// statement.
//

public int nTasks()
//
// Returns the total number of parallel branches.
//

public int getStatus(String n)
//
// Returns the execution status of a labelled branch.
//

public int getStatus(int n)
//
// Returns the execution status of a branch by index.
//

public Throwable getException(String n)
//
// Returns the exception, if any, thrown to a labelled branch.
//

public Throwable getException(int n)
//
// Returns the exception, if any, thrown to a branch
// identified by index.
//

public int findTaskIndex(String name)
//
// Returns the index for a labelled branch.
//

public void throwTo(String name, Throwable t)
//
// Throws an exception to a labelled branch.
//

public void throwTo(int n, Throwable t)
//
// Throws an exception to a branch identified by index.
//
```

7.7 Cursors

In relational databases, a query can return multiple tuples in the form of a result set; access to the elements of this set is then provided through a *cursor*. In JACK, these concepts have been extended to provide cursors which not only operate in the conventional manner but can also operate on the temporal evolution of a query. The latter type of cursor is typically used in

JACK applications to determine when a particular condition (such as the clock reaching a specified time) becomes true. Cursors which provide this additional capability within JACK are implemented as *triggered cursors* . Triggered cursors are not checked using a busy-wait loop – rather, they are only tested when the agent performs a modification action on one of the cursor's relations.

All JACK cursors have a `next()` method which provides access to the next element in the result set. As one would expect, the exact behaviour of the `next()` method is cursor dependent. A brief description of each cursor and the behaviour of its `next()` method is given below:

Time cursors – test a given value against an internal clock and return true or false depending on whether the time period has elapsed. Call to `next()` check to see if the clock has reached the specified time.

Again cursors – return true at regular time intervals after invocation. In between these times, they return false. The first call to `next()` (and the first call to `next()` after true is returned) causes the next trigger point to be set. All other calls to `next()` check to see if the trigger point has been reached.

Change cursors – monitor an `Observable` object and return true when the object has changed its state and satisfies a particular condition. Otherwise, false is returned. The first call to `next()` (which can occur immediately after construction if the appropriate constructor was invoked) returns true or false depending on the cursors `condition()` method. `next()`is then called whenever the object being observed notifies that a change has occurred and the cursor is tested using the `condition()` method.

Action cursors – are used to initiate long-running Java methods from plans, and to test for their completion. The first call to `next()` starts the action method; subsequent calls check the completion status.

RepeatAction cursors – are used to repeatedly execute an action. The first call to `next()` starts the action method; subsequent calls check the completion status. The call after true is returned causes the action to be repeated.

Beliefset cursors – query a beliefset relation, attempting to find a tuple that matches the given pattern. When first called, `next()` unifies the cursor's logical member(s) to the first values that satisfy the cursor query. On subsequent calls it rolls back the last unification of the cursor's logical members and then attempts to re-bind them.

Enumeration cursors – iterate over a `java.util.Enumeration`. True is returned while elements are available in the enumeration; false is returned when the end of the enumeration is reached. Successive calls to `next()` unify the cursor's logical member to successive elements in the enumeration.

Array cursors – are non-triggered cursors which can be used to bind logical variables to the values of an array. Successive calls to `next()` provide one binding at a time until the array is exhausted.

In all cases, the cursor may return a different truth value each time it is tested.

In situations where the cursor can result in the binding of logical variables (as in beliefset and enumeration cursors) the current binding is made available by invoking `next()` on the cursor. Within `#reasoning` methods the `.next()` is implicit when a condition or sub-condition of type `Cursor` occurs as either:

- a top level condition statement, or
- the condition part of a composite statement.

Composite statements which have condition parts are:

- the Java `if`, `while`, `do` and `for` statements; and
- the JACK `@achieve`, `@insist`, `@test`, `@determine`, `@wait_for` and `@maintain` statements.

Note that `.next()` is **not** implicit in `Cursor` valued right-hand side expressions of assignment statements.

Each of the above cursor types is described in more detail in the following sub-sections.

7.7.1 Time Cursors and Again Cursors

Time cursors are used for simulation purposes, and provide internal timing facilities for simulation measurement and synchronisation. JACK provides both real-time cursors as well as dilated clock cursors. Time cursors can use a *real-time clock*, which synchronises the agent with real-time systems; a *dilated clock* for simulated time increments that can be altered as required (effectively providing slow-motion, fast-forward, incremental steps, etc.); or a *simulation clock*, which can be ticked manually to provide even more control over the passage of time in a simulation environment. Each kind of clock is packaged in a `Timer` class.

A time cursor has the following constructor:

```
TimeCursor( long time, Timer clock )
```

The parameters are described below:

Parameter	Meaning
time	The time to be tested against the designated `Timer` clock.
clock	The clock that the time given above is tested against.

Table 7-21: The parameters in the `TimeCursor` constructor

A time cursor becomes true when *time* has been reached on the *clock*, i.e. if the *clock* time is at or past the designated *time*. It becomes false when *time* has not yet been reached on the *clock*. Hence, the time cursor is like a stopwatch for the agent. It is false as long as there is still time to go (the designated time has not been reached) and then true afterwards. Time cursors are useful for causing the agent to synchronise with other agents or objects in a simulation program.

Like time cursors, again cursors are used for simulation purposes, and provide internal timing facilities for simulation measurement and synchronisation. As with time cursors, again cursors can use a range of different clocks.

An again cursor has the following constructor:

```
Again( long interval, Timer clock )
```

Each parameter is described below:

Parameter	Meaning
interval	The interval between successive triggerings of the cursor.
clock	The clock that is used for determining whether or not triggering should occur.

Table 7-22: The parameters in the `Again` cursor constructor

An again cursor becomes true when the next *interval* has been reached on the *clock*. It becomes false when the next *interval* has not yet been reached on the *clock*. Again cursors are useful for causing the agent to synchronise with other agents or objects in a simulation program. For an example which uses an again cursor, refer to *Example 2* in the *Views* chapter.

With both time and again cursors, any clock of the JACK `Timer` class can be used in a cursor statement. In JACK, the following timer members are provided for agents to use:

- `aos.util.timer.RTClock.timer`
- `aos.jack.jak.core.Jak.timer`
- `aos.jack.jak.agent.Agent.timer`

These timer members are assigned a particular clock, which describes how time is manipulated. These clocks are listed below.

- `aos.util.timer.RTClock`
- `aos.jack.jak.util.timer.DilatedClock`
- `aos.jack.jak.util.timer.SimClock`

Each `Timer` member and `Clock` class is described in the following sub-sections.

aos.util.timer.RTClock.timer

This `Timer` member is the JACK *universal real-time clock*. Since it is `static`, there is only a single copy of it which is available to all agents. It is initialised from the operating system's real-time clock, and measures the current system time when tested.

This `Timer` ticks in milliseconds. Hence, when testing it with a long integer, this integer should be a measurement of system time expressed in a unit of milliseconds.

This `Timer` provides a base method `getTime()`. When called, this method will return the clock's current time. Because this timer measures system time, the `getTime()` method will return the same result as the normal Java `System.currentTimeMillis()` system method.

aos.jack.jak.core.Jak.timer

This `Timer` member is the JACK *relative real-time clock*. It is initialised from the universal real-time clock, but can be reset by any agent in the application as required to start measuring time from a specified moment. For example, in a soccer playing multi-agent system, you might use a relative real-time clock to measure match time. It would be reset at kickoff, and again at half time.

Like the universal real-time clock, this `Timer` is `static` and hence common to all agents and objects throughout the application.

aos.jack.jak.agent.Agent.timer

This is a *private real-time clock* for a specific agent. It is initialised from the relative real-time clock, but can be reset if required.

aos.util.timer.RTClock

This is the JACK *real-time clock* class. It measures time in milliseconds, is initialised from the operating system's clock and supplies the `getTime()` method for reading its current value.

aos.jack.jak.util.timer.DilatedClock

This is the JACK *dilated clock* class. When a `Timer` is defined to be of this class, it behaves like a dilated clock. A dilated clock is like a console on a video recorder: it can allow time to pass at normal pace, but it can also slow down, speed up or pause time if required. It is called a dilated clock because slowing down and speeding up the clock can be thought of as redefining the length of each tick.

The `DilatedClock` class provides two constructor methods:

```
DilatedClock( double dilation, Timer t )
DilatedClock( String name, double dilation, Timer t )
```

Each parameter is explained in the following table:

Parameter	Meaning
dilation	Specifies the dilation factor (or time between ticks). A dilation factor of: • `1` is real-time, • `> 1` is fast motion, • `< 1` is slow motion, and • `0` is stopped time. The dilation factor can be thought of as a multiplier. For example, when set at 2 the clock will run twice as fast as a real-time clock.
t	This is the `Timer` that will be used to specify this clock. The timer will determine whether the clock is the universal clock, a relative real time clock or a private clock for use by the agent only.
name	This parameter allows a name to be provided for the clock instance that is constructed. When absent, the clock is given a default name.

Table 7-23: The parameters in the `DilatedClock` constructor

The JACK `dilated clock` class also provides a base method to change the dilation factor, as well as a base method to read the current time. These methods are listed below:

`getTime()` – returns the clock's current time.

`setDilation (double` *`dilation`*`)` – sets the clock's dilation factor to the new value specified. This allows agents to manipulate the clock as they run.

`aos.jack.jak.util.timer.SimClock`

This is the JACK *simulation clock* class. This clock has been provided for specific simulation purposes where even greater time manipulation is required than is provided with the JACK *dilated clock* class. Unlike the dilated clock, the simulation clock is ticked manually.

The `SimClock` class provides two constructors:

```
SimClock()
SimClock( String name )
```

The second constructor allows you to specify a name for the newly created clock instance, while the first allocates the new clock a default name.

As well as `getTime()` to read the clock's current time value, the simulation clock provides two methods for the agent to manage the passage of time. These methods are listed below.

`setTime (long t)` – used to reset the time to a specific value.

`adjustTime (long delta)` – used to add delta ticks to the clock.

Hence, the agent is not constrained to increment the clock by a single tick: any number of ticks can be applied at once.

7.7.2 Change Cursors

Change cursors are used to observe `java.util.Observable` objects. A change cursor is intended to be used in an `@wait_for` statement to block condition testing until the cursor returns true. This occurs when the object being observed notifies that a change has occurred and the cursor's `condition()` method evaluates to true. Programmers can develop their own customised change cursors by extending `aos.jack.util.cursor.Change`.

A change cursor has two constructors:

```
Change( java.util.Observable observable, boolean flag )
```

and

```
Change( java.util.Observable observable )
```

The parameters are described in the following table:

Parameter	Meaning
observable	The object to be observed.
flag	Specifies whether the cursor condition will be tested on the first access to the cursor. If the flag is set to true, the value of the condition will be returned; if it is false, then false will be returned. Constructing the cursor with a false flag in a `@wait_for` statement will cause the `@wait_for` statement to wait until the condition is triggered even if the condition is initially true.

Table 7-24: The parameters in the `Change` cursor constructor

Note: For efficiency reasons JACK uses `aos.util.Watchable` internally (rather than `java.util.Observable`) as the base class for its observables – additional constructors are therefore available which accept an observable of type `Watchable`

A change cursor becomes true when the object being observed notifies that a change has occurred and the cursor's `condition()` method returns true. The default implementation of `condition` is to always return true – if this behaviour is inappropriate for a given application, the programmer must extend the `Change` class and override the `condition()` method. It may also become true if testing of the cursor immediately after construction has been enabled.

The following two examples illustrate the use of the change cursor:

Example 1: Monitoring data in a Java class

In this example, the agent is required to monitor the progress of a chemical reaction and determine when the reaction is complete. It is assumed that the reaction variables are sampled on a regular basis, and that the current reading is available in an instance of the `Reading` class (which `extends Observable`). The `ReactionCursor` class overrides the `condition()` method in the base class (`aos.jack.util.cursor.Change`), incorporating the code to test whether or not the reaction is complete. The monitoring plan could then be similar to the following:

```
public plan MonitorReaction extends Plan
{
    ...

    body()
    {
        r = new Reading();

        //code to start the reaction
            :
            :
        // now wait for completion - sampling is external
        // to this plan
        @wait_for(new ReactionCursor(r,false,7.0));
    }
}
```

The `ReactionCursor` class and `Reading` class would be similar to the following:

```
public class ReactionCursor extends Change
{
    Reading reading;
    double endpH;

    public ReactionCursor(Reading r,boolean test,double pH)
    {
        super(r,test);
        reading = r;
        endpH = pH;
    }

    public boolean condition()
    {
        if (reading.pH() < endpH)
            return true;
        return false;
    }
}
```

```
import java.util.Observable;
public class Reading extends Observable
{
    public long time;
    public double pH;

    public Reading()
    {
    }

    public void log(long p1,double p2)
    {
        if (Math.abs(pH-p2) > 0.05)
        {
            time = p1;
            pH = p2;
            setChanged();
            notifyObservers();
        }
    }
        ...
}
```

Example 2: Monitoring data in a beliefset

Since beliefsets extend `aos.util.Watchable` they can be used as the observable in a change cursor. In the following example, `jigs` is a beliefset which maintains the current status of the jigs in an assembly cell. The `AllocateJig` plan has the responsibility of assigning a jig to an incoming part. The plan tests to see if a jig is available (`busyJig()`) whenever a change occurs in jig status. The availability test involves determining whether the jig is in use by another part and, if it is not, whether it is in the loading position for the cell.

```
plan AllocateJig extends Plan
{
    #uses data Jigs jigs; // Jigs is a JACK beliefset

        ...

    body()
    {
        while ( busyJig() )
            @wait_for( new Change( jigs, false ) );
        // get here if a jig is available
        // allocation code goes here
    }

    #reasoning method busyJig( )
    {
        // code to test if the jigs are busy
        // it fails if a jig is in the correct position
        // and is empty
    }
}
```

7.7.3 Action Cursors and RepeatAction Cursors

If there is a need to include an action within a plan that may take a long time to complete (such as an external database access or a computationally intensive calculation), the use of an action cursor (`aos.jack.util.cursor.Action`) is recommended. The programmer extends the `aos.jack.util.cursor.Action` class, overriding its `action()` method with one that incorporates code that performs the time-consuming action. The user-defined cursor can then be incorporated in an `@wait_for` statement within a plan. When the `@wait_for` statement is executed, the action is performed in a separate thread and the plan waits until the cursor returns true, indicating that the action is complete.

Note: There is an `@action` reasoning statement which provides a more compact way of writing in-line action cursors.

As an example, suppose that an application required a robot to perform a pick and place action. One could encapsulate the action within an action cursor and perform the action within an `@wait_for` statement using code like the following:

```
import aos.jack.util.cursor.*;

public plan PickAndPlace extends Plan
{
    // a move event specifies part type, pick location and
    // place location
    #handles event  Move m;

    class MoveAction extends Action
    {
        int part;           // part type
        int from;           // pick location
        int to;             // place location

        MoveAction(int part,int from, int to)
        {
            part = p;
            from = p1;
            to = p2;
        }

        protected void action()
        {
            // code to pick and place the part
        }
    }

    body()
    {
            ...

        Cursor c = new MoveAction(m.part,m.from,m.to);

        //action starts the first time completion is queried
```

```
            @wait_for(c);
                ...
        }
    }
```

Another example of the usage of Action cursors can be found in *Example 2* in the *Views* chapter.

Note: If any plan step within a reasoning method (e.g. `body()`) takes more than 2 time slices (200ms) to complete, the following message will be displayed (with details about the agent and plan involved):

```
New Executor spawned due to plan step taking too long
```

This might be the result of a programming error (e.g. a call to Java code which never returns), or it might be a legitimate piece of code that just takes a long time to execute. The message indicates that the JACK kernel has dissociated itself from the thread that is taking too long and has started a new thread in its place for the continued processing of the other agents. If the code executing on the original thread ever finishes, the thread terminates and normal plan processing continues on the new thread. If the message was not caused by a programming error it is indicative of sub-optimal use of JACK and should be avoided. One could either incorporate the offending code into an action cursor which is then used to trigger a `@wait_for` statement, or directly into an `@action` statement.

The repeat action cursor provides a capability for repeatedly performing the same action. It is created in the same way as the action cursor, except that we create a class which extends `RepeatAction` (rather than `Action`). The user provides a constructor and `action()` method as before. If we require the action to be performed at regular intervals, the desired effect can be achieved by coupling an again cursor with a repeat action cursor.

7.7.4 Beliefset Cursors

The beliefset cursor statement is used to query beliefset relations. This is done most often in a plan's `context()` method and in reasoning method statements that take a logical condition, such as `@wait_for`, `@maintain`, `@achieve`, `@test`, `@insist` and `@determine`.

Beliefset cursor statements use unbound logical members to perform a pattern-match search over the relation's tuples. If any tuples can be unified with this pattern the beliefset cursor returns true.

Beliefset cursors and their behaviour are best explained by means of an example. Suppose a beliefset relation is defined for a political commentary agent that describes the ministers in politics. This relation might be defined as shown in the following code:

```
beliefset Politician extends OpenWorld
{
   #key field String name;
   #value field String party;
   #value field String portfolio;
   #indexed query
       who(logical String name,String party,logical String port);

   // other #indexed and/or #linear query definitions
}
```

Suppose that an agent makes use of this beliefset relation by having the following declaration included in its definition:

```
#private data Politician politician();
```

The agent now has a private relation called `politician()` of type `Politician`. Suppose that the agent's current tuples for this relation are as shown in the following table:

name	party	portfolio
Mr Important	Sensible Party	Prime Minister
Ms Action	Sensible Party	Minister for Sport
Mr Knockout	Silly Party	Minister for Sport

Table 7-25: Tuples stored in the Politician beliefset

If the agent has two unbound String logical members called `name` and `portfolio`, it could query this relation using the beliefset cursor statement shown in the following code.

```
politician.who(name, "Sensible Party", portfolio)
```

The beliefset cursor uses the appropriate `who()` query method for the relation based on the parameters. In the example above there is only one `who(...)` query method defined.

Unbound logical variables are used as *output* parameters for a query. Bound logical variables are still treated as *output* parameters but they can only match the already bound value.

Note: JACK considers logical members to be a completely different type to normal members, so passing a logical member where a normal parameter is expected will cause a type mismatch error. All logical variables are converted to the type `aos.jack.jak.logic.Variable`.

When the above expression is evaluated, the agent looks through each tuple and attempts to match it with the pattern provided by unifying the logical members.

- If a *matching tuple is found*, the logical members are *bound* to this tuple's values and the beliefset cursor statement returns *true*.

- If *no matching tuple can be found*, the logical members remain *unbound* and the beliefset cursor statement returns *false*.

In this example, the first tuple will match the pattern provided when `name` is bound to "Mr. Important" and `portfolio` is bound to "Prime Minister". The party field matches that specified in the pattern, so binding the logical members in this way will provide an exact match with the relation's first tuple. Hence, if executed under these circumstances, the beliefset cursor expression will return true, and the logical members will be bound to their designated values.

Hence, a beliefset cursor appears to always return the first matching tuple in the relation. However, when it appears in a composite logical expression, subsequent tests may reject the first tuple returned and the agent may have to back-track and bind with the next matching tuple instead. This is discussed in more detail in the *Composite Logical Expressions* section in this chapter.

7.7.5 Enumeration Cursors

Enumeration cursors iterate over a `java.util.Enumeration`. True is returned while elements are available in the enumeration; false is returned when the end of the enumeration is reached.

An enumeration cursor has the following constructor:

```
EnumerationCursor(
      java.util.Enumeration enumeration, Variable variable )
```

The parameters are described in the following table:

Parameter	Meaning
enumeration	The enumeration to be iterated through.
variable	A logical variable which will be bound to each element of the enumeration in turn.

Table 7-26: The parameters in the `EnumerationCursor` constructor

An enumeration cursor becomes true while there are more elements to be bound and is false otherwise.

The cursor's `next()` method calls `bindValues()` with the current element as its argument. The default implementation of `bindValue()` is to return true with no binding taking place. In most situations this is not the desired behaviour and the user must extend `aos.jack.util.cursor.EnumerationCursor` and override `bindValues()`. An example is shown in the following code.

```
public class CertainTypeEnumeration extends EnumerationCursor
{
    Variable variable; // logical CertainType $v;

    public CertainTypeEnumeration(Elements e,Variable x)
    {
        super( e, x );
        variable = x;
    }

    public bindValues(Object element)
    {
        if ( element instanceof CertainType )
            return variable.unify( (CertainType) element );
        return false;
    }
}
```

7.7.6 Array Cursors

`aos.jack.util.cursor.ArrayCursor` is a non-triggered cursor which can be used to bind logical variables to the values of an array. Successive calls to `next()` provides one binding at a time until the array is exhausted.

An `ArrayCursor` has two constructors:

```
public ArrayCursor(Object[] array, Variable var)
```

and

```
public ArrayCursor(Object[] array, Variable var, int start, int end)
```

The latter is for iterating through the subarray which starts at index `start` and ends with index `(end - 1)`.

The parameters are described in the following table:

Parameter	Meaning
array	The array to be iterated through.
var	A logical variable which will be bound to each element of the array in turn.
start	Specifies the start of a region in the array of interest.
end	Specifies the end of a region in the array of interest.

Table 7-27: The parameters in the `ArrayCursor` constructor

The `ArrayCursor` is particularly useful for use within a context condition in a plan. For example:

```
plan X extends Plan {
    #handles event interestedParties ev;

    logical String party;

    context()
    {
        new ArrayCursor(ev.parties, party);
    }


   body()
   {
      System.out.println(party.as_string());
   }
}
```

In this case, a plan instance is generated for each element in `ev.parties`, with `party` bound to that element.

7.8 Plan Programming Guide

This section gives some advice on how to write the plans that agents will use. It presents some typical plan templates that can be used as a starting point, and gives some advice on abstracting agent tasks into a set of related plans.

7.8.1 Plan Definition Templates

This section provides typical plan templates for both normal plans and meta-level plans. Since the fundamental purpose of meta-level plans is distinct from that of normal plans, their templates are slightly different.

7.8.1.1 Normal Plan Template

A template to use for defining normal plans is given below.

```
plan PlanType extends Plan
{
   // A declaration of the event that the plan handles

   #handles event EventType reference;

   static boolean relevant (EventType reference)
   {
      // code to determine whether the plan is relevant
      // to an instance of the above event.
   }

   context()
   {
      // logical condition to test applicability
   }

   // Declarations of any events that the plan may post

   #posts event Event1 handle1;

   #sends event MessageEvent1 messagehandle1;

   // Declarations of any beliefset relations that the
   // agent uses.

   #reads data Relation1 relation_name1;
   #modifies data Relation2 relation_name2;

   // Declaration of the interface that any agent which
   // uses this plan must implement.  This is optional and
   // only applies if the plan will be shared between
   // different agents.

   #uses agent implementing InterfaceType InterfaceName;

   #uses interface InterfaceType InterfaceName;

   // Declarations of all the reasoning methods that the
   // agent can execute when performing this plan.

   #reasoning method methodName (parameters)
   {
      // Code for the agent to perform.
   }

   body()
   {
      // The plan body. This is the main reasoning
      // method that the agent runs when it
      // executes an instance of this plan.
   }
}
```

7.8.1.2 Meta-level Plan Template

A template to use for defining meta-level plans is given below.

```
plan PlanType extends Plan
{
   // A declaration that this plan handles the PlanChoice
   // event (and therefore is a meta-level plan) along with
   // a declaration of the BDI events that this meta-level
   // plan should be used to choose applicable plan
   // instances for.

   #handles event PlanChoice event_handle;
   #chooses for event event1 event2 event3 ...
   #chooses for event eventn eventn+1 ...

   static boolean relevant (EventType reference)
   {
      // code to determine whether the plan is relevant
      // to an instance of the above event.
   }

   context()
   {
      // logical condition to test applicability
   }

   // Declarations of any events that the plan may post.

   #posts event Event1 handle1;
   #sends event MessageEvent1 messagehandle1;

   // Declarations of any beliefset relations that the
   // agent uses.

   #reads data Relation1 relation_name1;
   #modifies data Relation2 relation_name2;

   // Declaration of the interface that any agent which
   // uses this plan must implement.  This is optional and
   // only applies if the plan will be shared between
   // different agents.

   #uses agent implementing InterfaceType InterfaceName;

   // Declarations of all the reasoning methods that the
   // agent can execute when performing this plan.

   #reasoning method MethodName (parameters)
   {
      // Code for the agent to perform.
   }

   body()
   {
      // The plan body. This is the main reasoning
      // method that the agent runs when it
      // executes an instance of this meta-level plan.
```

```
        // This reasoning method should reference the
        // event handle, as it has the applicable set.
    }
}
```

7.8.2 Functional Abstraction

Of all JACK structures, the plan is usually the most complex. As well as including a number of #-declarations, it will also frequently contain a number of reasoning methods to express its functionality. The main `body()` reasoning method will always be present, but if the actions that the agent must take when it executes the plan are reasonably complex, they may be abstracted into other reasoning methods, or other entire plans which the agent can call with BDI statements or `@subtask` statements.

In most cases, you should abstract a plan into separate reasoning methods which express subtasks that the agent needs to perform when executing the plan's `body()`. You should only abstract it into entirely separate plans (to be called by the `@subtask` statement) when:

- these plans already exist and can be reused; or
- the subtask requires the agent to perform different sequences of actions, depending on the agent's circumstances.

In this case, the subtask is really better expressed as an event. The different approaches to handling it can then be expressed in different sub-plans, which the agent can choose between at run time by their respective context conditions.

7.8.3 Logical Statements

Most of the core differences between JACK Agent Language semantics and those of ordinary Java involve the way logical statements are used and interpreted. In Java, boolean methods are just like normal methods that happen to return a boolean result. In the JACK Agent Language, however, the logical expressions hold more meaning.

Some of the differences between JACK Agent Language logical expressions and regular boolean expressions are listed below:

- JACK Agent Language logical expressions can follow either an *Open World* semantics or a *Closed World* semantics.

  Open World semantics models 'real world' knowledge. It allows for three truth states: *True*, *False* and *Unknown*. The default value is Unknown: unless you have found out that something is either True or False, you know nothing about it.

  Closed World logic follows the more pure boolean logic. It assumes that everything is known, and can either be True or False. The default is False: unless you have found out that something is True, you assume that it is False.

In particular applications it may be appropriate to model some agent beliefsets using Closed World semantics and other agent beliefsets using Open World semantics.

- When an unguarded JACK Agent Language logical expression fails in a reasoning method, it causes the reasoning method to fail.

  Reasoning methods are essentially sequences of statements that an agent must perform when executing a plan. The plan's body is a reasoning method, which can in turn contain other reasoning methods. Whenever a logical statement of any form is executed in a reasoning method, it will cause the reasoning method to fail if there is no alternative for the agent to try.

  For example, if a reasoning method contains an `if` condition without an `else` clause and the `if` condition fails, the reasoning method it appears in fails. If there is an `else` clause, however, the agent has somewhere to proceed and so the reasoning method does not fail: the reasoning method only fails when the agent is left with no alternative to try.

Logical statements are important in JACK. They are used to determine which instances of a plan are applicable, and to determine the success and failure of these plans and their respective components.

Although logical statements are common to many programming languages, some unique and important properties should be mentioned with regard to their use in JACK.

7.8.3.1 Components of a Logical Statement

The components from which logical statements in JACK can be built are listed below.

- Logical constants, boolean values and boolean expressions.

  These constitute logical expressions in most other programming languages. A single boolean member can be thought of as a trivial logical expression that is true or false depending on its value.

  This boolean member can be included in boolean expressions with other boolean members and logical constants. When done, the logical expression formed is true or false depending on the values of each member, each logical constant and the connectives used between them. The normal rules of boolean algebra are applied.

  For example, if an object includes two members, A and B, the right-hand side of the following Java statement:

  ```
  answer = A && B;
  ```

  is an example of a logical expression.

- A beliefset cursor expression, and boolean expressions containing beliefset cursor expressions.

Beliefset cursors are logical expressions that are used specifically to query beliefset relations. They are called cursors because their behaviour is analogous to cursors in database query languages. A beliefset cursor takes a pattern involving at least one unbound logical member and a beliefset relation. It then attempts to unify this pattern with the relation's tuples.

Unification involves the agent trying to match each tuple with the pattern where any unbound logical member is a 'wild card'. If the agent can match a tuple, it temporarily unifies the pattern's logical member(s) to its values. Often, the bound variable will then be tested or used in some way (for example, in a `@determine` statement it is used in a subtask execution). If it is used successfully and the unification succeeds, the logical member(s) are bound to the values that they were matched with. Once bound, they cannot be unbound. However, if the cursor is being used in an expression which can roll-back to before the binding took place (such as a determine statement can when the subtask execution fails), this 'frees' the logical member(s) so that another binding can be found.

- All other cursor expressions.

  All other cursors are logical expressions and can be used in composite logical expressions. For more details refer to the section on cursors in this chapter.

- a **F**inite **S**tate **M**achine (**FSM**) statement.

  FSM statements are JACK statements that the runtime engine executes in a threadsafe way by atomic steps. They can occur as logical statements but not in compound statements.

7.8.4 Logical Members

Logical members can be included in any JACK Agent Language definition, but have specific support in plans. Most often, they are used in a plan's `context()` method or reasoning methods. Logical members bring elements of logic programming to JACK.

Logical members follow the semantic behaviour of variables from logic programming languages such as Prolog. That is, they are not place-holders for assigned values like normal Java members: rather, they represent a specific, but possibly unknown, value. Conceptually, the murderer in a classic murder mystery can be represented by a logical variable. That is, the murderer is a specific person, but the detective may not know this person. As the detective investigates, he or she attempts to match new evidence with what he or she already knows about the murderer in order to uncover the murderer's identity.

The process of investigation that the detective in the above example goes through is analogous to the way that an agent uses logical members. Instead of assigning values to them, the agent attempt to uncover their value through a process known as *unification*. Unification involves attempting to match the logical member with a known pattern. The logical member, therefore, behaves like a 'wild card' in this pattern matching exercise. If it is possible to find a value for the wild card to make it match the pattern, the agent will treat this as a possible value for the logical member. It will *bind* the logical variable to this value.

Therefore, an unbound logical member is a member whose value is still unknown to the agent, whereas a bound logical member is a member whose value has been determined.

A logical member is defined with the plan's other members. Their definition takes the following form:

```
logical Type name;
```

Each component of this definition is described in the following table:

Component	Meaning
`logical`	Identifies this member as a logical member, as opposed to a normal Java member.
`Type`	Identifies the type of this logical member. Unlike logic programming languages where an unbound logical variable can represent anything, JACK Agent Language logical members are typed. When unbound their value is still unknown, but the agent does know that whatever the value is, it will be of the designated type. A logical member's type can be an object, any *scalar type* (`int`, `float` or `boolean`) or a `String`.
`name`	The logical member's name, used to identify it during plan execution.

Table 7-28: Components of a logical type definition

Once bound, a logical member's value cannot be changed. Returning to the murder mystery example, this is equivalent to uncovering the murderer. Once the murderer has been uncovered, the murderer is known. This cannot change.

Logical variables are objects that, when bound, require the use of an accessor to gain access to the values they represent. The available accessors are:

```
boolean as_boolean()
char as_char()
byte as_byte()
short as_short()
int as_int()
long as_long()
float as_float()
double as_double()
java.lang.Object as_object()
java.lang.String as_string();
```

Each of these will throw a `LogicalException` if the user tries to determine the value of an unbound logical variable.

Unification is a boolean method of the logical member. This method can be called directly from within a reasoning method if required.

If a logical variable is unbound it can be bound using `unify()`. For example, a logical member of type integer can be defined and then unified with the value '3' in a reasoning method as shown below.

```
logical int x;
x.unify(3);
```

This is like assigning a member the value '3', except that once unified, the value cannot be changed. In this case if `x` was unbound before `x.unify(3)`, `unify()` would return `true`. If `x` was already bound and you used `unify` to attempt to bind it again, it would return `true` if you attempted to bind it to its current value and `false` otherwise.

The previous example shows the simplest form of unification – where the agent simply designates a value. In the murder mystery example, it would be like the detective deciding that Colonel Mustard is the murderer (presumably after some flawless deduction). Sometimes, however, logical members are used in ways that are more complex. The agent may not simply be able to bind the member to a particular value. Instead, it may attempt to bind the member to a possible value, then perform some test to see whether the value is suitable. This can be thought of as testing the *hypothesis* that the logical member's value has been bound correctly.

For example, consider a `@determine` statement in the context of the murder mystery example (the detective agent is trying to determine the identity of the killer). The detective may find that the victim was shot and hence conclude that the murderer had access to a gun. If there are only three people who had access to a gun, the detective may hypothesize that one of these people is the murderer. This will be followed by further investigation that will prove whether this is the case. If it is, the killer has been uncovered. Otherwise, the detective will repeat the process with the second candidate.

In a `@determine` statement, this would correspond to there being three tuples that satisfy the statement's logical condition. The agent temporarily binds the condition's logical variable to the first tuple and then posts an event using this tuple's values. The agent *commits* the binding if this succeeds; otherwise, the binding is *rolled back* and the next tuple's values are bound. Rolling back is just like the detective finding out that his hypothesis is wrong and hence withdrawing the accusation of guilt.

Roll-back is the only way that the binding of a logical member can be 'undone'. It only occurs when the logical member is used in a *cursor* and when this cursor appears in a composite logical expression, or a compound statement such as the `@determine`statement.

7.8.5 Composite Logical Expressions

Individual logical statements in JACK can be combined using the standard boolean connectives available in Java ('`&&`', '`||`' and '`!`' representing the logical connectives `AND`, `OR` and `NOT` respectively).

When a logical expression is used, the agent always evaluates it in a *short-circuit, left to right* manner. That is, the agent will start evaluating the logical expression from the left-most statement and will stop evaluating as soon as it has found enough information to determine whether the expression is *true* or *false*.

For example, suppose the beliefset relation described in the example from the previous section is used in the following logical expression (where `name` and `portfolio` are unbound logical members of type `String`):

```
politician.who(name, "Silly Party", portfolio)
       && portfolio.as_string().equals("Minister for Sport");
```

Evaluation is performed left to right, so that the agent first looks at the `politician` relation's tuples to unify the query pattern with each tuple. The first and second tuples do not have a `party` field that matches the "Silly Party" string, however the third one does. Hence, the agent binds `name` to "Mr. Knockout" and `portfolio` to "Minister for Sport", and the beliefset cursor returns *true*.

The agent now moves onto the second expression, which compares the value of `Portfolio` with the `String`, "Minister for Sport". This matches, so the logical expression is true.

As well as returning *true*, the logical expression has bound the logical members `name` and `portfolio` to values that satisfy the statement. If this expression were to appear in a `context()` method, it could lead to an applicable plan instance. In that case, the logical members would capture the context in which the plan was applicable.

Now consider a logical expression of the following form instead:

```
politician.who("Fred", "Silly Party", portfolio)
       && portfolio.as_string().equals("Minister for Sport")
```

Again, the agent starts with the left-most beliefset cursor and attempts to unify it with the relation's tuples. However, there is no tuple whose name field matches the "Fred" string, so the portfolio member will remain unbound and the beliefset cursor will return *false*.

The logical expression is of the form `A && B`, so if `A` is false there is no way that the expression will be true. Therefore, the agent does not test the second comparison (`B`). Evaluation is also short-circuiting, so the agent considers the logical expression to be false immediately. It is important that the agent does not try to evaluate the comparison statement at this point because the `portfolio` member is still unbound. Any attempt to read its value will throw a `LogicalException` exception.

Finally, consider a third logical expression as shown below:

```
politician.who (name, "Sensible Party", portfolio)
       && portfolio.as_string().equals("Minister for Sport")
```

In this logical expression, the beliefset cursor can unify with the first tuple:

```
("Mr. Important", "Sensible Party", "Prime Minister")
```

The `name` and `portfolio` members are bound in the same way as before, but when the second conjunct is tested, `portfolio`'s value "Prime Minister" does not match the expected string. The evaluation will back-track to the previous conjunct and try to choose a new binding for `name` and `portfolio`. In our example, `name` will become bound to "Ms. Action", `portfolio` will be bound to "Minister for Sport" and the query will return true. The entire logical expression succeeds, with the logical members bound as follows:

```
name = "Ms. Action"
portfolio = "Minister for Sport"
```

8 Meta-Level Reasoning

In order for an event to be processed by an agent, the agent must determine which plan (if any) should handle the event.

Note: Each plan is capable of handling only a single event type which is specified by the plan's `#handles event` declaration. However, it is possible that there is more than one plan capable of handling a particular event type. As will be discussed later, it is also possible for multiple plan instances to be generated from a single plan.

To decide which of the plans are applicable, JACK employs the following steps:

1. Identify the plans which *handle* the event type.
2. Use the `relevant()` method to eliminate plans on the basis of the data associated with the particular event instance.
3. Use the `context()` method to generate plan instances which are consistent with the agent's current beliefs.

The last step results in what is known as the applicable plan set. Elements in the applicable plan set are ordered according to the order in which their corresponding plans were declared – this ordering is called *prominence*. This ordering can be overridden by a process called *precedence*, where a ranking is provided for each entry via the `getInstanceInfo()` method.

Having generated the applicable plan set for an event, one entry must be chosen for subsequent execution. Normally JACK will perform this selection, choosing either the first entry in the set, or choosing one at random. The exact mechanism used is tunable via behaviour attributes associated with the event (this is discussed in detail in the section *Customising BDI Behaviour with Behaviour Attributes* in the *Events* chapter). Note that when a choice is being made, only the highest precedence entries are considered. If the event is a BDI event and there is more than one entry in the applicable plan set, a `PlanChoice` event is posted. If plans are provided to handle this event (referred to as *meta-level plans*), the applicable set can be interrogated and the entry deemed most appropriate for the current situation selected. Note again that the available choices are restricted to the highest precedence entries – there may be additional entries of lower precedence, but they will be considered only when all entries of higher precedence have been exhausted.

8.1 Applicable Set Generation

8.1.1 Handling the Event Type

Most #- declarations are optional in a plan definition, but the `#handles event` declaration is mandatory. It specifies for which event type the plan type may be relevant. Whenever an instance of this event occurs, the agent will consider this plan as a potential candidate. Unless the plan's `relevant()` method indicates otherwise, the agent will assume that this plan is relevant to the event.

The `#handles event` declaration takes the following form:

```
#handles event EventType event_ref;
```

`event_ref` is available within the plan to reference the event being handled and its data fields.

8.1.2 Relevance

To be *relevant* to a given event instance:

a) a plan must declare that it handles the event type that has arisen through a `#handles event` declaration; and

b) if a `relevant()` method is present, a value of `true` must be returned.

If a plan does not have a `relevant()` method, the plan is relevant for all instances of the event.

In the following example, the `relevant()` method ensures that the plan is only considered if there is a colour specified in the `colour` data member of the `Paint` event.

```
public plan PaintSpecifiedCurrentColour extends Plan
{
    #handles event Paint ev;

    static boolean
    relevant{Paint ev}
    {
        return ev.colour != null && ev.colour.length()>0;
    }

    // context method, discussed below.

    body()
    {
        // As appropriate to the plan
    }
}
```

8.1.3 Applicability

The `context()` method provides the next level of 'filtering' after `relevant()`. If a plan is relevant to a particular event, the `context()` method determines whether the plan is applicable given the agent's current beliefs.

The `context()` method does not take any arguments and its body is always a single JACK Agent Language logical expression. Logical expressions are composed of boolean members, logical members and beliefset cursor expressions which can, in general, bind to multiple values. When evaluating the `context()` method, the agent will consider all possible alternatives. For every possible set of values that satisfy the `context()` method, a separate entry will be created in the applicable plan set.

In the following example, the `context()` method ensures that the plan is only applicable if the `Paint` event requests a specific colour that is the same as the agent's current `paintColour`.

```
public plan PaintSpecifiedCurrentColour extends Plan
{
    #handles event Paint ev;

    static boolean
    relevant(Paint ev)
    {
        // As appropriate to the plan
    }

    #uses interface Robot self;

    context()
    {
        self.paintColour.equals(ev.colour);
    }

    body()
    {
        // As appropriate to the plan
    }
}
```

In the following plan, the `context()` method queries a BOM (Bill of Materials) beliefset using `getSubcomponent()`. The plan will only be applicable if the `getSubcomponent()` query can find a tuple with a key of `component` in the beliefset. Note that it is quite normal for a component to have more than one sub-component – if this is the case, entries corresponding to *each* sub-component will be added to the applicable plan set.

```
public plan FindSubcomponentPlan extends Plan
{
    #handles event FindSubcomponent fsc;

    static boolean
    relevant(FindSubcomponent fsc)
    {
        // As appropriate to the plan
    }

    #reads data BOM bom;

    context()
    {
        bom.getSubcomponent(fsc.component, sc);
    }

    body()
    {
        // As appropriate to the plan
    }
}
```

As usual, only one plan instance will be attempted, and a second one is only tried if the first one fails.

8.1.4 Prominence

In the absence of precedence, the order in which plans appear in an agent or capability (the order of the `#uses plan` declarations) determines the order in which the corresponding entries appear in the applicable plan set.

For example, the agent below uses two plans. Prominence dictates that entries arising from `PaintSpecifiedCurrentColour` will appear in the applicable plan set before entries arising from `PaintAnyColour`.

```
public agent Robot extends Agent
{
        ...

    #handles event Paint;

    #uses plan PaintSpecifiedCurrentColour;
    #uses plan PaintAnyColour;
        ...
}
```

Note that inner capabilities are more prominent than plans.

8.1.5 Precedence

Precedence is the second means of ordering entries in the applicable plan set. If a `getInstanceInfo()` method is provided in a plan when the entry for the applicable plan set is being generated, the `getInstanceInfo()` method will be called. `getInstanceInfo()` returns a `PlanInstanceInfo` object. The `PlanInstanceInfo` class has the following members and methods:

Name	Description
`public static final PlanInstanceInfo def[]`	An array of `PlanInstanceInfo` objects pre-constructed for ranks 0 – 9.
`public PlanInstanceInfo(int)`	For constructing an object with a given rank.
`public int rank`	The rank.
`public int getRank()`	Returns the rank.

Table 8-1: `PlanInstanceInfo` class members and methods

Note that a data member is available for holding a rank – the entries in the applicable plan set are ordered according to rank. In order to simplify the ranking process, an array of predefined `PlanInstanceInfo` objects (`def[]`) is provided. The array index corresponds to the rank – 0 is the lowest rank and 9 is the highest rank. For example, the following plan will have a rank of 4:

```
public plan PaintSpecifiedCurrentColour extends Plan
{
    #handles event Paint ev;
        ...

    public PlanInstanceInfo getInstanceInfo()
    {
        return PlanInstanceInfo.def[4];
    }
        ...
}
```

If a second plan is now introduced:

```
public plan PaintAnyColour extends Plan
{
    #handles event Paint ev;
        ...

    public PlanInstanceInfo getInstanceInfo()
    {
        return PlanInstanceInfo.def[5];
    }
        ...
}
```

then entries arising from the `PaintAnyColour` plan will always precede those arising from the `PaintSpecifiedCurrentColour` plan regardless of their prominence.

Note that `getInstanceInfo()` is executed after *each* binding of the `context`() method. Thus the current binding is available to `getInstanceInfo()` and can be used in determining the rank to be returned, as shown in the following example:

```
public plan DoSomething extends Plan
{

    #handles event SomeEvent ev;
    #reads data Order order;
    // BeliefSet "Order order(tag,precedence)" has tuples
    // of the form:
    //
    //   <"important",8>, <"normal",5>, <"background", 3>
    // and the incoming event includes a tag to match.

    logical int precedence;

    context()
    {
        order.get( ev.tag, precedence );
    }

    public PlanInstanceInfo getInstanceInfo()
    {
        try
        {
            return PlanInstanceInfo.def[ precedence.as_int() ];
        }
        catch (LogicException ex)
        {
            return PlanInstanceInfo.def[ 5 ];
        }
    }
        ...
}
```

When `context()` results in multiple bindings (as could occur in the earlier BOM example), `getInstanceInfo()` is executed after *each* binding. Thus, each entry added to the applicable plan set could be assigned a different ranking.

8.2 The Applicable Plan Set

The applicable plan set is used to determine which plan instance of those applicable to the event under current consideration should be executed. A plan instance is defined by:

- the plan type instance which is handling the event instance in question, and
- a binding of logical variables arising from execution of the `context()` method.

Note that multiple plan instances may arise from a single plan type instance.

In order to facilitate reasoning about plan instances JACK provides the `Signature` class. It provides a representation which enables plan instances to be uniquely identified and efficiently compared for logical equivalence. Note that a signature does not contain a reference to the actual plan type instance – rather, it keeps all the information needed for re-establishing the plan instance.

The `Signature` class provides the following methods for use by the programmer:

Name	Description
`public Event getEvent()`	Returns the originating event.
`public PlanInstanceInfo getInfo()`	Returns the `PlanInstanceInfo` object associated with this signature.
`public Plan getPlan()`	Returns a 'plan factory' which can be used to determine the type of the plan concerned, and to access the logical plan variables associated with the plan.

Table 8-2: `Signature` class methods

One can gain access to the `PlanInstanceInfo` object that was returned by the plan's `getInstanceInfo()` method via the `getInfo()` method.

The `ApplicableSet` class extends the `SignatureList` class and one can therefore access the applicable plan set using the following methods:

Name	Description
`public Signature first()`	Returns the first signature in the list.
`public Signature last()`	Returns the last signature in the list.
`public Signature next(Signature)`	Returns the signature in the list after the one given, or `null` if the one given is the last signature.
`public Signature prev(Signature)`	Returns the signature in the list before the one given, or `null` if that given is the first signature.
`public int size()`	Returns the number of signatures in the list.

Table 8-3: Methods to access the applicable plan set

Thus, given a reference to the applicable plan set for an event, one can iterate through the `Signature` objects, inspect their corresponding `PlanInstanceInfo` objects and decide which signature is most appropriate for the current situation. Note that the applicable set only contains the highest precedence signatures – lower precedence signatures become 'visible' only after all higher precedence signatures have been used and failed.

If one is going to explicitly select a signature, additional information regarding the plan instances will invariably be required. This can be provided by extending the `PlanInstanceInfo` base class and then creating a suitable `PlanInstanceInfo` instance in the `getInstanceInfo()` method.

8.3 Choosing a Plan Instance

With normal events, the agent selects an entry from the applicable plan set for a given event and executes only the plan instance associated with that entry. As noted earlier, the mechanism used for selection (pick the first entry in the list or a pick an entry at random) is tunable via behaviour attributes associated with the event.

With BDI events, if the applicable plan set contains more than one entry, a `PlanChoice` event is posted. *PlanChoice* events are described in detail elsewhere; for the purposes of this discussion note that the `PlanChoice` class allows access to the following data members:

Name	Description
`public Event event`	Holds the object level event instance that caused the plan choice event.
`public ApplicableSet applicable`	Holds information about the current set of applicable plan instances.
`public FailureSet failure`	Holds information about the current set of failed plan instances.
`public Signature chosen`	Assigned by the plan choice handling plan.

Table 8-4: `PlanChoice` class data members

If the user has provided meta-level plans to handle the `PlanChoice` event, one uses the `PlanChoice` data members to reason about the applicable plan set and to determine which is the most appropriate plan instance in the current situation. This process is called meta-level reasoning.

A meta-level plan is defined to `#handle` the `PlanChoice` event. It may further include one or more `#chooses for` statements to constrain to which object-level events the plan is relevant.

This is discussed in more detail in the section on *Plan Declarations* in the *Plans* chapter. The plan will also access the data members of the particular `PlanChoice` event instance.

These features are illustrated in the following simple example which prints "Hello World" in different languages. Each language is identified by a number (1 for English, 2 for Swedish ...) and each plan `#handles` an event of type `TransEvent`. This event has a single data member which contains the requested language number. The code for `TransEvent` is not listed but note that it *must* extend `BDIGoalEvent`. Meta-level reasoning is used to select the appropriate plan instance. Note that this example is for pedagogical purposes only – there are much simpler ways to achieve the same result.

```
public class LanguageType extends PlanInstanceInfo
{
    public int language;

    public LanguageType(int i)
    {
        super(5);          // 5 is the default precedence
        language = i;
    }
}
```

```
plan Trans1Plan extends Plan
{
    #handles event TransEvent ev;

    public PlanInstanceInfo getInstanceInfo()
    {
        return new LanguageType(1);
    }

    body()
    {
        System.out.println("Hello World");
    }
}

plan Trans2Plan extends Plan
{
    #handles event TransEvent ev;

    public PlanInstanceInfo getInstanceInfo()
    {
        return new LanguageType(2);
    }

    body()
    {
        System.out.println("Tjena Moss");
    }
}

plan ChooseLanguage extends Plan
{
    #handles event PlanChoice ev;
    #chooses for event TransEvent;
    body()
    {
        TransEvent te = (TransEvent) ev.event;

        for ( Signature s = ev.applicable.first();
              s != null ;
              s = ev.applicable.next( s )
            )
        {
            if ( s.getInfo() instanceof LanguageType )
            {
                LanguageType p = (LanguageType) s.getInfo();
                if ( p.language == te.n )
                {
                    ev.chosen = s;
                    return true;
                }
            }
        }
    }
}
```

9 Beliefset Relations

9.1 Introduction

Beliefsets are used in JACK to maintain an agent's beliefs about the world.

An agent's beliefset can be stored as either an `OpenWorld` or a `ClosedWorld` class. The beliefset represents these beliefs in a *first order, tuple-based relational* model. The logical consistency of the beliefs contained in the beliefset is automatically maintained. Hence, for example, if an agent adds a belief that contradicts a belief it already has, the beliefset detects this and automatically removes the old belief.

The beliefset is not the only way that an agent can represent information. Agents can also include ordinary data members and other data structures that have been implemented in Java. However, the advantage of using a beliefset over normal Java data structures is that beliefsets have been designed specifically to work within the agent-oriented programming paradigm. Therefore, it is fully integrated with the other JACK Agent Language classes, and provides facilities not available with other data storage techniques. In particular, a JACK beliefset provides:

- Automatic maintenance of *logical consistency* and *key constraints*.
- Either *Open World* or *Closed World* logic semantics for maintaining these beliefs.
- The ability to *post events* automatically when changes are made to the beliefset, and hence initiate action within the agent based on a change of beliefs.
- The ability to support *beliefset cursor statements*, providing a distinct tuple that unifies with the cursor's query expression each time the cursor attempts to rebind the query (in a complex logical expression).

Each beliefset class definition that an agent uses is called a *beliefset relation* . It describes a set of beliefs that the agent may have in terms of *fields*. When the agent wants to adopt a new belief, it specifies values for each of these fields and *adds* this belief to the relation. This creates a *tuple* for the relation. Every belief that an agent currently has stored in a given beliefset relation is represented as a tuple.

Tuples can either be *true* or *false*. This models the 'belief' aspect of the tuple. If the tuple is true, the agent believes that it is a true statement. If it is false, the agent believes that it is a false statement. For example, an agent may have a tuple to represent the statement that Mr Important is Prime Minister and member of the Sensible Party. If this tuple is stored as being true, this indicates that the agent will consider the statement true. If it is stored as false, the agent will assume that this statement is false.

The fact that beliefset relations represents their data as beliefs rather than 'absolute truths' distinguishes them from most other programming storage mechanisms and allows agents to more realistically exhibit rational behaviour. Agents do not treat what they know as absolute truth, but rather as beliefs that reflect what they have learned or have been told about the world to this moment. Like people, they will operate on the assumption that these beliefs are true until more information comes to light, but if something new is uncovered that contradicts them, they will update these beliefs accordingly.

9.2 Beliefset Definition

A beliefset definition uses a relational model to specify an agent's knowledge capacity. This knowledge capacity is expressed as a relation that the agent can use to express beliefs with. Each belief is represented by a specific set of values for each of the relation's fields. The general format for a beliefset relation's definition takes one of the two forms shown below:

```
beliefset RelationName extends ClosedWorld
{
   // Zero or more #key field declarations.
   // These describe the key attributes of each belief.
   // Zero of more #value field declarations
   // These describe the data attributes of each belief.
}

beliefset RelationName extends OpenWorld
{
   // Zero or more #key field declarations.
   // These describe the key attributes of each belief.
   // Zero of more #value field declarations
   // These describe the data attributes of each belief.
}
```

Each component of this definition is explained in the following table:

Syntax Term	Description
`beliefset`	JACK Agent Language keyword, identifies a beliefset relation's declaration.
`RelationName`	Used to identify the beliefset relation. Whenever an agent wants to query or modify this relation's tuples, it does so by using this name.
`extends ClosedWorld`	Identifies the beliefset relation as a *Closed World relation*. Closed World relations are relations that store *true* tuples and assume any tuple not found is *false*.
`extends OpenWorld`	Identifies the beliefset relation as an *Open World relation*. Open World relations are relations that store *true* and *false* tuples and assume any tuple not found is *unknown*.

Table 9-1: Components of a beliefset definition

For an agent to be able to use a beliefset, its agent definition file **must** include a data declaration. The data declaration describes the type of beliefset required by the agent and specifies an external name (*`BeliefType`*) and an internal name (*`beliefName`*) for the beliefset. The external name of the beliefset maps to a beliefset definition file of the same name.

Each of an agent's plans that makes use of a beliefset must contain a declaration specifying read-only or read-write access to the beliefset. This is achieved by using either a `#reads data`, a `#modifies data` or a `#uses data` plan declaration.

9.2.1 Closed World Relations

Closed World relations assume that the agent is operating in a world where every tuple that the relation can express is stored in the beliefset at all times as being either true or false. This means that there is no query the agent can make for which it does not have an answer because, theoretically, every possible tuple is always represented in the beliefset. All that the agent can change is whether it believes the tuple to be true or false.

Of course, most tuple fields have an almost infinite range of values, and hence in practical terms the beliefset cannot store every possible tuple. Instead, only those tuples that the agent believes to be true are stored. Any tuple that is not stored, therefore, is assumed to be false.

In a Closed World relation, adding a tuple to the beliefset causes the agent to believe what was false to now be true. Similarly, removing a tuple causes the agent to believe what was true is now false.

Closed World relations do not often occur in the real world, but are still useful in many applications. For example, consider an agent that plays chess. It needs to know the positions of the pieces on the board. This information could be modelled using Closed World relations.

9.2.2 Open World Relations

Open World relations model knowledge and beliefs as most people in the real world experience them: for any given set of beliefs, only some of the answers are known to the agent. Some things may be known to be true, others known to be false and still others unknown.

Unlike the Closed World example, therefore, Open World relations store both true and false tuples. This models the situation where the agent does not know what something is, but does know what it is not. For example, consider a detective in a classic murder mystery. This detective may not know who the murderer is, but may believe that the murderer is definitely not Ms Scarlet (due to some prior investigation). To reflect this knowledge, the agent will store the statement, "Ms Scarlet is the murderer" in the beliefset as a false tuple – one that is believed not to be the case.

Because Open World relations record both true and false tuples, any tuple that is not stored in the beliefset is assumed to be unknown. That is, the agent does not know whether the tuple is true or not.

Unlike Closed World relations, therefore, Open World relations effectively work with three logic values: *true*, *false* and *unknown*.

9.3 Beliefset Members and Methods

Just like the other JACK Agent Language classes, beliefsets provide a number of base members and methods that you can access. These members and methods are described in the following subsections.

Beliefset Construction

JACK supports three types of beliefsets; private, agent and global. Only private beliefsets can be populated by a beliefset constructor *and* manipulated by plans that use the `add()` and `remove()` methods. Beliefsets that are specified in an agent declaration as being agent or global are read-only, so after they are populated by their constructor they can only be queried. As the default constructor creates an empty beliefset when the agent that uses it is instantiated, it only makes sense to use the default constructor for private beliefsets.

A beliefset can be populated with an initial set of tuples by either writing a constructor that reads the required data from a file and explicitly adding the records to the beliefset or by using JACOB Object Modelling.

The following example shows a beliefset constructor which reads data from a file and explicitly adds tuples to the beliefset.

```
import java.io.*;
import java.text.*;

beliefset Foo extends ClosedWorld
{
   #key field int tag;
   #value field double stamp;

   #indexed query get(int t, logical double s);

   static MessageFormat format =
         new MessageFormat("(foo {0, number} {1, number))");

   Foo(String name)
   {
      try
      {
         BufferedReader in =
             new BufferedReader(new FileReader(name));

         for (String line; ((line = in.readLine()) != null); )
         {
            try
            {
               Object [] data = format.parse(line);

               int t = ((Number)data[0]).intValue();
               double s = ((Number)data[1]).doubleValue();
               add(t, s);
            }
            catch (ParseException e) { }
         }
      }
      catch (IOException e)
      {
         System.err.println("Problems loading file "+name+".");
      }
      catch (BeliefSetException e)
      {
         System.err.println("Loading of file "+name+" failed.");
      }
   }
}
```

If multiple constructors have been defined, the JACK kernel determines which beliefset constructor to use on the basis of the number and type of arguments supplied.

An alternative method is to initialise tuple objects using JACOB Object Modelling. JACOB is described in more detail in the *JACOB Manual*. Beliefsets have a `read()` method which takes the name of a file as its argument. The `read()` method populates the beliefset according to the contents of the file which should contain data in JACOB format.

Beliefsets also have a `write(String filename)` method that can be used to write the beliefset contents in the appropriate JACOB form to the given filename.

The following example illustrates how a beliefset can be initialised in this way.

Given the beliefset

```
public beliefset BookData extends ClosedWorld {
  #key field String title;
  #key field String author;
  #value field double price;

  indexed query get(String t, String a, logical double p);
}
```

and an agent containing the declaration:

```
#private data BookData books();
```

The beliefset could be populated using the `read()` method as illustrated below:

```
books.read("book.dat");
```

The data file `book.dat` could contain data similar to the following:

```
<TupleTable
 :tuples (
   <BookData__Tuple
     :title "Reading is Fun"
     :author "Walter Fox"
     :price 23.75
   >
   <BookData__Tuple
     :title "Spelling is Fun"
     :author "Walter Fox"
     :price 23.75
   >
  )
 >
```

where `TupleTable` is a pre-defined object for the purpose of initialising beliefsets in this way. Also note that in the above example BookData__Tuple has two underscores.

Note: An `OpenWorld` beliefset would be initialised by two consecutive `TupleTable` objects; the first for `true` beliefs and the second for `false` beliefs.

In addition, beliefset classes have a constructor which takes a filename as an argument and uses the read method to populate the beliefset.

This means that by declaring the beliefset as follows:

```
#private data BookData books("book.dat");
```

the tuples are initialised with the data contained in `book.dat`.

void postEvent(Event *e*)

BeliefSet modification callbacks use this method to post events when changes are made to a beliefset relation. Therefore, when registering any beliefset callbacks with a beliefset relation, the relevant events using this method must be posted.

void add(*parameters*)

This method is automatically generated from the beliefset definition file by the JACK compiler – the key and value fields of the relation become arguments of the `add()` method, as illustrated by the following example.

```
beliefset Foo extends ClosedWorld
{
   #key field int a;
   #value field boolean b;
        ...
}
```

This results in the following methods being generated in the class `Foo`:

```
add (int __v0, boolean __v1);
```

This method is used to add tuples to `Foo`. It assumes that the tuple to be added has a belief state of *true*.

```
add (int __v0, boolean __v1, BeliefState __d);
```

This method could also be used to add tuples to `Foo`. However, given that `Foo` has Closed World semantics, it is only valid to add tuples with a belief state of true. Attempts to add tuples with a belief state of false or unknown will result in a `BeliefSetException` being thrown. If `Foo` had been defined to have Open World semantics, it would have been valid to add tuples with belief states of either true or false. Attempts to add tuples with an *unknown* belief state into a beliefset with Open World semantics will cause a `BeliefSetException` to be thrown.

The `add()` methods allow an agent to add tuples to any of its private relations, but not to any of its agent or global relations (as these relations cannot have their tuple set changed after creation).

void remove(*parameters*)

This method is automatically generated from the beliefset definition file by the JACK compiler – the key and value fields of the relation become arguments of the `remove()` method as illustrated by the following example.

```
beliefset Foo extends ClosedWorld
{
   #key field int a;
   #value field boolean b;
        ...
}
```

This results in the following methods being generated in the class `Foo`;

```
remove (int __v0, boolean __v1);
```

This method is used to remove tuples from `Foo`. It assumes that the tuple to be removed has a belief state of *true*.

```
remove (int __v0, boolean __v1, BeliefState __d);
```

This method could also be used to remove tuples from `Foo`. Given that `Foo` has been defined to have Closed World semantics, this form of the method should only be used to remove tuples with a belief state of true. Attempting to remove tuples with a belief state of false or unknown from a beliefset with Closed World semantics will result in a `BeliefSetException` being thrown.

If `Foo` had been defined to have Open World semantics, it would have been valid to remove tuples with belief states of either true or false. Attempting to remove tuples with an *unknown* belief state from a beliefset with Open World semantics will cause a `BeliefSetException` to be thrown.

The `remove()` methods allow an agent to remove tuples from any of its private relations, but not from any of its agent or global relations (as these relations cannot have their tuple set changed after creation).

`public int nFacts()`

This relation method returns the number of tuples stored in the relation at the time of calling. This includes all tuple instances that physically appear in the relation; therefore, the meaning and results are different depending on whether the relation follows an Open World or Closed World logical model.

For Closed World relations, this returns a count of the number of True tuples that are currently stored for this relation. For Open World relations, this returns a count of both the number of True and False tuples that are currently stored for this relation. For example, if a relation stores tuples to represent statements of, "The tie is blue", "The tie is not green" and "The tie is not red", calling `nFacts()` on this relation will return 3: one tuple representing true information and two tuples representing false information.

9.4 Beliefset Declarations

BeliefSet definitions can include the following declarations:

```
#key field FieldType field_name;
#value field FieldType field_name;
#indexed query methodName(parameters);
#linear query methodName(parameters);
#complex query methodName(parameters) <statements>
#function query return-type methodName(parameters) <statements>
#posts event EventType handle;
#propagates changes [EventType];
```

Each of these declarations are described in the following sections.

#key field *FieldType field_name*

This declaration is used to describe a beliefset relation's key fields. Key fields describe attributes that uniquely identify the object or entity that the tuple is referring to. Each belief that is expressed using a beliefset tuple is a belief about *something*. Hence, the agent needs some way of knowing whether two tuples refer to the *same* thing or not.

The relation's key fields are used for this purpose. They describe a uniquely identifying characteristic of the thing that the tuple refers to.

For example, suppose an agent had a beliefset relation to represent bank accounts. When a tuple is added to the beliefset, how does the agent know which bank account it refers to? How does it know whether this new information contradicts what it already believes about bank accounts?

With bank account information, this is normally done using an account number. By definition, the account number uniquely identifies a given bank account. Therefore, if the beliefset already contains a tuple stating that account 54321 contains $100, adding a (true) tuple that says account 54321 contains $200 contradicts and replaces this belief. Similarly, adding a tuple that says account 12378 has $200 has no effect on this tuple, because the agent knows from its key that this tuple refers to a different bank account.

A `#key field` declaration takes the following form:

```
#key field FieldType field_name;
```

Each component of this declaration is described in the following table:

Component	Meaning
`#key field`	Adds a key field to the beliefset relation. Values given for this field in tuples will be used to identify the object that the tuple is referring to, and hence to determine whether this tuple's data clashes with an existing tuple and needs to replace it.
`FieldType`	The field's data type. A beliefset relation's key fields are constrained to be of type `String`, any `scalar type` or any type that implements `aos.jak.beliefset.Immutable`. If the type is a user defined type, the user may need to override the `equals()` and `hashCode()` methods of its parent class – refer to the `java.util.Hashtable` documentation.
`field_name`	Used to identify the key field.

Table 9-2: Components of a `#key field` declaration

In the last example, the bank accounts had a single key field. This is not necessarily the case in all situations. Sometimes a relation may have multiple key fields (for example in a relation that describes geographic location, where the key might require two key fields: a site's latitude and its longitude). Similarly, a relation might have no key fields. When a relation has no key fields, this means that there is only ever one object that the relation refers to (hence, it does not need to be specified).

Note: When a beliefset relation has no key fields, this does not necessarily mean that the beliefset will only ever hold one tuple. If the relation is Open World, the agent may store multiple false tuples about the object. For example, there may be only one winner. However, if the agent does not know who the winner is but knows that it is not Mr Important, adding the fact that it is not Ms Action will not contradict its existing belief.

The knockout of existing tuples due to key constraints occurs for true tuples only. As the example above demonstrates, an Open World beliefset can have many negative tuples about something without having them contradicting one another. In fact, if the beliefset contains a positive tuple and the agent adds a negative tuple that doesn't contradict it, the two tuples will coexist in the beliefset as well ("Ms Action is the Minister for Sport" and "Ms Action is not the Prime Minister", though redundant, do not contradict one another). The only way negative tuples will knock out positive ones is if the two are directly contradictory (for example, "Ms Action is the Minister for Sport" and "Ms Action is not the Minister for Sport").

`#value field` *FieldType field_name*

This declaration is used to specify a relation's data fields. Unlike key fields, data fields do not identify the object that the tuple is describing. Instead, they represent information about this object that the agent needs to know. To a certain extent, the value fields are the reason why the agent has the relation in the first place – because it wants to know this information about some kind of object.

A `#value field` declaration takes the following form:

```
#value field FieldType field_name;
```

Each component of this declaration is described in the following table:

Component	Meaning
`#value field`	Adds a value field to the beliefset relation. Data assigned to these fields is used to represent attributes about the object of which the agent needs to be aware.
`FieldType`	The field's data type. A beliefset relation's value fields can be of any type.
`fieldName`	Used to identify the value field.

Table 9-3: Components of a `#value field` declaration

For example, a bank account's key field is its account number. However, the account number does not provide any information about the account that an agent will want to record. Its purpose is purely to distinguish one account from another. The sorts of things that the agent might want to know about the account are its balance, owner's name, credit limit, etc. Each of there attributes would be described using value fields.

`#indexed query` *methodName*`(`*parameters*`)`

Once a beliefset relation has been defined and tuples have been added to the beliefset, the agent will need to access this data. It does so by performing a query on the relation. There are two kinds of query that an agent can perform on a relation:

- an *indexed query* (defined by the `#indexed query` declaration); or
- a *linear query* (defined by the `#linear query` declaration).

Both these queries produce the same results (search for the tuple(s) concerned). The only difference is in implementation. Indexed queries maintain a hash index of tuples, and can usually locate them more quickly, whereas linear queries do not maintain an index; thus the only way that matching tuples can be found is through a linear search.

Indexed queries occupy slightly more space (which is required for the index) and are slightly slower to update tuples (since the index must be updated as well), but are much quicker to query in most circumstances. Therefore, unless memory and update speed is at an absolute premium, you should use indexed queries for all but the smallest of relations (i.e. those that will hold at least 10 tuples in the beliefset).

When a query is performed and most of the fields are unified with unbound logical variables, the agent may not have enough information to use the index effectively. In this case, the indexed query would be just as slow as the linear one.

An indexed query's definition takes the following form:

```
#indexed query methodName(parameters);
```

Each component of this definition is described in the following table:

Component	Meaning
`#indexed query`	Defines an indexed query, namely one that builds and maintains an internal hash table index for query optimisation.
`methodName`	The name of the query.
(*`parameters`*)	The list of parameters used by the query. These parameters are matched in order with the relation's tuples. Parameters that are defined as *normal members* are *input parameters*. Parameters that are defined as *logical members* are *output parameters*.

Table 9-4: Components of an `#indexed query` definition

Only the prototype needs to be declared for each indexed query. The actual query class will be a derived class of `BeliefSetCursor` generated by the JACK compiler. The indexing is done using only the non-logical parameters.

For example, suppose a beliefset relation is defined as shown below:

```
beliefset Politician extends OpenWorld
{
   #key field String name;
   #value field String party;
   #value field String portfolio;
   #indexed query
       getPortfolio (String n,String p,logical String port);
}
```

This beliefset relation has an indexed query, which takes `name` and `party` as input parameters, and if successful will return the matching tuple's `portfolio`.

A beliefset relation may have many query methods, each specifying different input and output parameters.

Beliefset queries take a number of parameters which are either data values (which can either be literal, normal members or logical members that have already been bound to a specific value) or unbound logical members. The query attempts to match the given parameters against the relation's tuples, using the unbound logical members as 'wild cards'. If a match can be found, the logical members are bound to the tuple's values.

It is possible to overload the query methods by providing different parameter lists. When such polymorphic query methods are defined, the compiler will select the definition that best matches the parameters provided.

For example, consider the following beliefset relation definition:

```
beliefset Job extends OpenWorld
{
   #key field String name;
   #value field String employment;
   #indexed query jobQuery(String n, String e);
   #indexed query jobQuery(String n, logical String e);
   #indexed query jobQuery(logical String n, String e);
   #indexed query jobQuery(logical String n, logical String e);
}
```

The same query name has been defined with all different combinations of input and output parameters. This means that any combination of `String` and logical variables can be queried.

`#linear query` *`methodName`*`(`*`parameters`*`)`

The `#linear query` method is identical to the `#indexed query` method in all respects, other than the way queries search the beliefset for matching tuples. The `#indexed query` builds an index that allows for search speed optimisation while the `#linear query` is more efficient in terms of memory usage.

Each component of a `#linear query` is described in the following table:

Component	Meaning
`#linear query`	Defines a linear query – namely one that the agent executes by attempting to unify with each tuple in the beliefset in turn.
`methodName`	The name of the query.
(*`parameters`*)	This list of parameters must be passed in a query. These parameters are matched in order with the relation's tuples. • Parameters that are defined as *normal members* are *input parameters*. • Parameters that are defined as *logical members* are *output parameters*.

Table 9-5: Components of a `#linear query` definition

In all other respects, linear queries are identical to indexed queries in terms of how they are defined and how they are used.

`#complex query` *name*(*parameters*) <*body*>

Complex queries provide a way to combine simple queries (as described above) into one entity that can be used in the same way as a simple query. Suppose that there is a beliefset which contains only `parent(parent, child)` relations and a `parent(parent, child)` query has already been defined. A `grandparent()` query could be written as follows, avoiding the need to add specific `grandparent(grandparent, grandchild)` relations to the beliefset.

```
beliefset Ancestors extends OpenWorld
{
    #key field String parent ;
    #key field String child ;

    #indexed query parent( String p , logical String c );
    #indexed query parent( logical String p , logical String c );

    #complex query
            grandparent(logical String a, logical String c)
    {
        logical String b;

        return parent(a, b) && parent(b, c);
    }
}
```

Note: the return value expression of a `complex query` incurs an implicit `.next()` in the same way as a condition expression in a `#reasoning method`

A `#complex query` declaration takes the following form:

```
#complex query name(parameters) <statements>
```

Each component of this declaration is described in the following table:

Component	Meaning
`#complex query`	Declares that the following method is a complex query.
`name``(`*`parameters`*`)`	`name` is the name of the query. Parameters can be of any type.
`<`*`statements`*`>`	The code which constitutes the query. The method body can contain arbitrary Java code, but the method must return a `Cursor`.

Table 9-6: Components of a `#complex query` definition

`#function query` *`ReturnType name`*`(`*`params`*`)` `<`*`body`*`>`

In JACK, the code for indexed and linear queries is generated automatically – the user only provides the function prototypes. With a function query the user provides the entire function definition so queries can be constructed which use arbitrary Java code. A function query can contain logical member definitions, so it can be used to query a beliefset from JACK entities that do not support logical members. In the following example, the function queries `#function query String parent(String b)` and `#function query int numChildren( String p )` could be used from within an agent method.

```
beliefset Ancestorship extends OpenWorld
{
    #key field String parent ;
    #key field String child ;

    #indexed query parent( logical String p , String c );
    #indexed query parent( logical String p , logical String c );

    #function query String parent(String b)
    {
      logical String a;

      parent(a,b).next();
      return a.as_string();
    }

    #function query int numChildren( logical String p )
    {
        logical String child ;
        int i = 0;
        Cursor c = parent(p,child);
        while(c.next())
          i++;
        return i;
    }

    #function query int numChildren( String p )
    {
        logical String lp;
        lp.unify( p );
        return numChildren( lp );
    }
}
```

Note:

1. There is a requirement for an explicit `.next()` in the function queries. Implicit `.next()` only occurs inside reasoning methods and in the return statement of a complex query.
2. For the purposes of method overloading, it is important to note that the JACK compiler converts all logical variables into variables of type `aos.jack.jak.logic.Variable`. So in the example above, no ambiguity exists between the two `numChildren()` queries.

A `#function query` declaration takes the following form:

```
#function query ReturnType name(parameters) <statements>
```

Each component of this declaration is described in the following table:

Component	Meaning
`#function query`	Declares that the following method is a function query.
`ReturnType`	Unlike a `#complex query`, a `#function query` can return any type.
`name``(`*`parameters`*`)`	`name` is the name of the query. Parameters can be of any type.
`<statements>`	The code which constitutes the query. The method body can contain arbitrary Java code.

Table 9-7: Components of a `#function query` definition

`#posts event` *`EventType handle`*

Beliefset relations are able to post events when changes are made to their tuples. This is done by posting an event within a *beliefset callback*. Beliefset callbacks are described in more detail below, but essentially they are methods that will be called when a particular beliefset change occurs. This declaration specifies the event types that may be posted from within any of the callbacks defined for this beliefset type.

A `#posts event` declaration takes the following form:

```
#posts event EventType handle;
```

Each component of this declaration is described in the following table:

Component	Meaning
`#posts event`	Identifies that the beliefset relation can post an event. Typically, the details of when this event is posted will be specified in a beliefset callback.
`EventType`	The type of event that this beliefset relation can post via callbacks.
`handle`	A handle on this event, so that the event's posting methods can be accessed.

Table 9-8: Components of a `#posts event` declaration

`#propagates changes` [*EventType*]

`#propagates changes` marks that a beliefset may be a source beliefset in a team belief connection, and it provides an implementation of the connection dynamics, so that changes to the beliefset are propagated correctly. Belief propagation is only available when using JACK Teams. Refer to the *Teams Manual* for more details.

This propagation includes filtering when the `#propagates changes` declaration is used with an optional event type. This is discussed in more detail in the *Teams Manual*.

9.5 Beliefset Callbacks

A beliefset relation will only post an event if the specific event-posting callback has been written. Hence, a beliefset relation will only need to include `#posts event` declarations if such callbacks are going to be written. These callbacks are defined in the beliefset super-classes `OpenWorld` and `ClosedWorld`. Their prototypes are listed below:

- `public void addfact(Tuple t, BeliefState is);`

  This callback is executed whenever an attempt is made to *add* a `Tuple t` with the `is BeliefState` into the agent's beliefset, regardless of whether the tuple is already present.

  The `BeliefState` can either be `Cursor.TRUE` if the tuple is meant to be true, or `Cursor.FALSE` if the tuple is meant to be false (this only applies to `OpenWorld` relations).

- `public void newfact(Tuple t, BeliefState is, BeliefState was);`

  This callback is executed whenever a `Tuple t` with the `BeliefState is` is added to the beliefset.

  The `was BeliefState` is bound to the `BeliefState` that the tuple had in the beliefset previously.

  – If the tuple was not present in any form and the beliefset relation is `ClosedWorld`, `was` will be `Cursor.FALSE`.

  – If the tuple was not present in any form and the beliefset relation is `OpenWorld`, `was` will be `Cursor.UNKNOWN`.

- `public void delfact(Tuple t, BeliefState was);`

  This callback is executed whenever a `Tuple t` with `BeliefState was` is removed from the agent's beliefset.

  The `was BeliefState` can either be `Cursor.TRUE` if the tuple is meant to be true, or `Cursor.FALSE` if the tuple is meant to be false (this only applies to `OpenWorld` relations).

- `public void endfact (Tuple t, BeliefState was, BeliefState is);`

  This callback is executed whenever a `Tuple t` with `BeliefState was` is removed from the agent's beliefset. This removal can take place either through an explicit *remove*, the adding of its negation or through key constraint knockouts.

  The `is BeliefState` is bound to the `BeliefState` that the tuple has after being removed. This may be `Cursor.FALSE` if the tuple was negated or the beliefset relation is `ClosedWorld`, or it may be `Cursor.UNKNOWN` if the tuple is removed completely and the relation is `OpenWorld`.

- `public void modfact (Tuple t, BeliefState is, Tuple knocked, Tuple negated);`

  This callback is executed just before a `Tuple t` gets added to the relation's beliefset and changes its `BeliefState` to `is`. If the change knocks out another tuple due to key constraints, this tuple is assigned to the knocked parameter, and if the change knocks out another tuple due to inconsistency (negation), this tuple is assigned to the negated parameter.

- `public void moddb();`

  This is a catch-all callback. It is called whenever the state of the beliefset changes due to an `add()` or `remove()` method call.

Note: This is called after the change has been made to the beliefset.

When any of these callbacks are included in a beliefset relation's definition, it is imperative to declare any events that the callback posts in `#posts event` declarations.

For example, consider the following beliefset definition:

```
beliefset Politician extends OpenWorld
{
   #key field String name;
   #value field String party;
   #value field String portfolio;

   #indexed query
        getPortfolio(String n,String p,logical String port);

   #indexed query
        getWho(logical String n,String p,String port);

   #posts event electedEvent electref;

   public void newfact(Tuple t,BeliefState is,BeliefState was)
   {
      // Note that Politician__Tuple contains two underscores
      Politician__Tuple pt = (Politician__Tuple) t;

      if (pt.portfolio.equals("Prime Minister"))
      {
        // code to post the elected event. This code will
        // be executed whenever a new prime minister is
        // elected. For example:
        postEvent(electref.newElection(pt.name, pt.party,
              pt.portfolio));
           ...
      }
   }
}
```

In this example, the beliefset relation includes a callback that should be executed whenever a new prime minister is elected. When a new tuple is added to the beliefset whose `portfolio` field is 'Prime Minister' the `newfact()` callback is executed. The `name` member will be bound to the new prime minister's name and the `party` member to the new prime minister's party. It is up to the callback's author to implement how the event should be posted when these circumstances arise. Because an event of type `electedEvent` will be posted, the `#posts event electedEvent electref` declaration is required. The `newfact()` callback method can use this event's `electref` handle to access the event's posting methods.

9.6 Manipulating Beliefset Relations

The beliefset `OpenWorld` and `ClosedWorld` classes provide two base methods for manipulating the tuples in an agent's beliefset. These are `add()`, which is for adding information to the beliefset, and `remove()`, which is for removing information from the beliefset. Each of these methods are provided for both Closed World and Open World relations.

Beliefset cursors also provide a base method called `removeAll()` that can be used to remove a set of tuples from an agent's beliefset relation. When called, this method removes all tuples from the relation that unify with the beliefset cursor's query expression.

For example, suppose an agent uses a private relation called `politician()` (of type `Politician`). Suppose also that the current set of tuples for this relation are as shown in the following table:

Name	Party	Portfolio
Mr Important	Sensible Party	Prime Minister
Ms Action	Sensible Party	Minister for Sport
Mr Knockout	Silly Party	Minister for Sport

Table 9-9: Tuples in the Politician beliefset

If the agent executes the following code, this will remove the first two tuples from the agent's private `politician()` relation.

```
logical String name;
String party;
String portfolio;

party.unify("Sensible Party");
politician.getWho(name, party, portfolio).removeAll();
```

That is, it will remove all tuples that unify with the parameters provided to the relation's `get who` indexed query.

One simple way to clear a private relation for an agent is to use the `removeAll()` method on a beliefset cursor expression with unbound logical members for all query parameters. For example, to completely clear the above agent's politician relation of tuples, regardless of how many it has, an agent could execute the following reasoning method code:

```
logical String name;
logical String party;
logical String portfolio;

politician.who(name, party, portfolio).removeAll();
```

Since every tuple in the relation will unify with this query expression, every tuple will be removed.

9.7 Beliefset Iteration

A JACK beliefset is neither an array nor a list of records. It is structured in a way that permits efficient query-based information retrieval, and this is how it is best used. To access data linearly, it may be more appropriate to use a Java data structure rather than a JACK beliefset. Nevertheless, it is sometimes useful to be able to retrieve all the tuples from a JACK beliefset.

A JACK beliefset query returns a `Cursor` and it is possible to iterate over all the matching tuples using this cursor. For example, suppose we have a plan with:

```
//code that iterates through the beliefset

#reads data BeliefsetType bel;

logical something x, y, z;

for ( Cursor c = bel.get(x,y,z) ; c.next() ; )
{
     // process x,y,z with new bindings each time
}
```

The logic machinery knows of cursors as objects that can provide bindings for logical variables and that they carry information, so that when the `next()` method is called on the cursor, it will renew the bindings, if possible. The first call to `next()` provides the first binding, the second call provides the second binding, and so on, until all alternative bindings have been provided, at which point the `next()` resets bindings to the original input state and returns false.

However, note that if the subsequent processing changes the beliefset, or the beliefset is changed by some other task, the next `c.next()` call will result in a `BeliefSetException`.

Note that the code below does **not** iterate through the beliefset (as one may initially expect), but instead repeatedly performs the same processing while a certain beliefset state holds. This is because there is no call to `next()` on the cursor returned by `bel.get()` to renew the bindings:

```
//code that does NOT iterate through the beliefset

logical something x, y, z;

while ( bel.get(x,y,z) )
{
    // process x,y,z
}
```

9.8 Extending the `OpenWorld` or `ClosedWorld` classes

If you wish to create your own extension of the `OpenWorld` or `ClosedWorld` classes they must be marked as **abstract**.

It will also be necessary to create a *Cursor* class for your extended class. An example is shown below:

```
public abstract class MyOpenWorld
    extends aos.jack.jak.beliefset.OpenWorld {

    // Any constructors or methods
    // you wish to override go here.
}

public abstract class MyOpenWorldCursor
    extends aos.jack.jak.beliefset.OpenWorldCursor {
```

```
        // You need this class whether or not
        // you plan to override anything in it.
    }
```

The JACK compiler will define the required abstract methods for each specific type of beliefset that you define in your JACK application.

10 Views

10.1 Introduction

The *view* concept is central to JACK's data modelling capability, providing the means to integrate a wide range of data sources such as JACK beliefsets, Java data structures and legacy systems into the JACK framework. In performing this role, the `view` type level construct is used in conjunction with JACK query methods (in particular the `#complex query` and the `#function query` introduced in the *Beliefset Relations* chapter), and JACK cursors, which are described in the *Plans* chapter.

10.2 View Definition

A view is defined as a type level construct using the keyword `view`. There are no requirements on base classes or interfaces for the view. (Note that this is different from all other type level constructs in JACK.)

A view definition takes the following form:

```
public view ViewType
{
   // Declarations and Definitions
}
```

Note that a view definition can extend any class and implement any interface.

10.3 View Declarations

A `view` can contain the following declarations:

```
#uses data Type ref
#complex query methodName(parameters) <statements>
#function query ReturnType methodName(parameters) <statements>
```

Each of these declarations is described in the following sections.

`#uses data` *`Type ref`*

This is a declaration that this view requires data of type *`Type`*. *`Type`* can refer to a JACK beliefset or to an arbitrary data type. The reference name *`ref`* is used to refer to the data instance within this view. An actual view is bound to an actual data instance at construction time.

`#complex query` *methodName*`(`*parameters*`)<`*statements*`>`

This enables a complex query called *methodName* to be defined. Complex queries were introduced in the *Beliefset Relations* chapter. However, note that when used in a `view` (as opposed to a `beliefset`), the query can be defined to span more than one JACK beliefset. Note that *methodName* must return a `Cursor`. The parameters for the query can be of any type.

`#function query` *ReturnType methodName*`(`*params*`)<`*statements*`>`

This enables a function query called *methodName* to be defined. Function queries were introduced in the *Beliefset Relations* chapter. They enable queries which use arbitrary Java code to be constructed. Unlike a complex query, a function query can return any type. The parameters for the query can be of any type.

`#posts event` *EventType* `[`*reference*`]`

This statement declares an event that an agent can post from within a view using the `postEvent` method.

10.4 Usage

A `view` does **not** have automatic `add` or `remove` methods. Like beliefsets, views must be declared in the agents and plans that use them. The declaration within the agent takes the following form:

```
#private data DataType ref(arg_list); or
#agent data DataType ref(arg_list); or
#global data DataType ref(arg_list);
```

where *DataType* is the type of data involved, *ref* is the reference used within the agent and *arg_list* is the list of arguments that must be passed to the constructor.

The declaration within a plan takes the following form:

```
#reads data DataType ref; or
#modifies data DataType ref;
```

where *DataType* is the type of data involved and *ref* is the reference to the data.

Views can be used in the same way as JACK beliefsets (bearing in mind the differences noted above) and conversely, JACK beliefsets may include `#complex query` and `#function query` statements.

The following examples illustrate how the `view` construct can be used.

Using a view to form a query spanning multiple beliefsets

This example illustrates how you can use a `view` and a `#complex query` to formulate a query that spans two JACK beliefsets.

In this example, you have two beliefsets, one which records where people live and one which records what credit cards people have. A `view` could then be generated to obtain information about credit cards used by people in Melbourne. The associated definitions could look as follows:

```
//In a file Addresses.bel

public beliefset Addresses extends OpenWorld
{
    #key field PersonID who;
    #value field Address where;

    #indexed query get( PersonID p , logical Address a );
    #indexed query get( logicall PersonID p , Address a );
}

//In a file CreditCards.bel

public beliefset CreditCards extends OpenWorld
{
    #key field PersonID who;
    #key field CreditCard what;

    #indexed query get
            (logical PersonID p, logical CreditCard c );
}

//In a file MelbourneCards.view

public view MelbourneCards
{
    #uses data Addresses bel_where;
    #uses data CreditCards bel_card;

    #complex query get( logical CreditCard card )
    {
        logical PersonID who;

        return bel_where.get( who , new Address("Melbourne") ) &&
                bel_card.get( who , card );
    }
}
```

The view, `MelbourneCards`, is now available for use like a JACK beliefset. It is a class that has a constructor taking two arguments, which correspond to the data it uses in the order they are declared. However, the view does not have any automatic `add` or `remove` methods.

It could be used in the following way:

```
    //In a file CleverAgent.agent

    agent CleverAgent extends Agent
    {
            ...

        #private data Addresses where();
        #private data CreditCards cards();
        #private data MelbourneCards
                cards_in_melbourne( where , cards );

            ...
    }

    //In a file InterestingPlan.plan
    plan InterestingPlan extends Plan
    {
            ...

        #reads data MelbourneCards cards_in_melbourne;

            ...

    body()
    {
        logical CreditCard card ;

        // if nothing in beliefset, wait for it... (just an example)
        @wait_for( cards_in_melbourne.get( card ) && ... );
    }
}
```

Using a view to integrate an external process into JACK

The following example illustrates how a view can be used to integrate an external process into JACK. In this example, there is an assembly cell which is controlled by a program called BBS. Each machine in the cell has a designated input address and output address. The input address provides access to the machine's status word; the output address provides access to the machine's control word. BBS accepts requests on a UDP socket to either read the contents of the input address or to set the contents of the output address.

Control of a machine by an external agent is achieved as follows:

- The agent waits until the *idle* bit in the status word maintained by BBS is true. The agent then sets the *program* bits in the control word to the desired values and the *go* bit in the control word to true to start the desired operation.
- When the operation starts, the actual software controlling the machine sets the *idle* bit in the status word maintained by BBS to false and then the *go* bit in the control word to false. When the operation is complete, it sets the *idle* bit to true.

If the interface is wrapped in a view, it is possible to encapsulate the low level socket interface into a higher level functional interface consisting of the following queries:

`outputIdle(boolean value)` – this sets/clears the idle bit.

`inputIdle(int rate,boolean value)` – this polls the idle bit every `rate` milliseconds.

A plan which implements the control loop specified above can then be implemented as follows:

```
plan DefaultProgramRun extends Plan
{
    #handles event RunProgram ev;
    static boolean relevant(RunProgram ev)
    {
        return ev.program >=0 && ev.program <= 15;
    }

    #uses data BBSConnection bbs;

    body()
    {
        // uses bits 4,3,2,1 for program number
        bbs.output = ev.program << 1 ;

        // Tell the desired program number + go ahead bit
        @wait_for( bbs.outputIdle( true ) );

        // Wait for the operation to start -- hopefully it hasn't
        // completed already (race condition)
        @wait_for( bbs.inputIdle( 100, false ) );

        // Turn the go bit off (the PLC should do this ... )
        @wait_for( bbs.outputIdle( false ) );

        // Wait for the operation to complete
        @wait_for( bbs.inputIdle( 500, true ) );
    }
}
```

The implementation for the view (`BBSConnection`) is given below. Note the use of `RepeatAction` and `Again` cursors to implement polling. Polling is a strategy which should be avoided whenever possible as it can be extremely wasteful of CPU time. In this situation, there was no alternative, but inspection of the code reveals that:

- the read action is performed on a separate thread.
- the action is performed **only** every `rate` milliseconds. In between times the read action thread is in a *waiting* state.

Consequently, the impact of polling on the rest of the system is minimised.

```
import aos.jack.util.cursor.Action;
import aos.jack.util.cursor.RepeatAction;
import aos.jack.util.cursor.Again;
import aos.util.ThreadPool;
import java.net.DatagramSocket;
import java.net.DatagramPacket;
import java.net.InetAddress;
import java.io.IOException;
import java.net.UnknownHostException;

public view BBSConnection
{
    static String bbs_host = null;
    static InetAddress bbs_ip;
    static int bbs_port;
    static ThreadPool thread_pool = new ThreadPool(1);
    static DatagramSocket socket;

    public static void setBBS(String host,int port)
    throws UnknownHostException
    {
        if ( bbs_host == null )
        {
            bbs_host = host;
            bbs_ip = InetAddress.getByName( bbs_host );
            bbs_port = port;
            try
            {
                socket = new DatagramSocket();
            }
            catch (Exception e)
            {
                e.printStackTrace();
                System.exit( 1 );
            }
        }
    }

    int input_address = -1;
    int output_address = -1;

    public int input = 0;
    public int output = 0;

    public void setAddresses(int in,int out)
    {
        if ( input_address == -1 )
        {
            input_address = in;
            output_address = out;
        }
    }

    public void write(int value) throws IOException
    {
        sendBBS( "IRW, " + output_address + ", " + value );
    }
```

```
private void sendBBS(String s) throws IOException
{
    byte [] d = s.getBytes();
    DatagramPacket p =
        new DatagramPacket(d, d.length, bbs_ip, bbs_port);
    socket.send( p );
}

private final static int SIZE = 4096;

public int read() throws IOException
{
    try
    {
        sendBBS( "IRR, " + input_address );
        byte[] b = new byte[ SIZE ];
        DatagramPacket p = new DatagramPacket( b, SIZE );
        socket.receive( p );
        String s = new String( b, 0, p.getLength() );
        int ix = s.lastIndexOf( ' ' );
        return Integer.parseInt( s.substring( ix + 1 ) );
    }
    catch (NumberFormatException e)
    {
        e.printStackTrace();
        return 0;
    }
}

public Action writeAction(int value)
{
    return new Write( value );
}

public Action readAction()
{
    return new Read();
}

class Write extends Action
{
    int value;

    Write(int v)
    {
        value = v;
    }

    protected void action()
    {
        try
        {
            write( value );
        }
        catch (IOException e)
        {
            // Discarding this as too unlikely to happen.
        }
    }
}
```

```
    class Read extends RepeatAction
    {
        protected void action()
        {
            try
            {
                input = read();
            }
            catch (IOException e)
            {
                // Discarding this as too unlikely to happen.
            }
        }
    }

    // inputIdle(long,boolean) is a support query to repeatedly
    // review the input idle bit (bit 0) for a set or reset
    // value.
    //
    #complex query inputIdle(long rate,boolean set)
    {
        return new Again( rate ) && readAction() && isSet( set );
    }

    boolean isSet(boolean set)
    {
        int bit = input & 1;
        return ( bit != 0 )? set : !set ;
    }

    // outputIdle(boolean) is a support query to set or reset the
    // output idle bit (bit 0).
    //
    #complex query outputIdle(boolean set)
    {
        output = ( output & -2 ) + ( set? 1 : 0 );
        return writeAction( output );
    }
}
```

Appendix A: JackBuild

Description

The `JackBuild` utility is a Java program that invokes the JACK compiler and the Java compiler appropriately so as to rebuild outdated files after editing. The utility accepts arguments from the command line, from which it prepares command lines for first running the JACK compiler (if needed for any JACK files), and thereafter the Java compiler, as needed for any Java files. The command line is as follows:

```
java aos.main.JackBuild flags sources
```

The `flags` are used to define operating parameters for the `JackBuild` utility, while the `sources` includes files and directories to be visited by the compiler. If a directory is mentioned, the utility will operate on all files it recognises in that directory.

Extension	Contents	Produces	Produced By
x`.class`	Java class file	-	
x`.java`	Java source file	*x*`.class`	Java compiler
x`.agent`	JACK agent source file	*x*`.java`	JACK compiler
x`.plan`	JACK plan source file	*x*`.java`	JACK compiler
x`.event`	JACK event source file	*x*`.java`	JACK compiler
x`.bel`	JACK beliefset source file	*x*`.java`	JACK compiler
x`.cap`	JACK capability source file	*x*`.java`	JACK compiler
x`.view`	JACK view source file	*x*`.java`	JACK compiler
x`.api`	JACOB DDL file	*x*`.java`	JACOB compiler

Table A-1: File name extensions

Files are recognised by their file name extensions according to the table above, which also shows the file dependencies supported by the utility. In addition, the compiler also recognises a `.jack` extension. This can be used if it is desirable to keep all JACK source files in `*.jack` rather than `*.agent`, `*.plan` etc.

The flags for the `JackBuild` utility are as follows:

Flag	Meaning
`-h, -help`	Provide a list of options and exit immediately.
`-hj`	Provide a complete list of compiler options and exit immediately.
`-v, -version`	Provides details about the version of JACK.
`-a`	Visits all files/directories (including those that start with `.`)
`-n`	Checks only; no compiled files will be generated.
`-r`	Recursively enters subdirectories looking for code to compile.
`-d <dir>`	Specifies where to put generated `.java` and `.class` files.
`-dc <dir>`	Specifies where to put generated `.class` files. This overrides the -d option for the class files.
`-c, -clean`	Clean up by removing generated files.
`-q`	Queries only; outputs a list of the source files detected.
`-x`	Recognises JACOB files (extension `.api` and `Init__*.java`).
`-E[xxx.prj]`	Creates Jack Development Environment files from a set of JACK source files. Optionally creates a project file (`xxx.prj`) if a name is specified.
`-P <pkg>`	Sets the top level package name which will be stripped from the -E generated JACK Development Environment files.
`-f`	Forces compilation of as many files as possible, even if errors are found in sources.
`-cp <CLASSPATH>, -classpath <CLASSPATH>`	Set the `CLASSPATH`.
`-wd <dir>`	Searches the given directory, rather than the current directory.
`-i`	Tells the compiler to report errors relative to the internally generated code fragments rather than the original JACK code.
`-Dxxx`	Property is set and passed on to each JACK compilation command.

Table A-2: Flags for the `JackBuild` utility

Note that the `-r` flag tells the utility to operate recursively, which means that it will traverse any directories mentioned on the command line exhaustively and include all recognisable files in the operation. By default, files and directories beginning with a dot are not looked at or traversed. The `-a` flag can be used to override this.

Dependent files are generated by the respective compilers, and will end up in the same directories as their respective sources.

If no sources are mentioned on the command line, the utility will operate on all recognisable files in its current working directory.

If a source is missing, or rather is neither a file nor directory, the `make` utility will abort and leave the file system unaffected.

If the `-n` flag is given, the utility will merely analyse and report on what is outdated without affecting the file system.

The `JackBuild` utility operates as follows:

1. Go through all files concerned and remove dependent files.
2. If `-x` was given, invoke JACOB compiler on updated JACOB DDL files (`*.api`).
3. Go through all files (and directories) that were listed on the command line and collect updated JACK and java files. If none were listed on the command line the current directory is assumed. This is a recursive process if `-r` was given.
4. Invoke the JACK compiler on all updated files.
5. Collect all changed Java (`*.java`) files (in the relevant directories). Most of these will have been created by the JACK compiler.
6. Invoke `javac` on all updated Java files.

The `-DJAVACARGS=xxx` argument can be used to add extra arguments to any `javac` command invoked by the JACK compiler.

The `-DJAVAC=xxx` argument can be used to force the JACK compiler to invoke a different `javac` command, for example `jikes`.

The `-DJACOBARGS=xxx` argument can be used to add extra arguments to any `JacobBuild` command invoked by the JACK compiler.

Appendix B: Utility Classes

Introduction

The JACK distribution includes a number of utility packages. Most of the classes in these packages are intended for internal use but some are of more general applicability. The purpose of this appendix is to document briefly those utility classes which may be of use to JACK developers. The utility packages that are included in the JACK distribution are summarised below:

Package	Contents	Comments
`aos.util`	Convenience classes.	Most are intended for internal use; `PathEntry`, `Properties`, `Redirector` and `ThreadPool` are described below.
`aos.util.timer`	Timer classes.	Described in the manual in the *Time Cursors* section.
`aos.jack.util`	Convenience classes.	Intended for JACK internal use.
`aos.jack.util.cursor`	Cursor classes.	Described in the manual in the *Cursors* section.
`aos.jack.util.thread`	Synchronisation classes.	Described below.
`aos.jack.jak.util`	Convenience classes.	Intended for JACK internal use.

Table B-1: Utility packages included in the JACK distribution

Note that in the descriptions that follow, it is assumed the user is familiar with particular Java classes and the basic concepts of synchronisation in a multi-threaded environment.

`aos.util.PathEntry`

`PathEntry` provides a capability to open files and to load objects that are in one's `CLASSPATH`. To open a file, use:

```
static InputStream open(String filename)
```

To load an object, use either:

```
static byte[] loadObject(String filename)
```

or

```
static byte[] loadObject(InputStream is)
```

as appropriate.

aos.util.Properties

The `Properties` class provides an alternative interface to the system properties list. In particular, the loading of user defined properties is simplified and properties are returned as their intended type rather than as a `String` (which must then be converted to the desired type). The loading of user defined properties is achieved with the following method:

```
synchronized static final void readProperties(String list)
```

`list` can be either a UNIX filename or a URL.

The following methods are available for accessing properties:

```
static final int getIntProperty(String propertyName,int defaultValue)

static final long getLongProperty(String propertyName,long defaultValue)

static final double getDoubleProperty(String propertyName,double defaultValue)

static final boolean getBooleanProperty(String propertyName,boolean
defaultValue)

static final String getStringProperty(String propertyName,String defaultValue)

static final int getBitMapProperty(String propertyName,int defaultValue,String[]
bitNames)
```

The behaviour of the above methods is self-explanatory, with the exception of `getBitMapProperty()`. The value stored with the property can either be an integer or a sequence of bit position names separated by colons. If the name is preceded by !, the value of that position is set to 0, otherwise it is set to 1. The position is determined from the index of the bit position name in the `bitNames` array.

Note: The `jack.run.debug.options` property (discussed in the *Introduction*chapter) is specified as such a bitmap.

---

aos.util.Redirector

The `Redirector` class allows the user to dynamically redirect the standard input, standard output and standard error for a process. A single method

```
static void doRedirection()
```

is provided for this purpose; if any of the properties `debug.setInput`, `debug.setOutput` or `debug.setError` are set in the system properties list, the appropriate redirection will occur. The association between one of the standard streams and the file to be used for redirection can be achieved using either a properties file and `aos.util.Properties.readProperties()`, by using `System.setProperty()` or by using the `-D` option on the command line for the application.

aos.util.ThreadPool

A `ThreadPool` is an object that manages a set of threads that are used to process a queue of `Runnables`. The number of threads in the pool varies between a lower and an upper limit, which are specified at construction time. Three constructors are provided:

`ThreadPool()` The lower limit and upper limit are both set to 1 – i.e. the thread pool contains exactly 1 thread.

`ThreadPool(int i)` The lower limit and upper limit are both set to `i` – i.e. the thread pool contains exactly `i` threads.

`ThreadPool(int i,int j)` The lower limit is set to `i` and the upper limit is set to `j`.

In order to execute a `Runnable` object, the method

```
synchronized void run(Runnable r)
```

must be invoked by the programmer. This method adds `r` to the queue of `Runnables` which is being served by the threadpool.

aos.jack.util.thread.Semaphore

A `Semaphore` is a synchronisation resource which is used to establish mutual exclusion regions of processing in JACK plans and threads. A `Semaphore` is a binary resource which plans and threads may wait for and signal on when they have completed. Waiting entities queue up on the semaphore and acquire the semaphore in FIFO order.

The `Semaphore` class has a single constructor:

```
Semaphore()
```

The semaphore is grabbed initially by the constructing thread (or plan), and must thus be released by a call to `signal()`.

Methods are provided to grab and release the semaphore:

```
Cursor planWait()
```

`planWait()` returns a `Cursor` object to grab the semaphore. This method is called by JACK plans, which then use `@wait_for` to synchronise.

```
void threadWait()
```

`threadWait()` waits (via `Object.wait()`) to grab the semaphore. This method is called by Java threads (not JACK plans) to synchronise on the semaphore.

```
void signal()
```

`signal()` releases the semaphore to the next waiting thread or plan.

A program illustrating the use of semaphores in a JACK application is provided in the solutions to the practical exercises. This code can be found in the `doc/practicals/solutions/practical1/semaphore` directory of the JACK distribution tree.

aos.jack.util.thread.TaskJunction

A `task junction` is a synchronisation resource for plans. It allows a monitoring plan to wait until a group of plans are complete. The `TaskJunction` class has a single constructor:

```
TaskJunction()
```

The task junction which is constructed is initially idle: that is, there are no executing plans associated with it.

The `TaskJunction` class provides the following methods which enable a plan to attach and detach from a task junction, and to determine when a task junction becomes idle:

```
void join()
```

This method is called by a plan in order to join the task junction.

```
void leave()
```

This method is called by a plan in order to leave the task junction.

```
Cursor idle()
```

This method returns a `Cursor` object to observe when the task junction becomes idle.

It is sometimes useful for a joined plan to leave a task junction and then to rejoin at some later date when a particular condition is satisfied. For example, in a simulation one might use a task junction to keep track of all plans which are initiated within a given timestep – when the task junction becomes idle, we then move to the next timestep. However, it may well be that not all plans are able complete their execution within a single timestep, in which case we want the plan to leave the task junction and to then rejoin it when the plan is about to complete execution. This functionality is proved by the following method:

```
Cursor escape()
```

`escape()` returns a `Cursor` object that lets a joined plan escape the task junction while in an `@wait_for` statement. The `escape()` cursor should be used in conjunction with the actual condition waited for, and the Cursor ensures that the plan rejoins the task junction during the plan step which makes the actual condition come true.

```
Cursor escape(Cursor c)
```

In this variant of `escape()` a joined plan escapes the task junction only until the argument cursor (a triggered cursor) becomes true. This variant is safe with respect to external thread triggering; the task junction is re-joined immediately when the cursor triggers.

aos.jack.util.thread.Monitor

`Monitor` is a convenience class for allowing event handling to be monitored via a primary task junction. The Monitor class goes together with the `TaskMonitoring` interface. They support task execution reflection so that an agent can know whether it is busy or idle with respect to monitored events.

The following constructors are provided:

```
Monitor(Event e,String tj)

Monitor(Event e,Agent a,String tj)

Monitor(Event e,TaskJunction tj)

Monitor(Event e)
```

aos.jack.util.thread.TaskMonitoring

This interface is to be implemented by an agent in order to provide access to a primary task junction for `Monitor` objects. It consists of the following method:

```
TaskJunction getTaskMonitor()
```

Appendix C: JACK Properties

A number of properties are provided for customisation of the runtime behaviour of JACK tools and applications. Developers are of course free to provide their own application specific properties if required.

This appendix lists the effect of usage, possible values and default setting of each publicly available JACK compiler and runtime environment property. A complete list of the publicly available JACK properties, including properties used to customise tracing and debugging tools can be found in the *Tracing and Logging Manual*.

Several JACK properties are accessible from the JDE Preferences window. Refer to the *Development Environment Manual* for instructions on how to set these properties. JACK properties can also be used when running a JACK application. In this case, the property name must be preceded by a `-D` and entered either on the command line or in the Java Args field in the Run Application tab of the of the JDE Compiler Utility.

JACK Compiler Properties

Property	Description	Type	Default
`jack.compiler.emit.imports`	Generates full package paths to class names in Java code instead of import statements.	`boolean`	`false`
`jack.compiler.errors`	Specifies the maximum number of errors to be displayed by JackBuild.	`int`	10

Table C-1: JACK Compiler Properties

JACK Runtime Environment Properties

Property	Description	Type	Default
`jack.args`	Enables the specified value to be used as if it was passed from the command line to the application.	`String`	`null`
`jack.portal.name`	Specifies the name of the portal for the application.	`String`	`"%portal"`
`jack.portal.host`	Specifies the host of the portal for the application.	`String`	`"local host"`
`jack.portal.port`	Specifies the port number of the portal for the application.	`int`	Next available port number.
`jack.property.file`	Specifies the name of a file that contains JACK property settings.	`String`	`null`
`jack.run.nthreads`	Specifies the number of JACK threads to be made available to the scheduler. Note that having more JACK threads than CPUs on a machine does not benefit performance.	`int`	1
`jack.run.timeslice`	Specifies how many milliseconds are to be allocated to an agent on a JACK thread before the scheduler should intervene.	`int`	100
`jack.run.repeatable`	Equivalent to setting `jack.run.timeslice` to 1 hour. This will stop agent tasks from being suspended and restarted at arbitrary points.	`boolean`	`false`

Table C-2: JACK Runtime Environment Properties

Index