# JACK Intelligent Agents®
# Tracing and Logging 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 . . . . . 9**

**2 Design tracing tool . . . . . 11**

2.1 Introduction. . . . . 11

2.2 Overview . . . . . 11

2.3 Initialisation. . . . . 12

2.3.1 Preliminaries. . . . . 12

2.3.2 Design tracing and graphical plan tracing . . . . . 12

2.3.3 The Application. . . . . 12

2.3.4 Starting the DTT. . . . . 13

Tracing from the command line . . . . . 13

Tracing from the JDE . . . . . 14

2.3.5 The Trace menu . . . . . 15

2.3.6 The Tracing window . . . . . 17

2.4 Configuring design tracing . . . . . 17

2.4.1 Adding a trace row . . . . . 18

2.4.2 Removing a trace row. . . . . 18

2.4.3 Editing a trace row . . . . . 18

2.4.4 Selecting a design . . . . . 19

2.4.5 Tracing agent types . . . . . 19

2.4.6 Tracing individual agents . . . . . 20

2.4.7 Tracing all agents . . . . . 20

2.4.8 Saving and loading tracing configurations . . . . . 20

Saving a design tracing configuration. . . . . 20

Loading a design tracing configuration. . . . . 20

Design tracing configuration files . . . . . 21

2.4.9 Applying tracing settings . . . . . 21

Configuration errors . . . . . 21

2.5 Controlling design tracing. . . . . 21

2.5.1 Turning tracing on and off. . . . . 22

2.5.2 Tracing in descriptive mode . . . . . 22

2.5.3 Delaying tracing transitions . . . . . 22

2.5.4 Controlling tracing . . . . . 23

Starting tracing . . . . . 23

Stopping tracing . . . . . 23

Controlling tracing of individual traced design windows . . . . . 23

Stepping through tracing . . . . . 23

2.5.5 Reconfiguring tracing during execution . . . . . 24

2.6 Tracing visualisation . . . . . 24

2.6.1 Traced design window control bar . . . . . 26

2.6.2 Traced design window tool bar . . . . . 26
2.6.3 Viewing design documentation . . . . . 26
2.6.4 Viewing a design graph . . . . . 26
2.6.5 Viewing relevant agents . . . . . 26
2.6.6 Viewing relevant tasks . . . . . 27
2.6.7 Transition visualisation . . . . . 27
2.6.8 Resetting transition counts . . . . . 27
2.6.9 Tracing visualisation errors . . . . . 27
No traced design windows are shown . . . . . 27
2.7 A design tracing example . . . . . 27
**3 Plan tracing tool . . . . . 37**
3.1 Introduction . . . . . 37
3.2 An plan tracing example . . . . . 38
3.2.1 The multi-currency bank account example . . . . . 38
3.2.2 Walkthrough . . . . . 40
The plan tracing tool windows . . . . . 41
A sample run of the plan tracing tool . . . . . 44
Finishing the sample run . . . . . 52
Tracing more than one agent . . . . . 52
3.2.3 Running without the JDE . . . . . 52
3.3 Plan tracing tool configuration . . . . . 53
3.3.1 Plan tracing configuration files . . . . . 53
Default file generation . . . . . 54
Running with no tracing configuration file . . . . . 54
3.3.2 Runtime options . . . . . 55
**4 Agent Interaction Diagram . . . . . *59***
4.1 Introduction . . . . . 59
4.2 Enabling an Interaction Diagram . . . . . 59
4.3 Configuring an Interaction Diagram . . . . . *62*
**5 Audit Logging . . . . . 69**
**6 Generic Debugging/Agent Debugging . . . . . 71**
6.1 Using debugging . . . . . 71
6.2 The `AgentDebuggerCommand` interface . . . . . 72
6.3 Debug objects . . . . . 72
6.3.1 `DumpState` . . . . . 73
6.3.2 User defined debug objects . . . . . 73
6.4 Running debugging . . . . . 74
**Appendix A: JACK Properties . . . . . 75**
JACK Compiler Properties . . . . . 76
JACK Runtime Environment Properties . . . . . 76

JACK Debugging Properties . . . . . 78
Design Tracing Tool Properties . . . . . 78
Agent Interaction Diagram Properties . . . . . 79
Plan Tracing Tool Properties . . . . . 80
**Index . . . . . 83**

# 1 Introduction

JACK applications invariably involve multiple agents interacting asynchronously with each other and with their environment. Thus although the specification of individual behaviours may be straightforward, the system behaviour that is observed at execution time can be extremely complex and difficult to interpret. The benefits of using graphical visualisation to display distributed system behaviour has long been recognised and visualisation tools have been provided with JACK since its first release. Visualisation is not only of benefit to the end users of an application, but also to the developers and to Subject Matter Experts (SMEs).

From a developer's perspective, appropriate visualisation of system behaviour can greatly assist in the debugging of an application. The role of the SME in the development of an agent application is to provide agents with domain-specific behaviours. The JDE provides tools for graphical specification of agent behaviours which have proven to be well-suited for use by SMEs who are non-programmers. However, verification of these behaviours can be difficult for the reasons mentioned above. Visualisation of agent behaviours during system execution has proven to be of great assistance in the verification process.

The people involved in an application development will require different visualisations of behaviour depending on the nature of the application and their role in its development. Consequently JACK provides a selection of visualisation tools that focus on particular aspects of system behaviour:

| Tool | Purpose | Intended Users |
|---|---|---|
| Design Tracing Tool | Graphically trace the project design diagrams of executing JACK applications | Application designers, SMEs and developers |
| Graphical Plan Tracing Tool | Graphically trace the events and graphical plans of executing JACK applications | Application designers, SMEs and developers |
| Agent Interaction Diagram | Graphically trace inter-agent communication of executing JACK applications | Application developers and users |

**Table 1-1:** Visualisation tools that focus on system behaviour

While the tools listed previously are extremely useful in helping a developer to debug an application, situations occasionally arise when a more detailed trace of system execution is required. For these situations, JACK provides the following logging tools:

| Tool | Purpose | Intended Users |
|---|---|---|
| Audit logging | Trace messages and events in JACK applications | Application developers |
| Agent Debugger | Extract and customise information about agents in executing JACK applications | Application developers |

**Table 1-2:** Logging tools

The remainder of this manual discusses each of the behaviour visualisation and logging tools in detail.

# 2 Design tracing tool

## 2.1 Introduction

The JACK Intelligent Agents® *Design Tracing Tool* (DTT) provides developers with the ability to view the internal details of a JACK application during execution. Execution of applications is shown by highlighting the links in design diagrams that correspond to transitions between JACK elements. The DTT can be used to view different aspects of executing applications by tracing a combination of all agents, selected agent types and individual agents in project design diagrams.

The DTT runs within the JACK™ Development Environment (JDE) and requires connection to a portal associated with the traced JACK application. Once tracing settings have been configured and the application to be traced is running, tracing visualisation can be run, stepped or paused.

Designs traced in the DTT are shown in *traced design windows* (TDWs). TDWs can be shown in descriptive mode, and tracing can be viewed more slowly by delaying transitions between design elements. During application execution, the DTT displays the number of previous executions of transitions between design entities. The agents and tasks relevant to a TDW can also be viewed.

The DTT can be configured to trace a combination of all executing agents, agent types and individual agents in team and non-team projects. Any design element in a design diagram apart from a `Note` and a `Named Role` is traceable.

The intended users of the DTT are application designers, subject matter experts (SMEs) and agent system developers. Application designers can use the tool to verify system behaviour. SMEs can use the DTT to demonstrate running JACK applications. In particular, the descriptive mode of the DTT can be used to show a running description of application execution. Application designers can use the DTT as an agent behaviour debugging tool.

This chapter assumes that the user is familiar with the JACK™ Development Environment (JDE) and JACK™ agents. If further information is required on these topics, refer to the *Development Environment Manual* and the *Agent Manual*.

## 2.2 Overview

This chapter introduces concepts and terms associated with the DTT. Setup of the DTT, including configuration of design tracing, is described in detail. Design tracing control and different options for tracing visualisation are also described. The chapter concludes with a worked example of running and stepping through design tracing of an application.

# 2.3 Initialisation

## 2.3.1 Preliminaries

The DTT runs within the JDE. An application that is examined with the tool runs in a separate process. The DTT uses JACK's interprocess communications layer (DCI) to pass data between the application and the tool. Consequently, before the user can examine the application using the DTT a DCI connection needs to be established. Also, since the tool connects to a running application, the programmer must incorporate a pause mechanism into the application. This ensures that execution of the application is halted at an appropriate point until the DTT has connected to the application and is in control of execution.

## 2.3.2 Design tracing and graphical plan tracing

Design tracing and graphical plan tracing are mutually exclusive and must be used separately. There are several differences in the requirements for setting up and running both types of tracing. If both types of tracing are attempted at the same time, the behaviour of the JDE is undefined.

For further information on tracing graphical plans see the *Plan tracing tool* chapter of this manual.

## 2.3.3 The Application

To trace the agents in an application, it is necessary to connect the JDE to a portal created by the application.

When running an application a portal with the name `%portal` is created by default, with the host `localhost` and the next available port. An application can also be created with a non-default portal with the use of the DCI command line arguments or by setting the following Java portal properties:

- `jack.portal.name`
- `jack.portal.host`
- `jack.portal.port`

From a programming viewpoint, the only modifications that may need to be made to an existing JACK application for it to be used with the DTT:

- If DCI command line arguments are used, a call to `aos.jack.Kernel.init(String[] args)` must be included in the application's `main()` method in order to process the arguments that will be provided when the application is started
- If DCI command line arguments are not used, the application is run with the Java portal properties: `jack.portal.name`, `jack.portal.host` and `jack.portal.port`
- Code is to be provided to suspend execution while the JDE is connected to the portal created by the application. Code to resume execution when the user is ready to take control of execution using the DTT must also be included. A simple way of achieving this is to insert a user prompt in the `main()` method after the agents required by the application have been created. When the prompt is displayed, the user would then connect the JDE to the application, pass control of the application to the tool and then respond to the prompt and resume execution of the application.

## 2.3.4 Starting the DTT

A traced application must be launched with the following Java argument:

```
-Djack.tracing.enabled=true
```

A traced application can be started from the command line or from the Compiler Utility of the JDE.

### 2.3.4.1 Tracing from the command line

If a traced application is launched from a command line, the portal used by the JDE to connect to it must be specified with the Java portal properties specified in the previous section. Design tracing is displayed within the JDE. Once an application has started, the JDE must be launched and connected to the portal created by the application (see the Trace menu section for further details).

The following is an example of the command used to start a traced application named `Program` from the command line:

```
java -Djack.tracing.enabled=true -Djack.portal.name=Server
-Djack.portal.host=localhost -Djack.portal.port=9999 Program
```

### 2.3.4.2 Tracing from the JDE

When a traced application is started from within the JDE, it is launched from the Run tab of the Compiler Utility. A portal can be created by the application either with Java portal properties or DCI command line arguments. The following figure shows an application named `Program` run with DCI command line arguments.

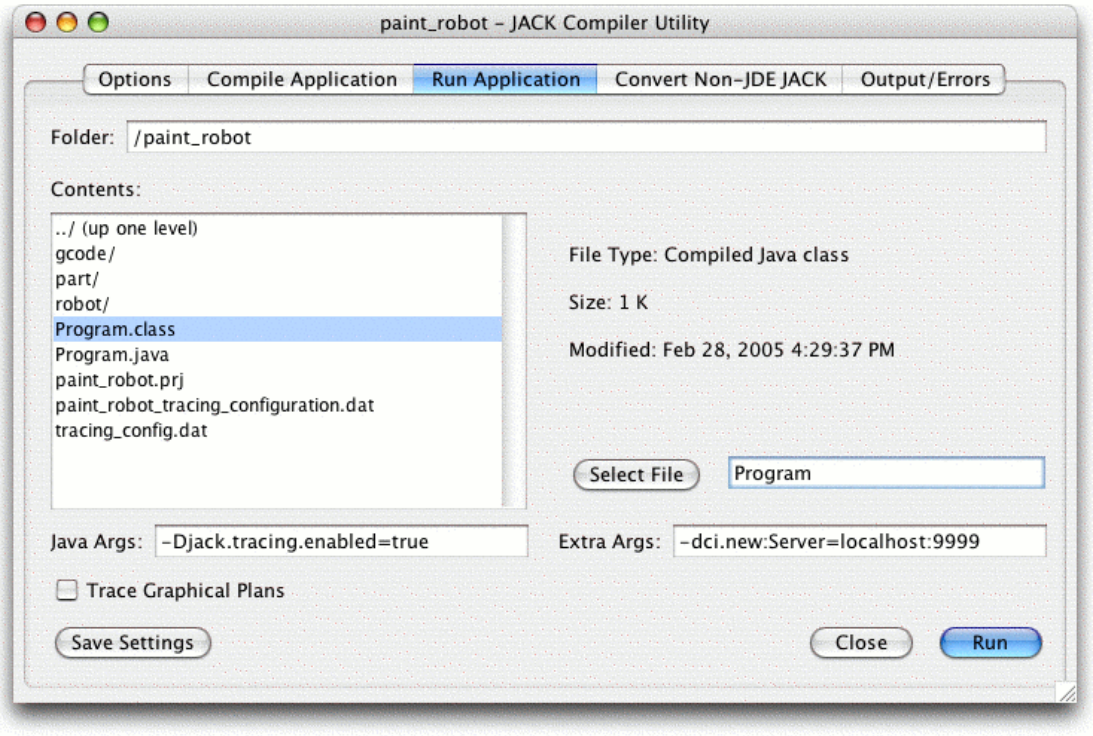


**Figure 2-1:** Starting tracing from the JDE

For more information on DCI, refer to the *Agent Manual*. The application will now execute until its suspension point is reached.

## 2.3.5 The Trace menu

Once the application has been started, the DTT can be launched from within the JDE.

The JDE provides access to DTT functionality via the Trace menu. This menu has 5 choices:

1. Connect To Nameserver...

   Use this menu option to connect to a DCI nameserver. For example, if the application had been started using the DCI arguments described in the previous section, the user would enter `localhost:9999` or `9999` to connect to the application's nameserver. Nameserver connection is optional; however, without a nameserver, the DTT user needs to know explicit portal addresses in order to connect the DTT to each portal, rather than using portal names.

Note that the nameserver address can also be entered using the Connect To Portal... menu option.

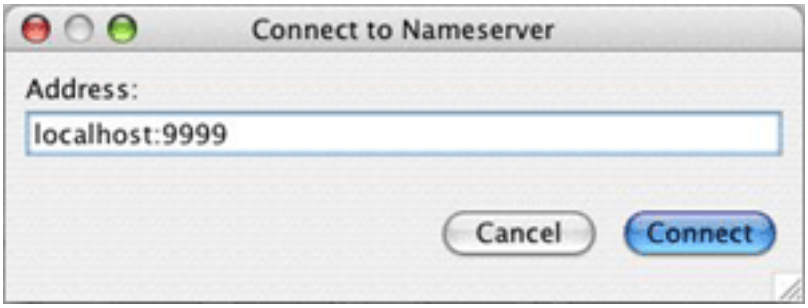


**Figure 2-2:** Connect to Nameserver dialog

2. Connect To Portal...

   Use this menu option to connect to a portal via the portal name or address. Once connected, the Tracing window will be displayed, showing all agents connected to that portal.

   If already connected to a nameserver, you would normally connect by specifying a portal name. Otherwise, use an explicit address such as `localhost:9999` or `9999`.


**Figure 2-3:** Connect to Portal dialog

3. Configure Design Tracing...

   Use this menu option to configure settings for design tracing. Design tracing should be configured whilst an application is stopped or paused. Refer to the *Configuring design tracing* section in this chapter for information on how to configure design tracing.

Once connected to a portal and design tracing is configured, an application can be resumed from its suspension point and traced with the DTT.

4. Design Tracing Controller

   Use this menu option to control tracing of designs. Refer to the *Controlling Design Tracing* section in this chapter for information on how to control design tracing.

5. Ping Agent...

   Use this menu option to check if an agent is still responding. The number of attempts and the delay in seconds can be set between ping attempts. The agent name and portal name need to be entered, e.g. `robot23@Server`.

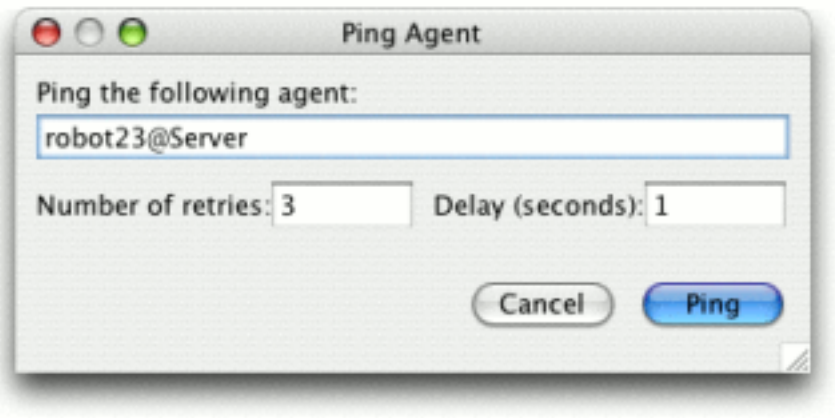


**Figure 2-4:** Ping Agent dialog

### 2.3.6 The Tracing window

Once connected to a portal the Tracing window opens. The tracing window displays the name of the portal and all agents in a running application connected to the portal. Tracing of an application may be stopped by right clicking on any agent in the Tracing window and selecting the Quit tracing on portal option, or by closing the Tracing window.

## 2.4 Configuring design tracing

The Design Tracing Configuration window enables design tracing settings to be added, removed and edited. The design tracing settings configured in the window can also be saved and loaded. Design tracing settings are usually configured before starting an application. These settings can also be configured while an application is paused.

The Design Tracing Configuration window is opened from the Trace menu of the JDE. The window contains a table with a row for each design tracing setting and a columncorresponding to each trace configuration field.

The Design Tracing Configuration window can contain any combination of rows. This allows tracing of all agents, specific agent types and/or individual agents in a design diagram. Each row is usually shown as a separate TDW during application execution. However if two rows with the same name specify the same design diagram they are shown in one TDW.

The following figure shows the Design Tracing Configuration window configured with rows for three TDWs. The first row traces all agents in the `Paint_requesting_capability` design. The second row traces the agent named `robot23` and the third row traces all `Part` type agents.

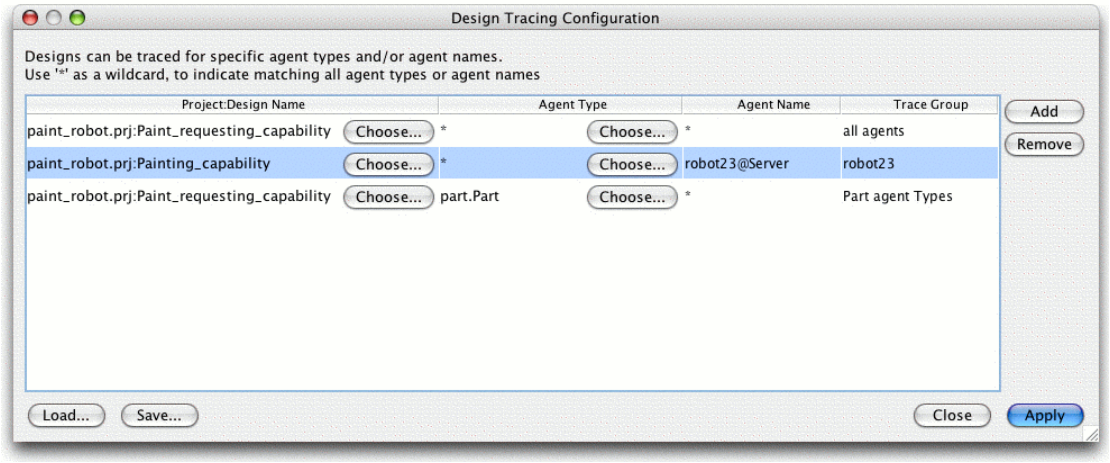


**Figure 2-5:** The Design Tracing Configuration window

## 2.4.1 Adding a trace row

A new trace row is added by clicking the Add button of the Design Tracing Configuration window. The new trace row can then be edited to trace all agents, all agents of a specified type or an individual agent in a design diagram. The new trace row can also be edited to trace a different aspect of an existing row. Note that it must be given the same name as the existing row.

## 2.4.2 Removing a trace row

A trace row is removed by clicking and highlighting the trace row in the Design Tracing Configuration window and clicking the Remove button.

## 2.4.3 Editing a trace row

A trace row must specify a project and design diagram. The Agent Type and Agent Name fields are optional and are filled with asterisks (*) by default, which means all agents will be traced. Trace rows can be edited as required (if a traced application is already running it should be paused first).

## 2.4.4 Selecting a design

A design diagram is selected by typing the project path, file name and design name or by choosing the project and design graphically.

To choose a design diagram manually, click the Project:Design Name column of the Design Tracing Configuration window and type the project name and design name in the following format:

```
file_path/project_name.prj:design_name
```

To choose a design diagram graphically, click the Choose... button in the Project:Design Name column of the Design Tracing Configuration window. A window named Select a project file, then choose a design diagram will appear.

Click the required project file from the list of files. A list of design diagrams associated with the project file will be displayed. Click on a design diagram name and then click Choose.

Once a design has been selected, the path of the project file will be shown in the Project:Design Name column of the Design Tracing Configuration window. The width of the Project:Design Name column can be increased to view the full path and name of the project and design.

## 2.4.5 Tracing agent types

Agent types to be traced are configured in the Agent Type column of the Design Tracing Configuration window. When a new row is created it contains an asterisk (*) in the Agent Type column by default, meaning all agent types will be traced. Different agent types in a design diagram are traced by adding a new row for each type.

An agent type is selected by typing the agent type or by selecting it graphically. To select an agent type manually, click the Agent Type column of the Design Tracing Configuration window and type the agent type in the following format:

```
package_name.Agent_Type
```

To select an agent type graphically, click the Choose... button in the Agent Type column of the Design Tracing Configuration window. A window labelled Select a project file, then choose an agent type will appear. Click on the required project file from the list of files. A list of agent types associated with the project file will be displayed. Click on an agent type and then click Choose.

Once an agent type has been selected, the agent type will be shown in the Agent Type column of the Design Tracing Configuration window.

**Note:** Two or more agent types in a design diagram may be traced in the same TDW by naming rows with the same name.

## 2.4.6 Tracing individual agents

Individual agents to be traced are configured in the Agent Name column of the Design Tracing Configuration window. When a new row is created it contains an asterisk (*) in the Agent Name column by default, meaning all agents of the type specified in the Agent Type field of the row will be traced (if Agent Type is also * then all agents in the design diagram will be traced). The execution of agents in a design diagram can be viewed in separate TDWs by configuring rows with individual agents.

Individual agents are traced by typing the agent name and server name in the Agent Name column. Agent names should by typed as they appear in the code of the application to be traced. For example, an agent named `robot23` in an application that connects to a server named `Server` is typed as `robot23@Server`.

Two or more individual agents in a design diagram may be traced in the same TDW by naming rows with the same name.

## 2.4.7 Tracing all agents

To trace all agents in an application, select the required design and leave the agent type and agent name columns as the default settings (asterisks).

## 2.4.8 Saving and loading tracing configurations

The design tracing configuration of an application in the Design Tracing Configuration window can be saved to a *design tracing configuration file* and loaded for later use.

### 2.4.8.1 Saving a design tracing configuration

To save the current configuration in the Design Tracing Configuration window click the Save... button. A Save Trace Configuration File dialog window will open, where a new design tracing configuration file can be named and saved.

### 2.4.8.2 Loading a design tracing configuration

To load a previously saved design configuration file, click the Load... button in the Design Tracing Configuration window and select the required design tracing configuration file from the Load Trace Configuration File dialog window.

Note that when a design tracing configuration is loaded from a file, any previously configured trace settings in the Design Tracing Configuration window are replaced by the trace settings from the file.

#### 2.4.8.3 Design tracing configuration files

A design tracing configuration file contains the date and time of the file creation and details of each trace setting. The design tracing details that are saved are:

- the path and name of the project file
- project name
- design name
- agent type
- agent name
- row name.

### 2.4.9 Applying tracing settings

The design tracing configuration of an application must be applied before commencing tracing. Once all trace settings have been configured, click the Apply button in the Design Tracing Configuration window. The Design Tracing Controller window will open.

#### 2.4.9.1 Configuration errors

If a design, agent type or agent name of a row is incorrectly specified, an Error applying configuration window will appear.

When an error occurs, check the trace settings in the Design Tracing Configuration window. Ensure that the following details are correct:

- project file names and file paths
- design diagram names
- agent type names
- individual agents' names.

An error may also occur due to:

- the type of individual agents not being specified
- empty rows being present.

## 2.5 Controlling design tracing

Execution of an application traced with the DTT is controlled with the Design Tracing Controller. This window will appear after configuring tracing, or can be manually chosen from the Trace menu.

The window is divided into Global Trace Settings and Design Trace Control. Global Trace Settings allow tracing to be turned on or off, designs to be traced in descriptive mode and configuration of delayed tracing visualisation. Design Trace Control allows tracing to be stopped, run or stepped.


**Figure 2-6:** The Design Tracing Controller window

## 2.5.1 Turning tracing on and off

Tracing visualisation is turned on and off by selecting the Tracing On checkbox in the Design Tracing Controller window. Unchecking the checkbox stops tracing at the next traced transition and the executing application continues without tracing visualisation. This option is selected by default.

## 2.5.2 Tracing in descriptive mode

Tracing designs in descriptive mode is turned on by selecting the Trace in Descriptive Mode checkbox in the Design Tracing Controller window. This option displays elements of all traced designs with their descriptions. Designs are not traced in descriptive mode by default.

Descriptive mode in an individual TDW can also be toggled on and off using the Toggle descriptive mode button on the tool bar of the window.

## 2.5.3 Delaying tracing transitions

A traced application runs in realtime by default, and the transition time between each traced design element is 0. Application execution and tracing visualisation can be delayed by increasing the Transition Delay in Seconds to a number larger than 0. The transition delay can be edited before or during design tracing.

## 2.5.4 Controlling tracing

Execution of a traced application is controlled with the Stop, Run and Step buttons of the Design Tracing Controller window.

### 2.5.4.1 Starting tracing

When a traced application is resumed after connecting to a portal, the application begins running and pauses at the first step of a TDW. The Stop button is selected and tracing can be continued by selecting the Run or Step button.

### 2.5.4.2 Stopping tracing

Design tracing of an application is stopped by using the Tracing On checkbox or the Stop button. Tracing of an application may be stopped completely by:

- Right clicking on any element in the Tracing window and selecting Quit tracing on portal. (this closes the portal)
- Closing the Tracing window
- Closing the Design Tracing Controller.

When tracing is stopped the application continues to run and nothing is highlighted in TDWs.

### 2.5.4.3 Controlling tracing of individual traced design windows

Tracing of individual TDWs can be stopped and continued using the Stop/Continue tracing design button on the control bar of the window. Transitions will not be highlighted in the TDW as the traced application continues executing.

### 2.5.4.4 Stepping through tracing

The Step button is used to trace a single step of execution of the traced application. For each step, all transitions between design elements that occur at that step are shown. Tracing visualisation is stopped once the step is completed.

If JACK elements of an application without corresponding elements TDWs are being executed, there may be some delay before the next highlighted link is displayed.

### 2.5.5 Reconfiguring tracing during execution

The following steps describe how to reconfigure design tracing of an application during execution:

1. Click the Stop button in the Design Tracing Controller.
2. Open the Design Tracing Configuration window.
3. Reconfigure tracing settings and click the Apply button.
4. Resume tracing with the Run or the Step button in the Design Tracing Controller.

## 2.6 Tracing visualisation

Design tracing is displayed in traced design windows (TDWs). Traced design windows contain documentation, a design graph, a list of relevant agents and a list of relevant tasks. TDWs also have a control bar and tool bar to view components of the traced design and to control tracing visualisation. The design graph of a TDW is the only component displayed by default.

A separate TDW is displayed for each row. If two different rows with the same name trace the same design only one TDW will be shown.

The DTT highlights transitions that correspond to direct execution transitions between JACK elements. This means that although indirect links appear in a design diagram they will not be highlighted during design tracing.

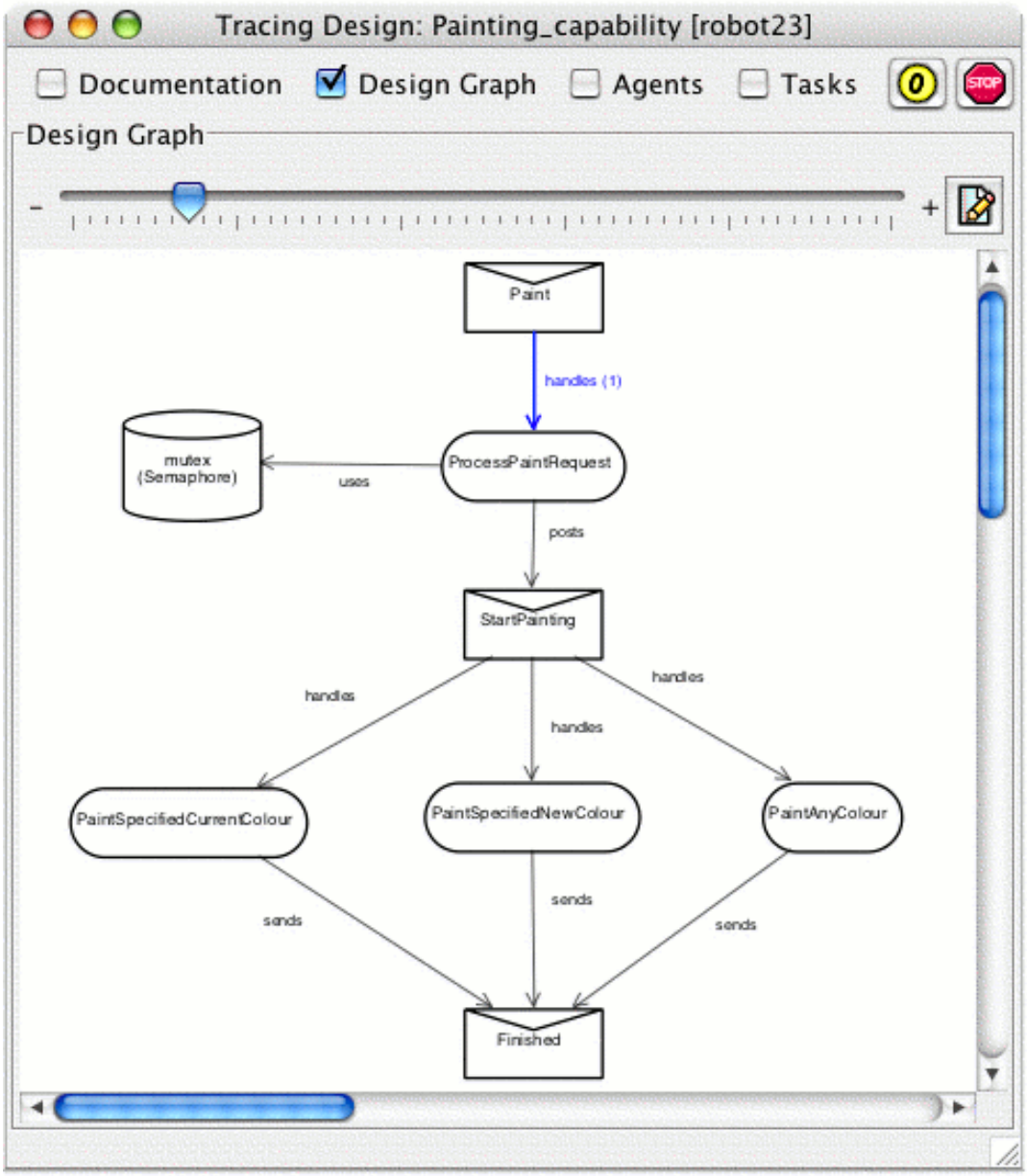


**Figure 2-7:** A design graph in a traced design window

Traced design windows can be resized, maximised, and moved within the JDE window to view design tracing in more than one window.

## 2.6.1 Traced design window control bar

The control bar of a TDW has checkboxes for displaying:

- Documentation
- Design Graph
- Agents
- Tasks.

The control bar also has two buttons for controlling tracing visualisation:

- Reset the count on links
- Stop/Continue tracing design.

The first button resets the number of times traced links in the design have been followed to zero. The second control bar button is used to stop and continue tracing of that particular window.

## 2.6.2 Traced design window tool bar

The tool bar of a TDW has a zoom slider and a Toggle descriptive mode button. The zoom slider is used to shrink or enlarge the design graph. The Toggle descriptive mode button is used to display elements in the design diagram with or without documentation.

## 2.6.3 Viewing design documentation

To view the documentation of the design graph in a TDW, select the Documentation checkbox in the control bar of the window.

## 2.6.4 Viewing a design graph

The design graph of a TDW contains the project design diagram specified in the corresponding design tracing configuration row. Design graphs are displayed for tracing visualisation only and are not editable.

To view a design graph in a TDW, select the Design Graph checkbox in the control bar of the window (selected by default). A non-editable design graph will be displayed.

## 2.6.5 Viewing relevant agents

To view a list of agents that are relevant to a TDW, select the Agents checkbox in the control bar of the window. The names, types and portals of agents relevant to the TDW are displayed.

### 2.6.6 Viewing relevant tasks

To view a list of tasks that are relevant to a TDW, select the Tasks checkbox in the control bar of the window. The details of all executed tasks relevant to the TDW are displayed. These details include the task number, the JACK elements involved and the name, type and portal of the agent executing the task. When a relevant task is selected, the design graph of the TDW appears grey to indicate that the selected task and associated transition has occurred.

### 2.6.7 Transition visualisation

Links between design elements are highlighted when a transition between corresponding JACK elements occurs. The number of times a link has been followed is shown in brackets next to the link label. For example, a link between a plan and a posted event followed twice is shown as `posts (2)`.

The traced design window containing the most recently traced transition always appears in front of other traced design windows. Transitions that occur simultaneously between design elements in the same TDW are shown as links highlighted at the same time. To view both transitions, ensure that both windows are visible (e.g. side by side in the JDE).

If JACK elements without corresponding design elements in a TDW are executed, tracing visualisation does not change. The last followed link in the current TDW remains highlighted until a link on another TDW is followed.

### 2.6.8 Resetting transition counts

The number of times links in a TDW have been followed can be reset to 0 by clicking the Reset the count on links button on the TDW control bar.

### 2.6.9 Tracing visualisation errors

#### 2.6.9.1 No traced design windows are shown

If no TDWs are displayed when a traced application is executed:

- check that all other previous executing applications have finished
- if the application is being re-executed, ensure that a new portal instance is created, you have connected to that portal and a new tracing configuration is applied.

## 2.7 A design tracing example

This section briefly describes the `paint_robot` example and explains the steps necessary to set up and run the application with the DTT. The `paint_robot` example is based on an example in the JACK™ Agent practical exercises. The example is accessed by selecting the `paint_robot` example from the Create Project From Example option in the JDE Help menu.

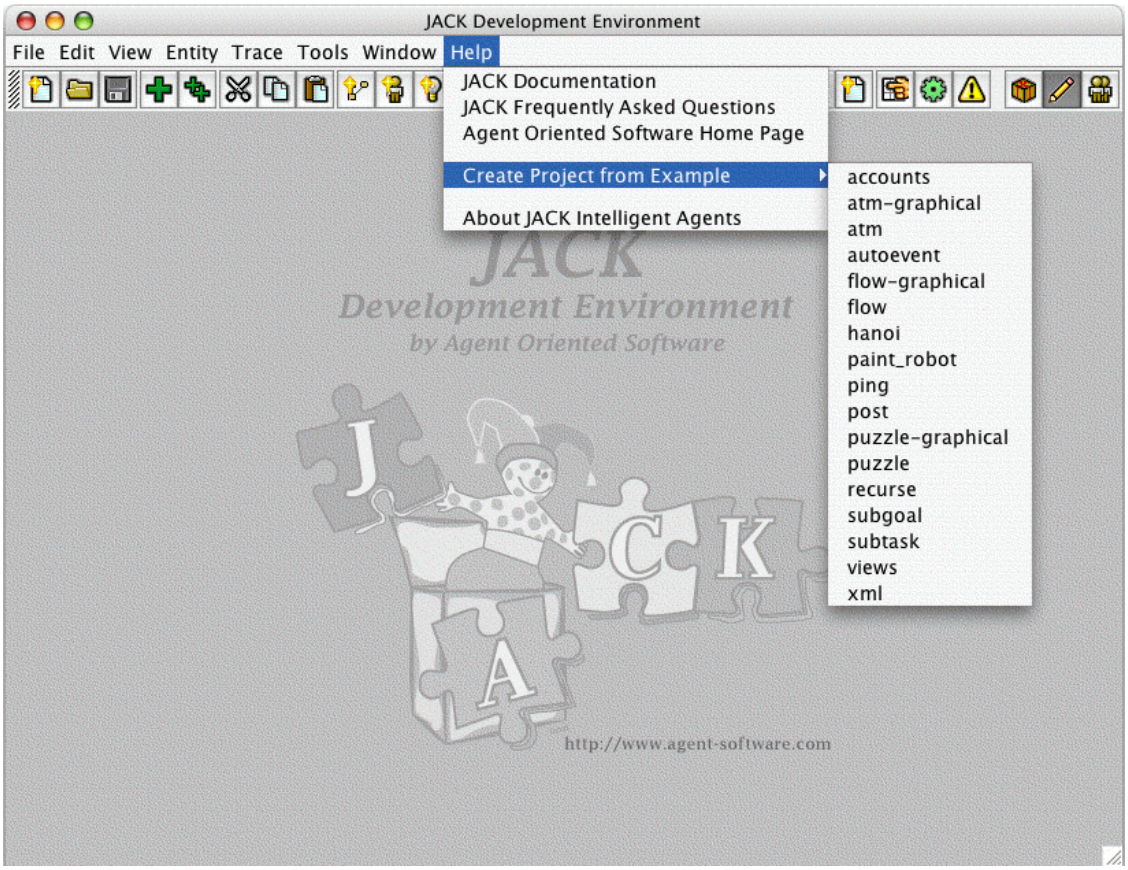


**Figure 2-8:** Create Project from Example options

The `paint_robot` example contains a Robot agent and several Part agents. The Part agents are responsible for being painted and to achieve this, they send appropriate requests to the Robot agent. The current status of the part (painted/not painted) is held by each part agent in its own beliefset.

Two design diagrams are contained in the example, `Painting_capability` and `Paint_requesting_capability`. The `Painting_capability` design contains design elements which correspond to JACK elements that are used by the `Painting` capability of the `Robot` agent type. The `Paint_requesting_capability` design contains design elements which correspond to JACK elements that are used by the `PaintRequesting` capability of the `Part` agent type.

### Step 1: Develop the application

The specification is the same as in Practical 1 of the *Agent Practicals* except that the part agent now maintains a beliefset that stores its current status. The implementation can be examined using the JDE; here we only consider the application's `main()` method:

```
import part.*;
import robot.*;
import java.io.*;
import javax.swing.JOptionPane;
import javax.swing.JFrame;

public class Program {

    public static void main( String args[] )
    {
        Robot robot1 = new Robot("robot23");
        Part part1 = new Part("part1");
        Part part2 = new Part("part2");
        Part part3 = new Part("part3");
        Part part4 = new Part("part4");

        JOptionPane.showMessageDialog(null,
            "If you intend to trace the designs in this project,"
             + "first connect\n"
             + "to the correct portal, then set up"
             + "the designs for tracing\n"
             + "before clicking OK to start the example.",
            "Start example",
            JOptionPane.INFORMATION_MESSAGE);

        System.out.println("test with red");
        part1.submitPaintRequest("robot23","red");
        System.out.println("test with no specified colour (null)");
        part2.submitPaintRequest("robot23",null);
        System.out.println("test with white");
        part3.submitPaintRequest("robot23","white");
        System.out.println("test with white again");
        part4.submitPaintRequest("robot23","white");

        JOptionPane.showMessageDialog(null,
                "The example has finished executing. Click OK to finish.",
                "Example finished",
                JOptionPane.INFORMATION_MESSAGE);
        System.exit(0);
    }
}
```

In order to trace designs in the `paint_robot` example, a new line has been added to the `Program.java` file:

- `JOptionPane.showMessageDialog(null, "Connect to the correct portal\nand click OK to start example", "Start example", JOptionPane.INFORMATION_MESSAGE);`

  This line is used to stop execution of the application until the user has connected the DTT to the application and passed control of execution to the DTT. Note that it occurs after the agents have been created, but before any events are posted.

### Step 2: Starting the application

To compile the application, select the Compiler Utility window's Compile Application tab and click the Compile button.

Once compiled, select the Compiler Utility window's Run Application tab. To enable the DTT, select the `Program.class` file and in the Java Args field enter:

```
-Djack.tracing.enabled=true -Djack.portal.name=Server
-Djack.portal.host=localhost -Djack.portal.port=9999
```

This creates a DCI nameserver and portal named `Server` on `localhost` at port 9999.

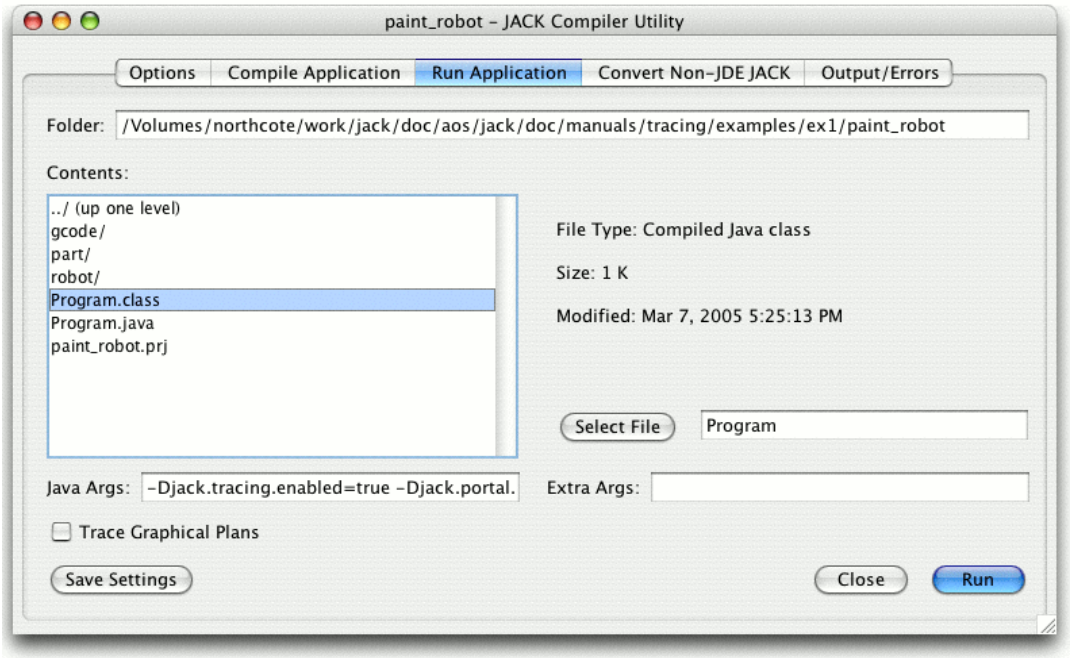


**Figure 2-9:** JACK Compiler Utility Run Application tab

Click Run. This starts the application, which will create the agents and then bring up the following dialog.


**Figure 2-10:** Start example dialog

Before clicking OK in the above dialog, the DTT needs to be connected to the application (step 3).

### Step 3: Connect the DTT to application

Select the Connect to Portal option from the Trace menu. Enter `Server` as the portal name and `9999` as the Address.

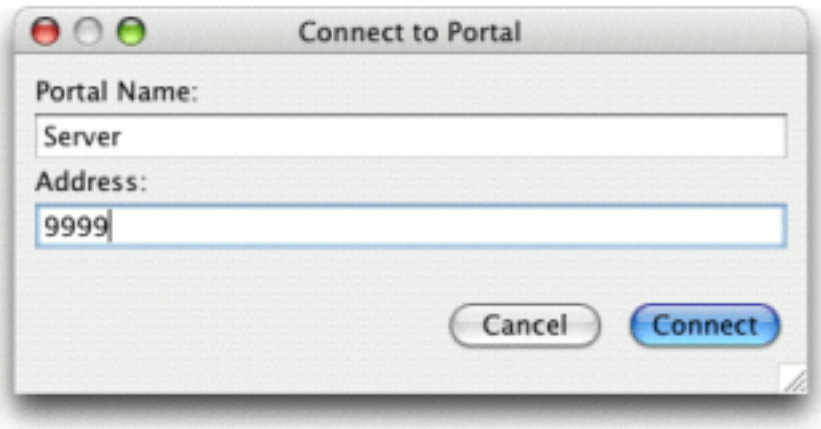


**Figure 2-11:** Connect to Portal dialog

The DTT Tracing window will now appear in the JDE. Details of the portal and existing agents will be displayed in this Tracing window.

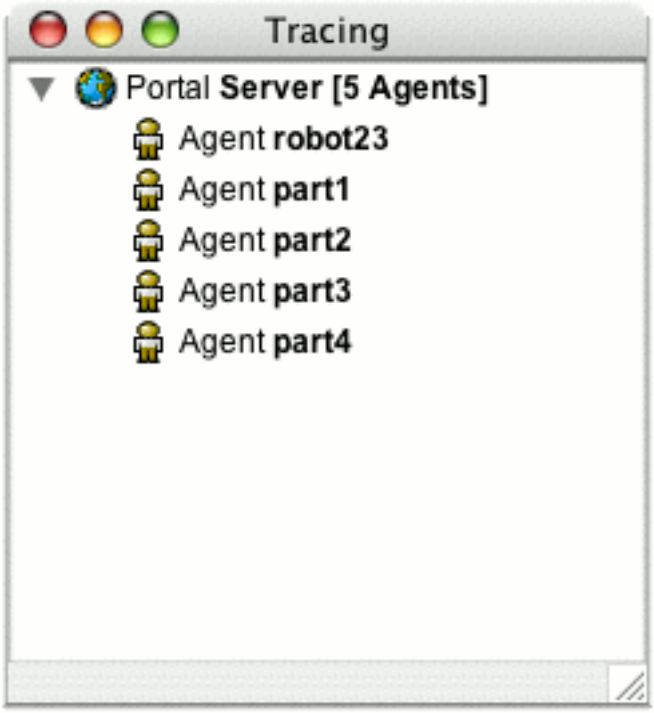


**Figure 2-12:** Tracing window

### Step 4: Configure design tracing

This step describes how to configure tracing of all agents in the `Paint_requesting_capability` design and tracing of the `robot23` agent in the `Painting_capability` design.

1. Begin by selecting the Configure Design Tracing... option from the Trace menu, which opens the Design Tracing Configuration window.
2. To add a new row for tracing all agents in the Paint_requesting_capability design, first click the Add button.
3. Click the Choose... button in the Project:Design Name column of the new row. A window will open named Select a project file, then choose a design diagram. Select the project name, `paint_robot.prj`, and then the design, `Paint_requesting_capability` and click Choose.
4. Leave the Agent Type and Agent Name fields as asterisks. To identify the row during tracing, it can be given a descriptive name such as "`all agents`" in the Trace Group column.

5. To add and configure a row that traces the `robot23` agent in the `Painting_capability` design diagram follow steps 2 and 3 above (choose the `Painting_capability design` in step 3).
6. In the Agent Name field of the row, type the name of the agent, `robot23`. To identify the row during tracing, it can be given a descriptive name such as `robot23@Server` in the Trace Group column.

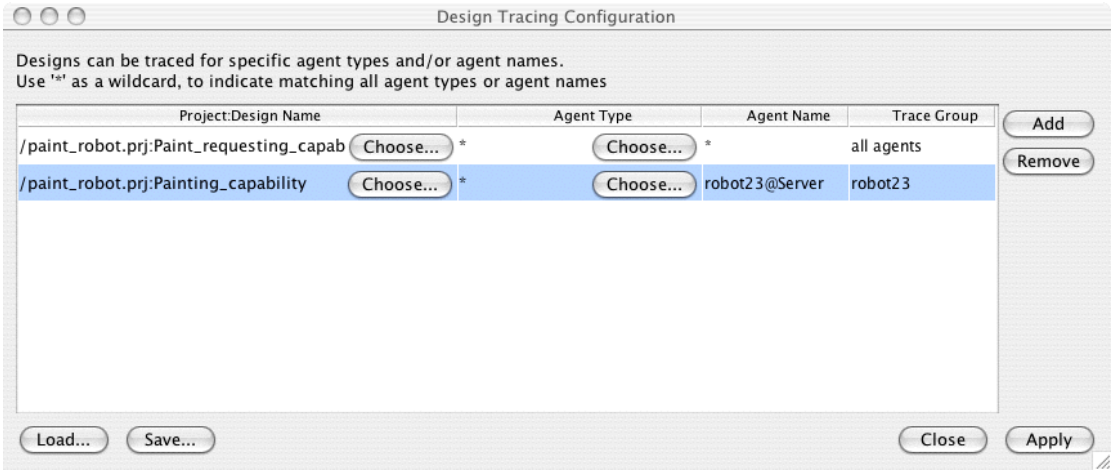


**Figure 2-13:** Trace settings in the Design Tracing Configuration window

7. To apply the design tracing configuration settings and open the Design Tracing Controller, click the Apply button. The Design Tracing Controller will open automatically.

### Step 5: Start design tracing

This step describes how to start tracing and how to step through tracing visualisation.

1. Ensure that the JDE and Design Tracing Controller are both visible so that tracing can easily be viewed and controlled at the same time.
2. Resume the application by clicking OK in the Start example dialog. The first step of visualisation will be shown.


**Figure 2-14:** Start example dialog

3. Click the Run button in the Design Tracing Controller. Transitions between design elements will be shown in both TDWs as execution passes to corresponding JACK elements.

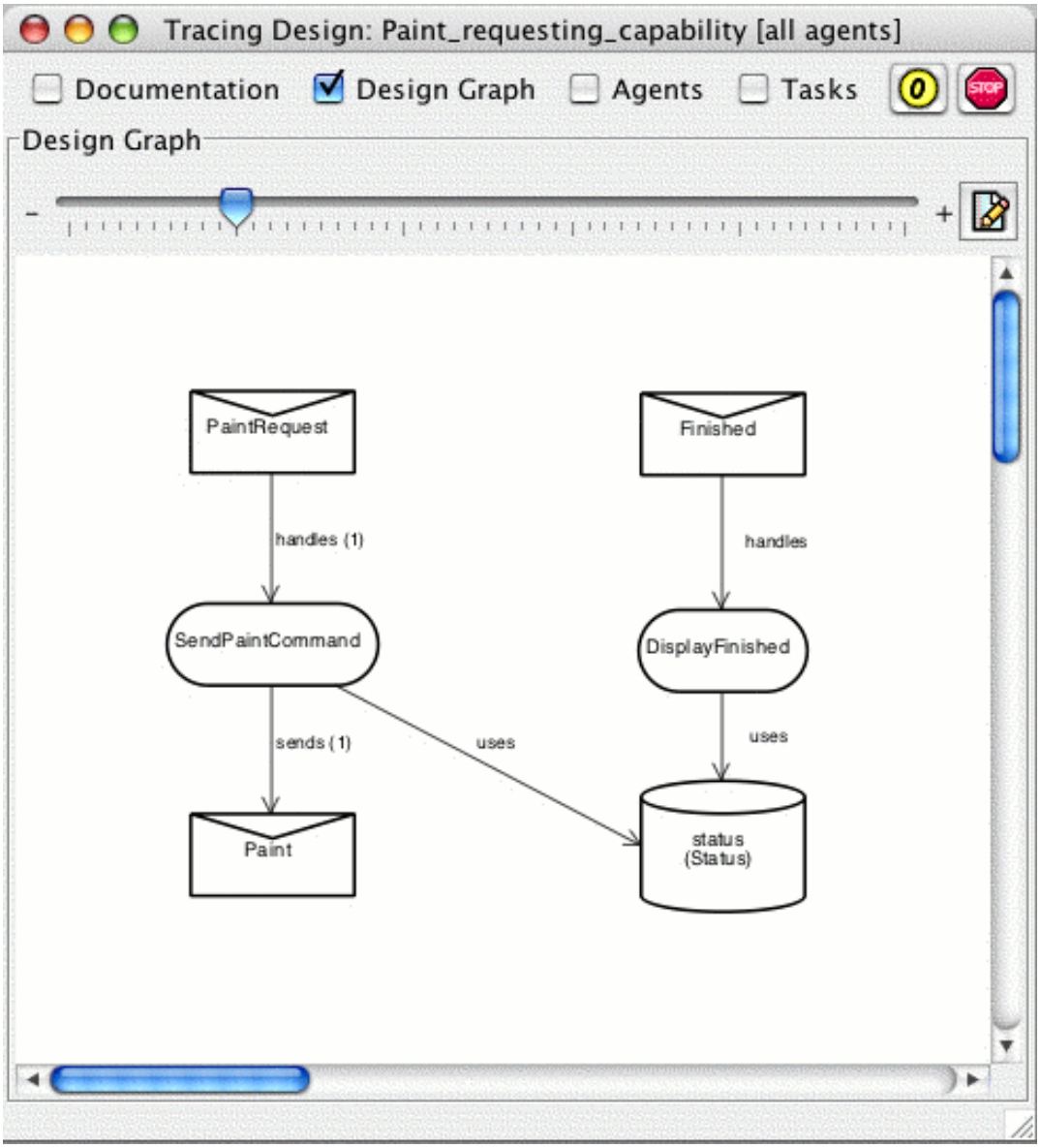


**Figure 2-15:** Tracing the `paint_robot` example

4. Once the first few steps of execution have been shown, step through tracing by clicking the Step button in the Design Tracing Controller.
5. View the relevant tasks in the robot23 TDW by selecting the Tasks checkbox on the control bar of the window.

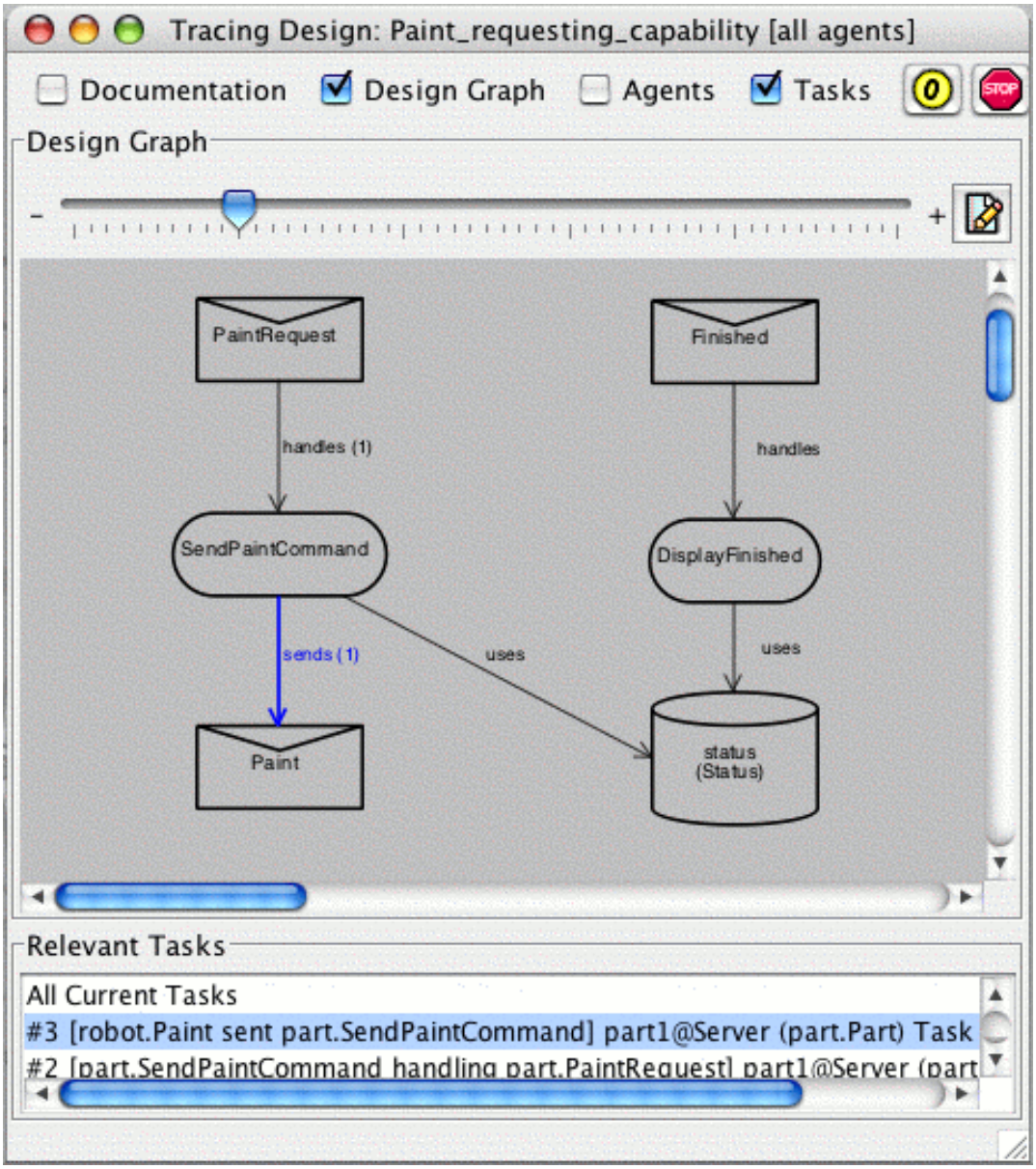


**Figure 2-16:** Relevant tasks of the robot23 traced design window

6. Continue running or stepping through tracing until the application has completed. Once the application has finished, disconnect from the `Server` portal by closing the Tracing window.

This example demonstrates the use of the DTT with a simple design tracing configuration. Use of different DTT features will vary according to the application being traced and the intended viewers of the design tracing visualisation.

# 3 Plan tracing tool

## 3.1 Introduction

The JACK Intelligent Agents® *Plan Tracing Tool* (Plan Tracing Tool) provides the ability to display and trace the execution of Graphical Plans and the events that handle them in JACK Intelligent Agents® (JACK) applications.

The Plan Tracing Tool is a graphical tool that displays agent behaviour by tracing executing tasks in a graphical environment. Tasks consist of events and the plan instances that handle them. Each event that is traced within a task can be shown graphically, with nodes that are highlighted according to actions within the task. Each plan that is traced within a task can also be shown graphically, with individual nodes of reasoning methods highlighted to reflect the current actions and decision making processes expressed in the plan.

At each stage of execution, the execution history of a task – including events and applicable plan instances – may be examined by the user. The Plan Tracing Tool displays the documentation, graph, variables and exceptions of traced events and plans. The context conditions of plans may also be viewed. Other details of traced events displayed by the tool are plan instances that are applicable to the event and plans that have failed to handle the event. The events handled by plans in the task, including applicable and failed plan instances, can also be examined.

The user can control execution of individual agents with the ability to start, stop and step tracing of those agents. This provides a powerful, visual way to interact with an executing agent system.

Graphical plans in an application may be traced after creating them in the JACK™ Graphical Plan Editor, generating JACK files with the Generate traceable plans option and compiling the application. The Plan Tracing Tool will trace events in JACK applications with or without graphical plans.

Only plans that are created in the Graphical Plan Editor can be graphically traced with the Plan Tracing Tool. Textual plans created in the JACK Development Environment can also be traced, but without the detailed step-by-step interaction possible with graphical plans.

JACK Teams™ projects may be traced with the Plan Tracing Tool. However, the reasoning methods of team plans will be shown in code form.

This chapter assumes that the user is familiar with the JACK™ Development Environment (JDE), the JACK™ Graphical Plan Editor (GPE) and JACK agents. If further information is required on these topics, refer to the *Development Environment Manual*, the *Graphical Plan Editor Manual* and the *Agent Manual*.

## 3.2 An plan tracing example

In this document, we use a simple multi-currency Bank Account system as an example and as a source of figures. It is available in the `examples/accounts` directory of the JACK installation. The example is also accessible from the Create Project from Example option of the JDE Help menu.

This section briefly describes the example application and then works through the mechanics of using the Plan Tracing Tool to investigate its operation.

To follow the example, open the project file `accounts.prj` in the JDE.

### 3.2.1 The multi-currency bank account example

The example application is a (small) part of a Banking application which maintains bank accounts in nominated currencies, and performs currency conversions to allow transactions against the accounts to occur in any currency.

The example consists of a `BankAccount` agent which maintains a table of bank accounts and performs transactions on those accounts, a `CurrencyExchange` agent which maintains a table of currency exchange rates and uses it to convert amounts from one currency to another, and a `Communicator` agent which acts as an interface.

#### The `BankAccount` agent

The `BankAccount` agent maintains a table of bank accounts in an instance of `Account` as described in the following table:

| Field | Type | Description |
|---|---|---|
| `accountNumber` | `int` | Number identifying the account. |
| `name` | `String` | Name of the account owner. |
| `currency` | `String` | Three letter code for the currency of the account. |
| `balance` | `double` | Current balance of the account. |

**Table 3-1:** Details stored about each bank account

Accounts can be held in any currency, and apart from an integer account number and the owner's name, the only other attribute stored is the current balance.

The `BankAccount` agent receives messages creating accounts, enquiring about account attributes and transactions.

Accounts are created with `CreateAccountRequest` and are credited or debited with `CreditAccountRequest` and `DebitAccountRequest` messages. Information requests are made with `AccountInfoRequest` messages.

Transactions can be in any currency, and the `BankAccount` agent makes use of the services of a `CurrencyExchange` agent to convert amounts from one currency to another.

Accounts must maintain a non-negative balance, so withdrawals that exceed the current balance are not permitted.

A transaction is performed in the currency of the account. If the `CurrencyExchange` agent cannot convert the transaction's currency to that of the account, the transaction is not permitted.

The agent uses the `SendAndWaitPlan` plan to send messages to the `CurrencyExchange` agent, and wait for and collect the replies.

When a transaction cannot be performed, the `BankAccount` agent replies with a `RequestError` message.

### The `CurrencyExchange` agent

The `CurrencyExchange` agent maintains a table of currency exchange rates in an instance of ExchangeRate, as described in the following table:

| Field | Type | Description |
|---|---|---|
| `currency1` | `String` | First currency. |
| `currency2` | `String` | Second currency. |
| `rate` | `double` | Amount of the second currency that 1 unit of the first currency will buy. |

**Table 3-2:** Details stored about the currency exchange rates

The `CurrencyExchange` agent receives messages setting exchange rates and asking for amounts in one currency to be converted to another.

Exchange rates are set with `SetExchangeRate` messages. These are handled by `SetExchangeRatePlan`.

Currency conversions are performed in response to `ExchangeRequest` messages. These are handled by `ExchangePlan`, which posts a `ComputeRate` event to determine the appropriate exchange rate to use.

The CurrencyExchange agent has several plans to handle the ComputeRate event. They are listed below.

| Plan | Description |
|---|---|
| IdentityRatePlan | Used for conversion from a currency to itself. |
| ComputeRatePlan | Used for conversion from one currency to another for which a direct rate is available in the ExchangeRate table. |
| TwoStepRatePlan | Used for conversion from one currency to another via a third. Used when no direct rate is available in the ExchangeRate table. |

**Table 3-3:** Plans used to handle the ComputeRate event

### The Communicator agent

The Communicator agent is used to interface between Java code and the BankAccount and CurrencyExchange agents. It uses SendAndWaitPlan to send messages to the other agents, and wait for and collect the replies.

### The Accounts class

The Accounts class has a main method that creates the three agent instances. The class then uses the Communicator agent to send requests to the other two agents. This is sufficient to explore the Plan Tracing Tool's functions.

## 3.2.2 Walkthrough

The Plan Tracing Tool works only with JACK projects created with the JACK Development Environment (JDE). It is designed to be most useful with applications containing plans created graphically with the Graphical Plan Editor, but can also display, though not trace, the internal steps of textual reasoning methods created with text editors within the JDE. The tool graphically displays the execution of traced events that are handled by both graphical and non-graphical plans, along with graphical plans.

A user therefore begins by developing the plans in an application with the Graphical Plan Editor tool within the JDE. JACK code is then generated with tracing support enabled and the application is compiled.

When the program is compiled and ready to run, it can be run under the control of the Plan Tracing Tool from the Compiler Utility's Run Application tab, or directly from a command prompt.

### 3.2.2.1 The plan tracing tool windows

The Plan Tracing Tool is a graphical tool that controls the execution of a JACK application and enables the user to view the execution steps taken and the values of variables computed within plans and events.

Plan tracing is controlled by an Agent Tracing Controller window.


**Figure 3-1:** The Agent Tracing Controller

With this window the user controls the execution of the agents in the traced JACK application.

Each agent within an application runs several tasks at any one time. Traced instances of plans and the events they handle are shown in task windows corresponding to the task they are associated with.

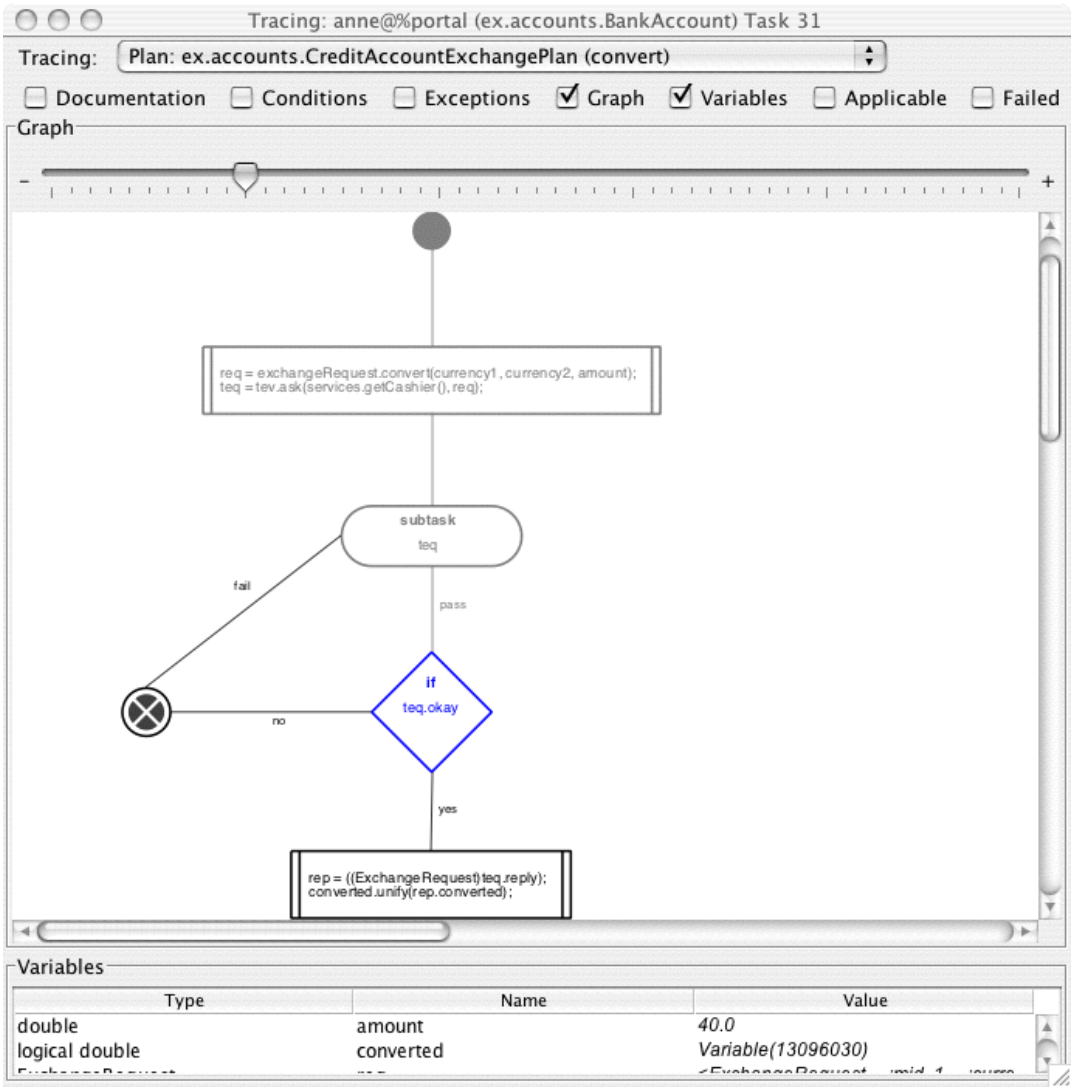


**Figure 3-2:** A task window showing an instance of the `CreditAccountExchangePlan.convert` reasoning method

The previous figure illustrates the `convert` reasoning method of `CreditAccountExchangePlan`. Execution is paused on the decision node, which is highlighted in blue to represent this. Previously executed nodes, and the links traversed to reach them, are shown in grey.

The values of the reasoning method's parameters and variables, as well as variables of the `CreditAccountExchangePlan` plan (marked with a "+"), can be seen in the Variables section of the task window. Their values are updated in the Variables section as the plan is executed.

The plan's documentation and conditions are also available in task windows.

The following figure illustrates the `CreateAccountRequest` event. Execution is paused on the `Invoke Plan` node, which is highlighted in blue to represent this. Nodes that have previously executed are shown in grey.

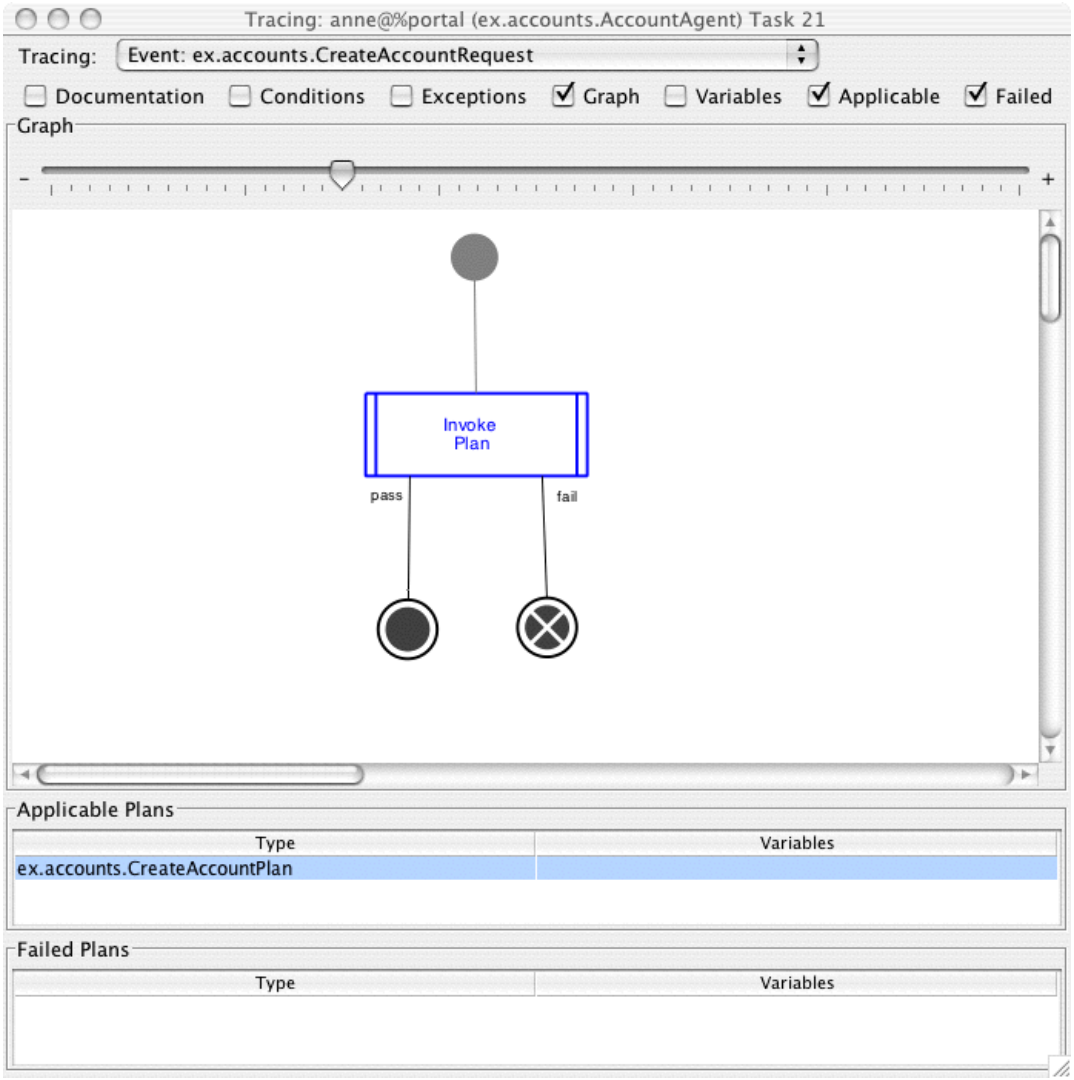


**Figure 3-3:** A task window showing the graph and plans of a `CreditAccountRequest` event instance

The graph of event execution is shown by default. Event documentation and variables are also shown in the task window.

A list of applicable plan instances that are currently being considered for processing of an event is available in the task window. The list includes details of the plan type and plan variables. If one of these plan instances has been selected to handle the event and is executing in the task, it is highlighted. A list of applicable plans can be completely recomputed during event processing.

The task window also contains a list of failed plans. These are plan instances that have been chosen to handle the event, but have failed to process the event successfully. Some event types do not have a list of failed plans, in which case the list in the task window remains empty.

A new task window is created for each new task the agent begins as soon as the task enters a reasoning method or posts an event that is being traced.

The Plan Tracing Tool determines which reasoning methods and events to trace either by tracing all of them (the default) or by reading a Plan Tracing configuration file at startup.

### 3.2.2.2 A sample run of the plan tracing tool

In this section we work through an example of compiling and running an application with tracing, to provide some immediate familiarity with using the Plan Tracing Tool.

Start plan and event tracing with the following steps:

1. Using the JACK Development Environment, open the Accounts project with the Create Project from Example option of the Help menu.
2. Open the Preferences window for the JDE. In the Preferences window, select the Graphical Plans tab.
3. Ensure that the Generate traceable plans box is selected, save preferences, and close the window.
4. Select the Generate All JACK Files option from the File menu. This is necessary after setting Generate traceable plans to ensure that all the plans in the application are compiled with tracing support.
5. Open the Compiler Utility.
6. Using the Compile Application tab, compile the application.
7. Using the Run Application tab, select the file `Accounts.class` and select the Trace Graphical Plans checkbox. Use the Select or Create... button to find the `trace1.cfg` file in the `examples/accounts` directory and set it as the Plan Tracing configuration file.
8. Run the application.

The Agent Tracing Controller window will appear, followed by a task window similar to that displayed in the the following figure.

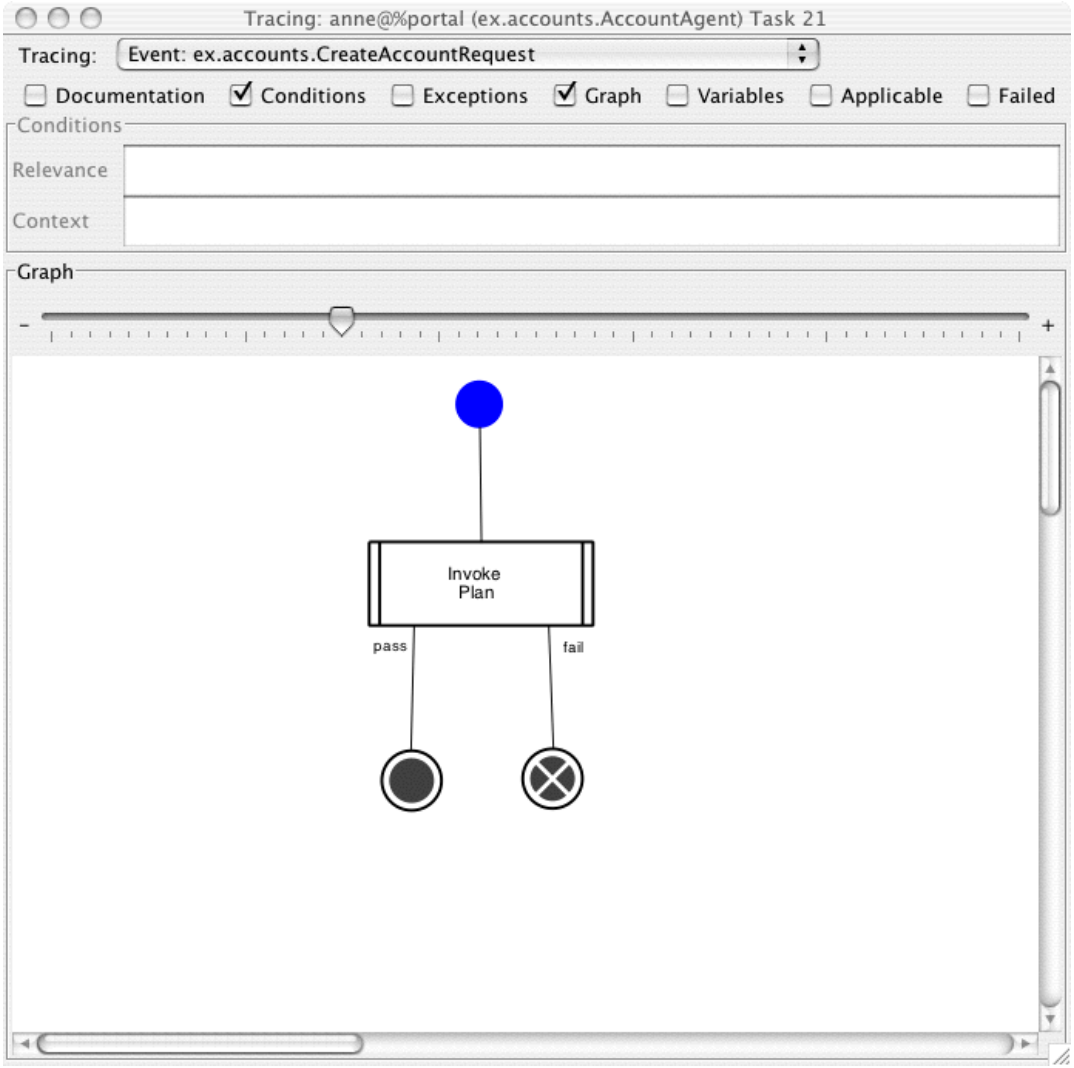


**Figure 3-4:** A task window showing the conditions and graph of a `CreditAccountRequest` event instance

The Tracing selection box at the top of the task window shows the traced reasoning methods and events active in the task. There is only one event at this stage. The row of checkboxes below it can be used to select different information to be shown in the window.

The task window currently displays the graph of event execution and the Conditions section. To view the Applicable and Failed sections instead of the Conditions section, click on the Conditions checkbox to unselect it and select the Applicable and Failed checkboxes. A task window then appears that is similar to the following figure.

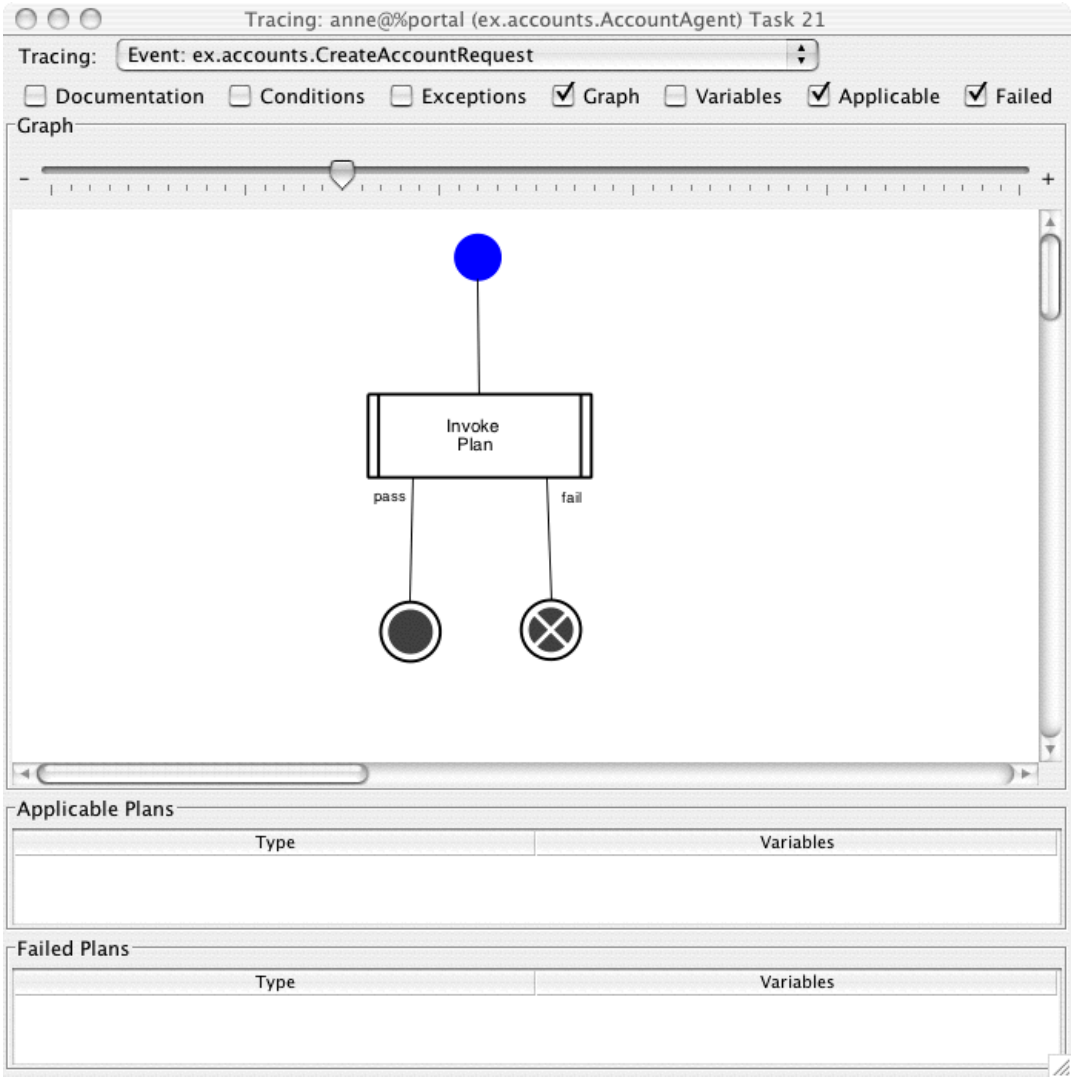


**Figure 3-5:** A task window showing the `CreditAccountRequest` event with Applicable and Failed plans

The Applicable and Failed sections will currently be empty, as the plans applicable to the event have not yet been examined.

Click on the Step button to go to the next step of execution. The `Invoke Plan` node of the graph will be highlighted in blue, and an applicable plan will be highlighted in the Applicable section, as shown in the following figure.

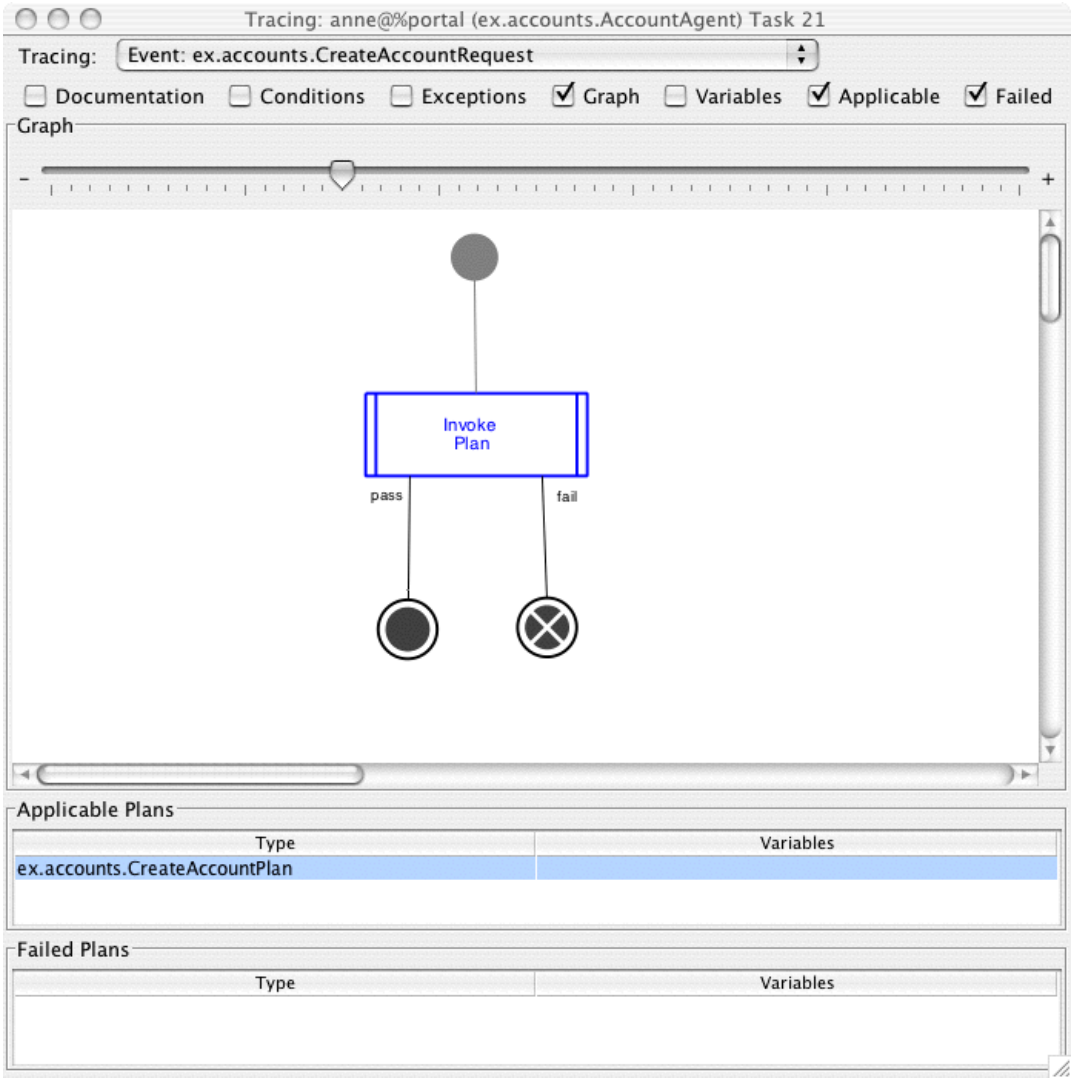


**Figure 3-6:** A task window showing the `CreditAccountRequest` event

Click on the Step button to continue plan and event tracing until the fourth task window appears. The window will be similar to the following figure.

**Note:** You may need to turn the Conditions checkbox back on and turn the Applicable and Failed checkboxes off.

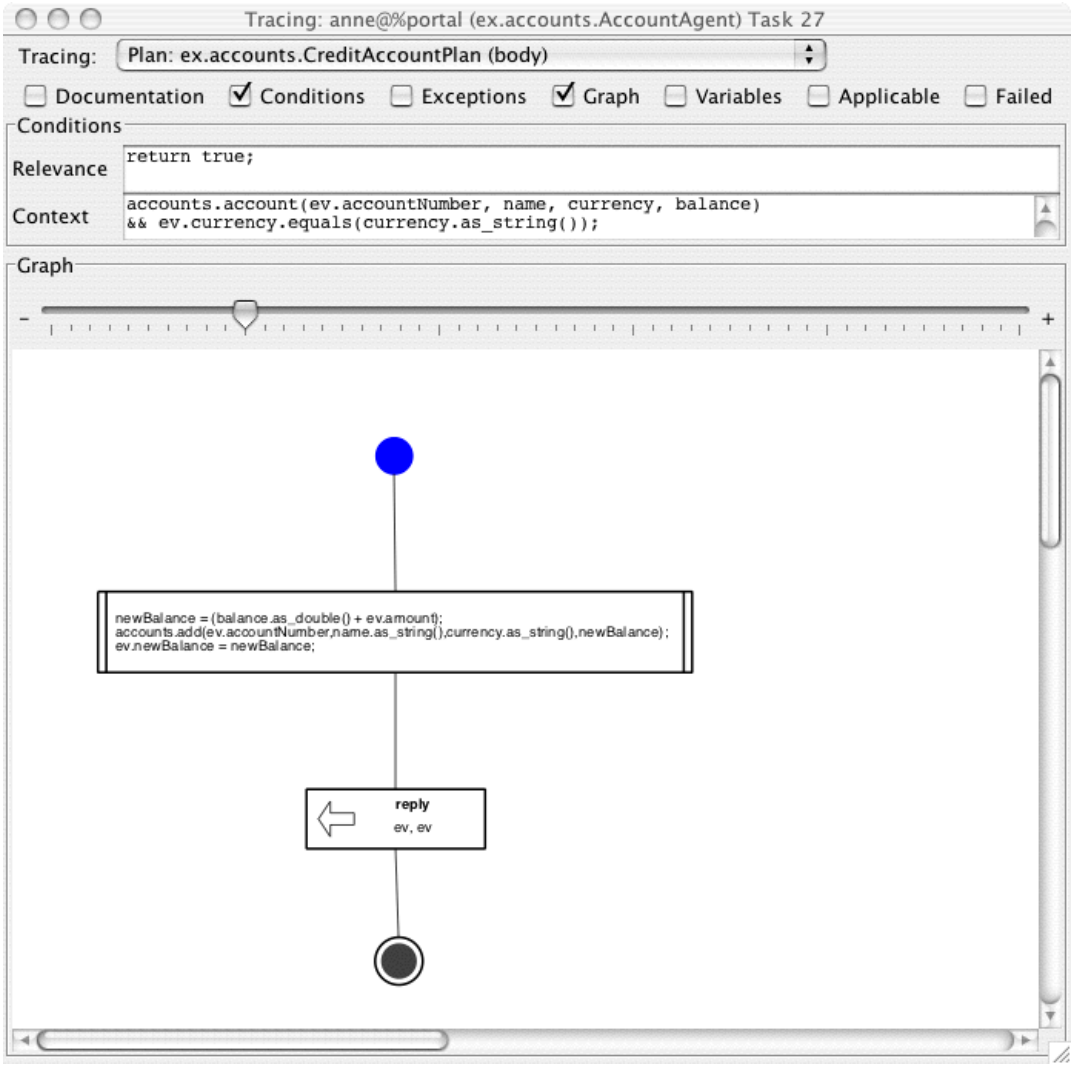


**Figure 3-7:** A task window showing an instance of the `CreditAccountPlan.body` reasoning method

In the Graph display, the node that will be executed next is shown in blue. In the previous figure, this is the `Start` node of the reasoning method.

Push the Step button on the Agent Tracing Controller to view the following figure.

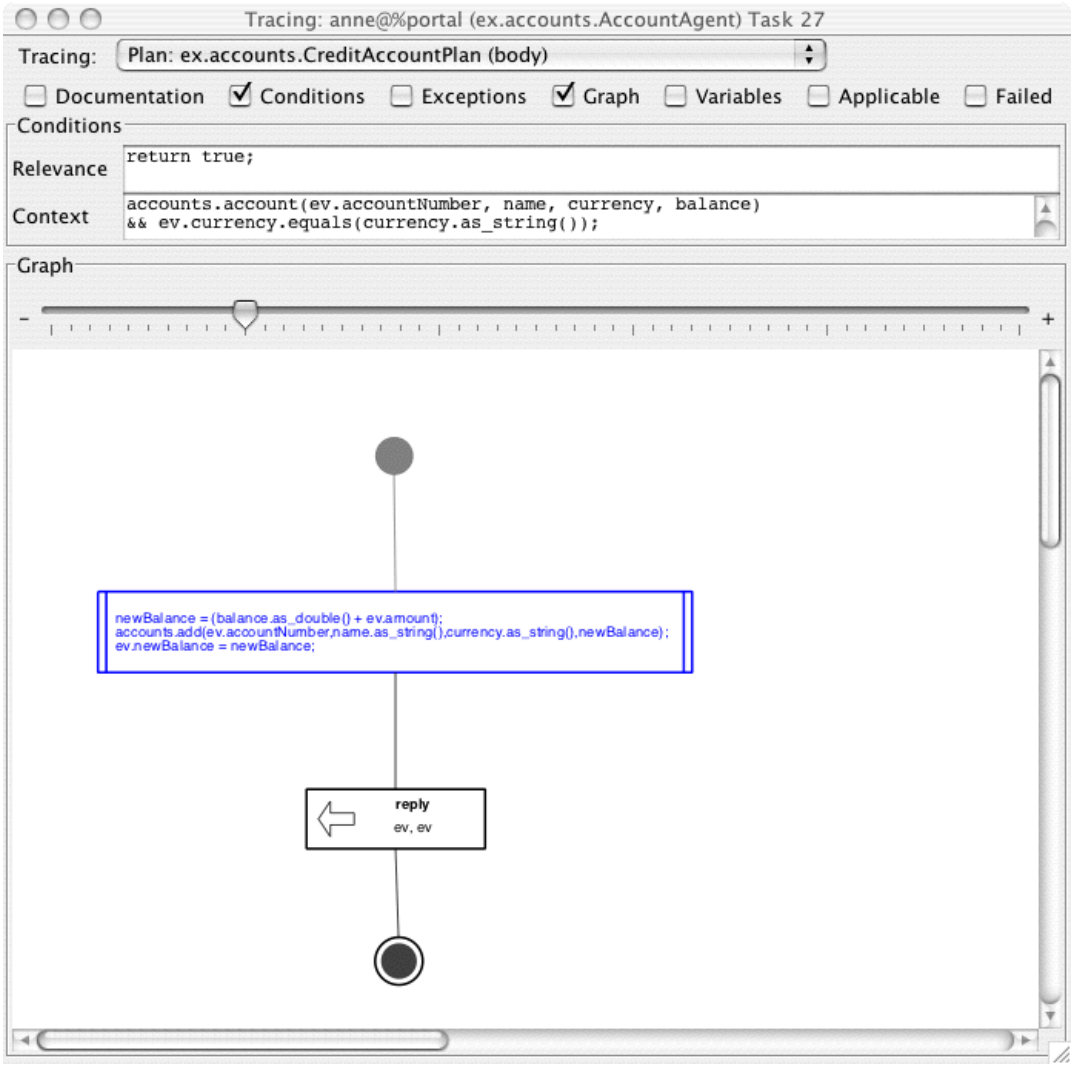


**Figure 3-8:** A task window showing an instance of the `CreditAccountPlan.body` reasoning method after a step has been taken under the control of the Agent Tracing Controller

This fourth task traced by the example is handled completely by the single plan. Click the Step button until it is completed. The task window remains, but may now be hidden by a new task window displaying the body method of `CreditAccountExchangePlan`. If so, move this new window aside to find the previous one and close it. Task windows remain on the screen after their tasks have completed until they are closed by the user.

The new task window is displayed in the following figure.

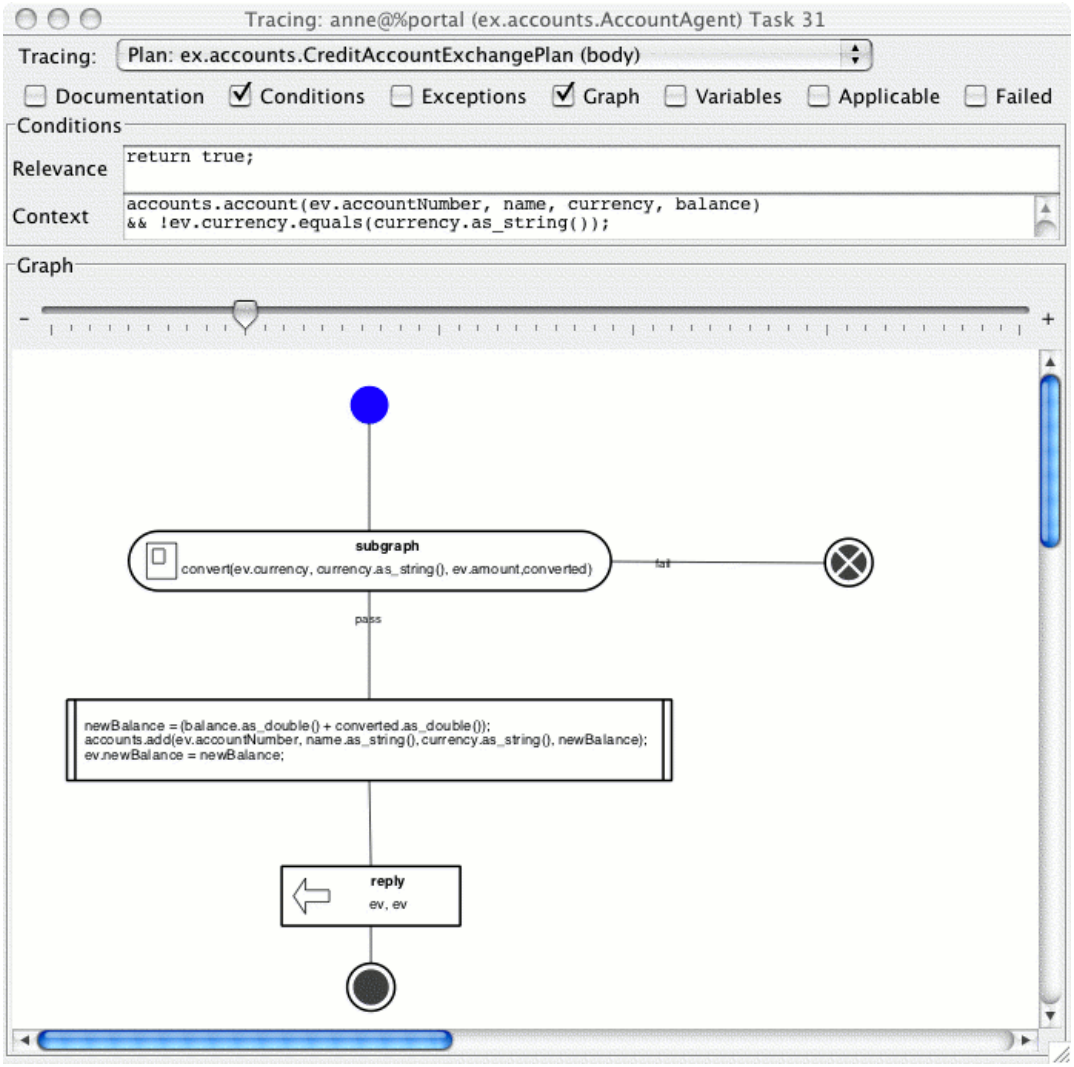


**Figure 3-9:** A task window showing an instance of the `CreditAccountExchangePlan.body` reasoning method

Step this new window three times with the Agent Tracing Controller to produce the next figure.

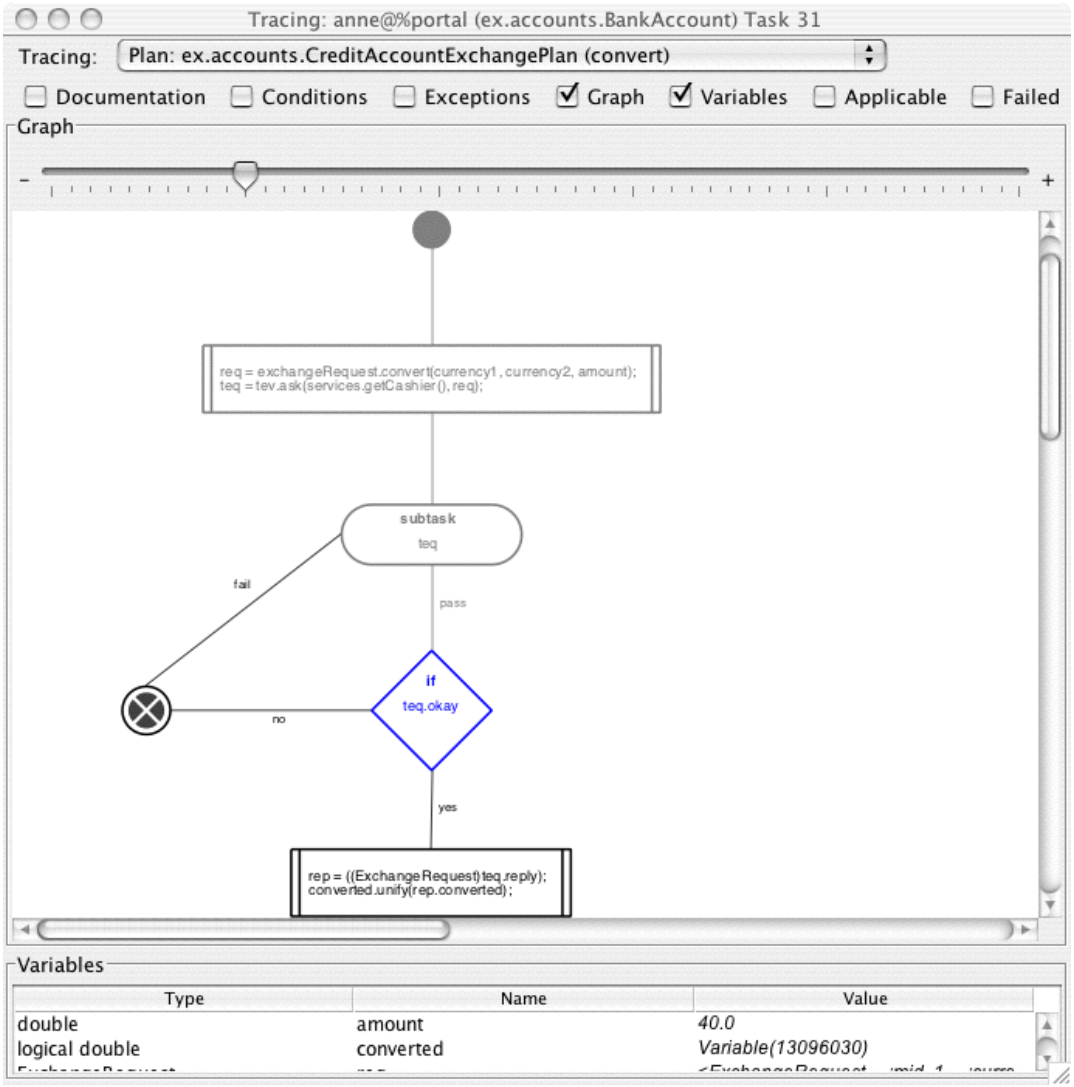


**Figure 3-10:** A task window showing an instance of the `CreditAccountExchangePlan.convert` reasoning method

Using the Tracing selection box at the top of the window, examine the `body` method and the `convert` method. Both are running in the same task, with one executing in response to the subgraph node of the other.

### 3.2.2.3 Finishing the sample run

There are several other controls on the Agent Tracing Controller.

Pressing Run will cause the agents to run freely, through all trace steps. Task windows are still created and updated as appropriate. The agents can be slowed down in this mode by setting a Transition Delay (in seconds) in the Agent Tracing Controller. Each affected agent running freely will then wait for this period at each place where it could have stopped if being stepped.

The agents to be controlled during tracing can be chosen in the selection box above the row of Stop, Run, and Step buttons. In the state reached in the figure of the `body` reasoning method of the `CreditAccountExchangePlan` plan, the only choices will be All Agents, or the `BankAccount` agent "anne". If other agents are being traced, they can be selected in this box too.

To complete running the example, click Run. After all task windows have appeared and stepped through the nodes in the reasoning methods and events shown, the system will stop. You should have sixteen (16) task windows on the screen at this stage, five (5) for each of the `creditAccount` calls in `Accounts.main`. The remaining eleven (11) correspond to the handling of traced events, less the one you closed earlier.

To finish the tracing run, close the Agent Tracing Controller. The dialogue in the following figure will appear. Choose Exit to finish the run.


**Figure 3-11:** The Exit Application dialogue

### 3.2.2.4 Tracing more than one agent

Plans from more than one agent can be specified in the Plan Tracing configuration file. For example, use the `trace2.cfg` file in the example to trace the `CurrencyExchange` agent plans and events responsible for converting currencies.

## 3.2.3 Running without the JDE

It is not necessary to use the JDE to run a traced program. Provided that the JACK application has been compiled with tracing enabled as described in the previous sections, you can run the program under the control of the tracer with a command such as:

```
java -Djack.plan.tracing.enabled=true
 -Djack.plan.tracing.config=trace1.cfg Accounts
```

Ensure that the Java `CLASSPATH` is set appropriately to allow Java to find `jack.jar` and the classes in the example.

In the remainder of this chapter, we will cover the construction of Plan Tracing configuration files, and the options that adjust the runtime behaviour of the Plan Tracing Tool.

# 3.3 Plan tracing tool configuration

Configuring the Plan Tracing Tool is a two-part process. First, a Plan Tracing configuration file is created, either using the Run Application tab of the Compiler Utility in the JDE, or by hand with a text editor. The desired runtime options are then set when the JACK application is run.

## 3.3.1 Plan tracing configuration files

A Plan Tracing configuration file is composed of lines identifying JACK agents, and the plans and events within them that are to be traced.

Each line has three tokens separated by white space. They are listed in order below:

| Field | Value |
|---|---|
| `Plan/Event Type` | The fully qualified class name of the plan or event type. |
| `Agent Type` | The fully qualified class name of the agent type. |
| `Agent Name` | The name of the agent instance. |

**Table 3-4:** Contents of a line in the Plan Tracing configuration file

**Note:** Each of these can be a wildcard "`*`" to match any agent, plan or event that matches the other columns.

For example, the `trace1.cfg` file from the walkthrough earlier contains:

```
ex.accounts.CreditAccountPlan  ex.accounts.BankAccount  *
ex.accounts.CreditAccountExchangePlan ex.accounts.BankAccount  *
ex.accounts.CreditAccountErrorPlan ex.accounts.BankAccount  *
ex.accounts.CreditAccountRequest ex.accounts.BankAccount  *
ex.accounts.AccountInfoRequest  ex.accounts.BankAccount  *
ex.accounts.CreateAccountRequest ex.accounts.BankAccount  *
ex.accounts.CreditAccountRequest ex.accounts.BankAccount  *
ex.accounts.DebitAccountRequest  ex.accounts.BankAccount  *
ex.accounts.TransportRequest  ex.accounts.BankAccount  *
```

**Figure 3-12:** Example Plan Tracing configuration file

This arranges for the tracing of the plans and events used by `ex.accounts.BankAccount` agent types to process requests to deposit money into an account.

**Note:** Lines in Plan Tracing configuration files that begin with "#" or "//" are comments.

### 3.3.1.1 Default file generation

When the Select or Create... button on the Run Application tab of the JDE's Compiler Utility is pressed and the file selected does not already exist, the JDE will offer to create it.

Plan Tracing configuration files created by the JDE enable tracing for all the plans and events within agents of the current project. This is a useful way to generate an initial configuration file that can be edited to enable tracing on only some plans, events or agents.

**Note:** If the Plan Tracing Tool cannot find a specified Plan Tracing configuration file it behaves as if no file has been specified.

### 3.3.1.2 Running with no tracing configuration file

If the Plan Tracing Tool is run without a Plan Tracing configuration file, then all plans in all agents are traced.

**Note:** If the application was not compiled with the Generate Traceable Plans preference enabled, only partial information about plans will be available to the Plan Tracing Tool. The tool will display normal tracing of event processing.

## 3.3.2 Runtime options

The behaviour of the Plan Tracing Tool can be further modified at runtime by a number of JACK options. We've seen two earlier, when we ran the Plan Tracing Tool from the Unix or Windows command line:

```
java -Djack.plan.tracing.enabled=true
-Djack.plan.tracing.config=trace1.cfg Accounts
```

These options can also be set as Java Args in the Run Application tab of the JDE's Compiler Utility.

In this section, we describe the runtime options.

### `jack.plan.tracing.enabled`

If set to `true`, this option turns on the Plan Tracing Tool at runtime. This option **must** be set in order to run plan tracing.

It is set automatically if the Trace Graphical Plans box in the Run Application tab of the Compiler Utility is ticked, but can also be set manually to run an application with the Plan Tracing Tool from the command line.

### `jack.plan.tracing.config`

This option is used to select a Plan Tracing configuration file to determine which agents, plans and events are traced by the Plan Tracing Tool.

### `jack.plan.tracing.alwaysraise`

If set to `true`, the current task window is always brought to the front of all windows whenever any node in a graph it shows is traced.

This option is `false` by default.

### `jack.plan.tracing.alwaysrestore`

If set to `true`, and a task window is minimised, this option will always restore the window when any node in a graph it shows is traced.

This option is `true` by default.

## jack.plan.tracing.runmode

If set to `true`, whenever a new agent is created, the Plan Tracing Tool is put in to Run mode as soon as tracing begins.

If set to `false`, agents are stopped when traced until Stepped or Run from the Agent Tracing Controller.

The New Agents Trace in 'Run' Mode on Creation checkbox on the Agent Tracing Controller can be used to set this property to `true` for new agents.

This option is `false` by default.

## jack.plan.tracing.zoomcombo.show

If set to `true`, shows the zoom combo box in task windows.

This control offers fixed percentage scaling values for the currently traced graph, with similar effect to the zoom slider below.

This option is `false` by default.

## jack.plan.tracing.zoomslider.show

If set to `true`, shows the zoom slider in task windows.

The slider can be adjusted to enlarge or shrink the currently traced graph.


**Figure 3-13:** The zoom slider of a task window

This option is `true` by default.

## jack.plan.tracing.zoomslider.showtickmarks

If set to `true`, shows tick marks on the zoom slider in task windows.

This option is `true` by default.

## jack.plan.tracing.tracetextual

If set to `false`, ignores reasoning methods that are non graphical.

This option is `true` by default.

## jack.plan.tracing.listallfacts

If set to `true` this option, lists all the facts in `BeliefSets` displayed in the Variables section of the task window.

If set to `false`, it lists only the number of facts in `BeliefSets` displayed in the Variables section of the task window.

This option is `false` by default.

## jack.plan.tracing.descriptivemode

If set to `false`, the Plan Tracing Tool displays reasoning method nodes in their Code form.

If set to `true`, the Plan Tracing Tool displays reasoning method nodes in Descriptive Mode.

This option is `false` by default.

# 4 Agent Interaction Diagram

## 4.1 Introduction

Inter-agent communication is a difficult aspect of programming to test and debug. When monitoring a system, it is often not sufficient to trace the internal behaviour of each agent — in order to determine exactly what is going on, it may become necessary to monitor communication between each agent and the timing of their respective messages. The Agent Interaction Diagram is provided to facilitate these activities.

Enabling the Agent Interaction Diagram in an application allows messages sent and received by each agent in the application (and those in other applications that register with the Interaction Diagram) to be viewed. The Agent Interaction Diagram is a useful tool both for analysing and debugging communication between agents. It has also proven to be invaluable when analysing and developing agent systems with a high volume of inter-agent messaging, especially when a relationship exists between message order and agent behaviour.

Having enabled the Agent Interaction Diagram, one can optionally configure the Diagram.

## 4.2 Enabling an Interaction Diagram

To enable an interaction diagram, we specify which processes are contributing to the interaction diagram, the type of interaction diagram associated with each process and optionally a JACOB file for performing interaction diagram configuration. In a multi-process application, processes can be declared to share a single diagram.

Enabling of an interaction diagram is achieved through properties which can be specified on the command line or in a properties file (one per process).

| Property | Description |
|---|---|
| `jack.tracing.idisplay.name` | The name of the interaction diagram. This is either of the form `name` or `name@portal`. `name` is used in single process configurations or for the process in a multi-process configuration which hosts the interaction diagram. `name@portal` is used for those processes in a multi-process configuration that do not host the interaction diagram. |
| `jack.tracing.idisplay.type` | The type of the interaction diagram. This is either `idproxy`, `id` or `stdout`. `idproxy` is used for those processes in a multi-process configuration that do not host the interaction diagram. `id` or `stdout` are used for single process configurations or for the process in a multi-process configuration which hosts the interaction diagram. `id` specifies that the content of the diagram is displayed on the interaction diagram GUI (which is deleted when the application exits). `stdout` specifies that the content of the diagram is displayed in text form on the standard output. This content can be made available for offline analysis by redirecting the standard output to a file. |
| `jack.tracing.idisplay.control` | The location of a configuration file specified in JACOB format. |

**Table 4-1:** Properties for enabling an interaction diagram

To make the Agent Interaction Diagram easier to understand a `message` member is available in the `MessageEvent` and `BDIMessageEvent` classes. It takes the form;

```
public String message;
```

The `message` member is accessible with the `getMessage` method. The base implementation of this method returns a member or an empty `String` if the member is null. The method takes the form:

```
public String getMessage()
```

When writing posting methods for message events, a descriptive statement should be assigned to this member. This text will then appear in the Agent Interaction Diagram, allowing easy identification of the message that has been sent.

Below we summarise the steps involved in the creation of an interaction diagram called `blue_poles` for both single process and multiple process applications.

1. Single Process Applications
   - In the application, use `MessageEvents` or `BDIMessageEvents` for communication between agents and assign meaningful text to the `message` member of each message event.
   - In the directory from which the application is launched, create a file called `pfile` which contains the following property assignments:

     ```
     jack.tracing.idisplay.type=id

     jack.tracing.idisplay.name=blue_poles
     ```

   - Invoke the application with the following additional command line argument:

     ```
     -Djack.property.file=pfile
     ```

   The Agent Interaction Diagram will appear when the application is launched.

2. Multiple Process Applications
   - Refer to the *Inter-agent Communications* chapter of the *Agent Manual* for an explanation of how to configure multi-process applications.
   - In the application, use `MessageEvents` or `BDIMessageEvents` for communication between agents and assign meaningful text to the `message` member of each message event.
   - Choose one of the processes to host the interaction diagram and choose a portal name for it. We will use `portal0` for this example.
   - Choose names for the property files for each process.
   - Create the file `pfile0` in the directory in which the hosting process is launched and add the following property assignments:

     ```
     jack.tracing.idisplay.type=id

     jack.tracing.idisplay.name=blue_poles
     ```

   - In each of the directories in which the remaining processes are launched, create the appropriately named properties file, all with the following contents:

     ```
     jack.tracing.idisplay.type=idproxy

     jack.tracing.idisplay.name=blue_poles@portal0
     ```

   - Invoke each process with the following additional command line argument:

     ```
     -Djack.property.file=property_file
     ```

   The Agent Interaction Diagram will appear when the application is launched.

## 4.3 Configuring an Interaction Diagram

Configuration is concerned with the look and content of the actual interaction diagram. This (optional) configuration can be achieved using either a properties file (typically the file used for enabling the interaction diagram for the process hosting the interaction diagram) or a JACOB configuration file (specified by the `jack.tracing.idisplay.control` property for the process hosting the interaction diagram). If configuration details are provided for a process using both methods, the configuration details in the properties file are ignored.

1. Configuration Using Properties

The following properties can be used to configure the look and content of an interaction diagram:

| Property | Description |
|---|---|
| `jack.tracing.idisplay.x`,<br>`jack.tracing.idisplay.y`,<br>`jack.tracing.idisplay.height`,<br>`jack.tracing.idisplay.width` | Control the shape of the interaction diagram frame. |
| `jack.tracing.idisplay.font` | Sets the font for all text in the display. |
| `jack.tracing.idisplay.autoscroll` | Boolean value — it controls whether or not the display scrolls automatically to show the last event. The default is `true`. |
| `jack.tracing.idisplay.temporal` | Boolean value — it controls whether or not to present the temporal order of events, as seen by the displayer. The default is `false`. |
| `jack.tracing.idisplay.showagents` | Boolean value — it controls whether or not to include sender and receiver with the message text. The default is `false`. |
| `jack.tracing.idisplay.goals` | Boolean value — it controls whether or not to display goal trace events. The default is `false`. |
| `jack.tracing.idisplay.agentwidth` | The pixel width for each agent column. |
| `jack.tracing.idisplay.messageheight` | The pixel height for trace events. |
| `jack.tracing.idisplay.headheight` | The pixel height for the source side indicator. |

**Table 4-2:** Properties to configure the appearance of an interaction diagram

**Note:** The JACK kernel also has the boolean property `jack.tracing.idisplay.details` which tells whether or not to trace all events, or just the top level ones. This only has an effect on the display when `jack.tracing.idisplay.goals` is set to true. However, it will affect the performance even if the `jack.tracing.idisplay.goals` flag is set to false.

2. Configuration using JACOB

If the property `jack.tracing.idisplay.control` specifies a JACOB file which uses objects from the dictionary `aos/jack/gui/idnew/control.api`, this file is used for interaction display configuration. This dictionary defines three types of objects, each of which controls a different aspect of the interaction diagram.

a. The `InteractionDisplayTuning` Object

The configuration file should contain a single instance of this type. Notice that most of the fields have the same names as system properties in the table above, since they perform the same functions.

The fields of the `InteractionDisplayTuning` object are detailed below.

| Field | Type | Description | Default Value |
|---|---|---|---|
| `x` | `int` | Horizontal location of left edge of window. | |
| `y` | `int` | Vertical location of top edge of window. | |
| `width` | `int` | Width of window in pixels. | 400 |
| `height` | `int` | Height of window in pixels. | 400 |
| `font` | `String` | Default font for all text. | Courier-12 |
| `agentWidth` | `int` | Default width for agent columns. | 50 |
| `messageHeight` | `int` | Default height for messages. | 20 |
| `headHeight` | `int` | Default height for the message head symbol. | 20 |
| `autoScroll` | `boolean` | Whether to scroll automatically with new messages. | `true` |
| `temporal` | `boolean` | Show the 'true' temporal order of send/receive events. | `false` |
| `showAgents` | `boolean` | Include agent names in messages. | `false` |
| `goals` | `boolean` | Show all agent goals. | `false` |
| `agents` | `aggregation` | Aggregation of AgentDisplay objects (see below). | |
| `messages` | `aggregation` | Aggregation of MessageDisplay objects (see below). | |

**Table 4-3:** The fields of the `InteractionDisplayTuning` object

**Note:** The JACK kernel has the boolean property `jack.tracing.idisplay.details` which tells whether or not to trace all events, or just the top level ones. This only has an effect on the display when the `goals` field is set to true. However, it will affect the performance even if the `goals` field is set to false.

b. The `AgentDisplay` Object

This object stores configuration information that applies to a single agent's display in the interaction diagram. As many of these objects as required can be included in the `agents` field of the `InteractionDisplayTuning` object. The fields are detailed below.

| Field | Type | Description | Default Value |
|---|---|---|---|
| `name` | `String` | The name of the agent concerned. | |
| `hide` | `boolean` | Turns off display of this agent if set to `true`. | `false` |
| `showAtStart` | `boolean` | Causes this agent to be displayed when the interaction diagram starts up, even if the agent doesn't exist at that point. | `false` |
| `width` | `int` | Defines the column width for this agent. | |
| `displayedName` | `String` | The name that should be used to identify this agent in the interaction diagram (if null or omitted, the agent's actual name is used). | |
| `nameBackgroundColour` | `String` | The colour used for the box behind the agent's name. | |
| `nameTextColour` | `String` | The colour used for the agent's name (at the top of the interaction diagram). | |
| `lineColour` | `String` | The colour used for the vertical line representing this agent. | |
| `font` | `String` | The font used for the display of this agent's name at the top of the diagram. | |

**Table 4-4:** The fields of the `AgentDisplay` object

The three fields that specify colours (`nameBackgroundColour`, `nameTextColour` and `lineColour`) can take either comma-separated RGB values (with a range of 0-255 for each element) or the names of constant colours defined in the `java.awt.Color` class, specifically:

- `black`
- `blue`
- `cyan`
- `darkgray` (or `darkgrey`)
- `gray` (or `grey`)
- `green`
- `lightgray` (or `lightgrey`)
- `magenta`
- `orange`
- `pink`
- `red`
- `white`
- `yellow`

c. The `MessageDisplay` Object

This object stores configuration information that applies to the display of individual messages in the interaction diagram. As many of these objects as required can be included in the `agents` field of the `InteractionDisplayTuning` object. The fields are detailed below.

| Field | Type | Description | Default Value |
|---|---|---|---|
| `pattern` | `String` | Pattern expression for which this tuning applies. Asterisks ('*') can be used as wildcard characters; this tuning will then be used for any messages matching the pattern. | |
| `hide` | `boolean` | Turns off display of these messages if set to `true`. | `false` |
| `height` | `int` | Defines the pixel height to use for these messages. | |

**Table 4-5:** The fields of the `MessageDisplay` object

To create and edit a configuration file using the JACOB graphical editor, follow these steps:

1. Create an empty configuration file (`cfile.cfg`) with. for example, Notepad (Windows) or the touch command (UNIX).
2. Invoke the JACOB graphical editor with the following command:

   ```
   java aos.main.Jacob cfile.cfg -t aos/jack/gui/idnew/control.api
   ```

   An icon for the file (`cfile.cfg`) appears in the left pane of the window.
3. To add objects to the file, first display the icon's contextual menu (right-click/control-click).
4. Then from the menu, choose 'Add Top Level Object' and select the kind of object that you want to create.
5. When editing of the configuration file is completed, choose 'Save' from the 'File' menu to save the file.

**Note:** For more information about using the JACOB graphical editor, refer to the *JACOB Manual*.

# 5 Audit Logging

The programmer can trace the activity of JACK entities by setting the `jack.run.debug.options` property on the command line. For example:

```
java -Djack.run.debug.options=messages:events Test
```

activates both tracing of messages and events. As this example illustrates, more than one mode of tracing can be activated by supplying a list separated by colons (`":"`). A list of the more commonly used trace modes is given below:

| Mode | Traces |
|---|---|
| `applicable` | Applicable sets for handling events. |
| `applsets` | Creation of applicable plan sets. |
| `beliefs` | JACK beliefset activity. |
| `bindings` | Variable bindings and backtracking. |
| `doit` | Processing of the agents todo list. |
| `eventfailure` | Event failure. |
| `eventpass` | Event success. |
| `events` | Event posting and processing. |
| `excep` | Exception posting and handling within tasks. |
| `exec` | Processing of JACK executor (usually agents). |
| `messages` | Message sending and receiving. |
| `observers` | Activity of Watchable entities. |
| `planfailure` | Plan failure. |
| `planpass` | Plan success. |
| `plans` | Plan activity. |
| `relevance` | Relevance sets for handling events. |
| `scheduler` | transitions in the JACK scheduler. |
| `tasks` | Task creation and completion. |

**Table 5-1:** JACK trace modes

Use of the debugging flags results in the details of the execution being logged to the standard error. The debugging flags are often used together with redirector properties. For example,

- `Ddebug.setError=outputFile1`
- `Ddebug.setOutput=outputFile2`
- `Ddebug.setInput=inputFile`

would redirect (the messages produced from the `jack.run.debug.options` property from) standard error to file `outputFile1`, from output to file `outputFile2` and redirect standard input to file `inputFile`.

# 6 Generic Debugging/Agent Debugging

Generic debugging of a JACK™ (JACK) application (also known as Agent Debugging) is performed using JACOB™ (JACOB) objects that implement the `AgentDebuggerCommand` interface. This generic debugging tool is a simple and extensible remote agent debugger. It accepts connections on the `jack.debugger.port` and then creates a thread that reads JACOB debug objects from it and casts them to the `AgentDebuggerCommand` type. The debugger then calls the `process` method in the debug object.

Debugging capabilities are specified with debug objects, which are defined with JACOB dictionary files. The default debugging capability is defined with the `DumpState` object which is currently the only supported debug object. See the Debug objects section of this chapter for information on the `DumpState` object. The debugging capabilities of the tool are extended by defining new debug objects. This process is described in the User defined debug objects section of this manual.

The `AgentDebuggerCommand` interface is controlled with two Java properties:

- `jack.debugger.port` .The TCP port on which the generic debugging tool will accept objects.
- `jack.debugger.command`

  A list of the dictionary files to be loaded, separated by colons (":").

This chapter assumes that the user is familiar with JACK agents and JACOB. If further information is required on these topics, refer to the *Agent Manual* and the *JACOB Manual* .

## 6.1 Using debugging

To use generic debugging the JACK kernel must listen for a connection on a portal. The user then connects to the portal using telnet and can enter commands (debug object class names) which will result in the kernel logging information about the agents. Commands are entered on the telnet command line.

To start generic debugging the user sets the following properties on the command line:

```
java -Djack.debugger.port=nnnn
     -Djack.debugger.commands=<JACOB_definition_file>
```

It is possible to extend generic debugging capabilities with additional commands entered by the user. Additional commands are entered with:

```
java -Djack.debugger.port=nnnn
     -Djack.debugger.commands=
      <command1.api>:<command2.api>:<command3.api>
```

This establishes a thread that accepts connections from the specified port. Once a connection is established a thread reads JACOB objects off the socket and invokes the `process` method of the `AgentDebuggerCommand` interface. The user can then telnet to the nominated port and type debugging commands to dump the state of agents in the application.

The initial debugging command available is `DumpState` and it can be used at the telnet command line as follows:

```
<DumpState [:agent "myAgent"] [:stderr :true]>
```

This command will print a JACOB object that describes the agent `myAgent` to the standard error of the process. If the code `:agent myAgent` is omitted, it will dump the state of all agents. If the code `:stderr :true` is omitted, any output will go to the socket instead of the standard error.

## 6.2 The `AgentDebuggerCommand` interface

The `AgentDebuggerCommand` interface has one method, `process`.

```
public interface aos.jack.jak.agent.AgentDebuggerCommand
{
   public void process(java.net.Socket, aos.apib.inStream,
                                  aos.apib.OutStream);
}
```

The `process` method is called on debug objects that are read by the thread established by the debugger. The method passes in the streams `java.net.Socket`, `aos.apib.inStream`, and `aos.apib.OutStream` to allow the client object to perform further communications with the client.

## 6.3 Debug objects

Debug objects are defined with JACOB. They implement the `AgentDebuggerCommand` interface and override the `process` method. The objects allow the debugging tool to be extended to examine different attributes of an executing JACK process.

### 6.3.1 DumpState

This debug message is included in all JACK applications. It is used to dump the state of a running JACK process. The `DumpState` object has two fields:

- `agent`

  The name of an agent. If it is null or "" then all agents are dumped.

- `stderr`

  A boolean flag that determines if the dump output is sent to `stderr` or returned as a message down the socket. If `stderr` is `true` the dump information is output on the JACK processes stderr. This is useful if you are capturing output from the executing JACK process. If `stderr` is null or false, the dump information output is sent to the socket.

### 6.3.2 User defined debug objects

The user can define debug objects by using JACOB to implementing the `process` method of the `AgentDebuggerCommand` interface. For example, if you wanted to define a message to determine how much memory a JACK process was using you could do so by creating a dictionary file, in this case named `MyDebugMessages.api` with the following contents:

```
<Code :lang "java" :code "package mydebug">

<Class :name "GetMemUsage"
   :implements ( <APIString :val "aos.jack.jak.agent.AgentDebuggerCommand">
                 )
   :Directive (
      <Code :lang "java" :code `
public void process(Socket s, InStream in, OutStream out)
{
   Runtime r = Runtime.getRuntime();

   PrintWriter pw = new PrintWriter(new OutputStreamWriter(out));
   pw.println("Memory Report: Free="+r.freeMemory()+" Max="+r.maxMemory()
                  +" Tot="+r.totalMemory());
   pw.flush()
}
      `>
       )
>
```

## 6.4 Running debugging

To run a JACK application with debugging:

1. Create a dictionary file that defines debug objects. This example uses the `MyDebugMessages.api` file.
2. Use JACOB to generate classes from the dictionary file. See the JACOB Manual for further information on using JACOB.
3. Compile the classes generated by JACOB.
4. Add the generated classes to the classpath.
5. Run the application. The following is an example of debugging an application named `DebugApplication`:

```
java -Xmx90m aos.main.Jack -Djack.debugger.port=19999
     -Djack.debugger.command=mydebug.Init__MyDebugMessages
     DebugApplication
```

Alternatively, use the above Java properties when running the application from the JACK Development Environment.

6. Use telnet to connect to port 19999:

```
telnet localhost 19999
```

7. At the telnet command line type:

```
<GetMemUsage>
```

A message similar to the following will be displayed:

```
Memory Report Free=102920 Max=129873 Tot=627181
```

# Appendix A: 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 property. The properties are listed under the tools that they customise.

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.

The following is an example of modifying the behaviour of the Plan Tracing Tool with the JACK property `jack.plan.tracing.descriptivemode`. When this option is set to `true` plan tracing is shown in descriptive mode when a traced application starts. To run an application with this property activated from the command line use:

```
java -Djack.plan.tracing.enabled=true
-Djack.plan.tracing.descriptivemode=true Application
```

# 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 A-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. |

| Property | Description | Type | Default |
|---|---|---|---|
| `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 A-2:** JACK Runtime Environment Properties

# JACK Debugging Properties

| Property | Description | Type | Default |
|---|---|---|---|
| `jack.debugger.port` | Specifies the TCP port on which `AgentDebugger` commands will be accepted. | `int` | Next available port |
| `jack.debugger.commands` | Specifies a colon-separated list of dictionary files that define debugging objects that are to be used with the application. | `String` | `null` |
| `jack.run.debug.agents` | Specifies the prefix for the filenames that agent log messages will be redirected to instead of `stdout`. The filenames will be of the form `<prefix>-<agentname>.log`. | `String` | `null` |
| `jack.run.debug.options` | Specifies a colon-separated list of tracing modes that are to be applied to the application. | `String` | `null` |
| `jack.run.debug.show.count` | Numbers each debug message. | `boolean` | `false` |
| `debug.setError` | Redirects standard error output to the specified file. | `String` | `null` |
| `debug.setOutput` | Redirects standard output to the specified file. | `String` | `null` |
| `debug.setInput` | Redirects standard input from the specified file. | `String` | `null` |

**Table A-3:** Debugging Properties

# Design Tracing Tool Properties

| Property | Description | Type | Default |
|---|---|---|---|
| `jack.tracing.enabled` | Enables tracing of project design diagrams. | `boolean` | `false` |

**Table A-4:** Design Tracing Tool Properties

# Agent Interaction Diagram Properties

| Property | Description | Type | Default |
|---|---|---|---|
| `jack.tracing.idisplay.name` | Specifies the name of the interaction diagram. | `String` | "IDTRACER" |
| `jack.tracing.idisplay.type` | Specifies the type of the interaction diagram. This is either "idproxy", "id" or "stdout". | `String` | `null` |
| `jack.tracing.idisplay.control` | The location of a configuration file specified in JACOB format. | `String` | `null` |
| `jack.tracing.idisplay.x,`<br>`jack.tracing.idisplay.y,`<br>`jack.tracing.idisplay.height,`<br>`jack.tracing.idisplay.width` | Controls the shape of the interaction diagram frame. | `int`<br>`int`<br>`int`<br>`int` | `-1`<br>`-1`<br>`400`<br>`400` |
| `jack.tracing.idisplay.font` | Specifies the font for all text in the display. | `String` | "Courier-12" |
| `jack.tracing.idisplay.autoscroll` | Scrolls the display automatically to show the last event. | `boolean` | `true` |
| `jack.tracing.idisplay.temporal` | Displays the temporal order of events, as seen by the displayer. | `boolean` | `false` |
| `jack.tracing.idisplay.showagents` | Includes the sender and receiver with the message text. | `boolean` | `false` |
| `jack.tracing.idisplay.goals` | Displays goal trace events. | `boolean` | `false` |
| `jack.tracing.idisplay.agentwidth` | Specifies the pixel width for each agent column. | `int` | 50 |
| `jack.tracing.idisplay.`<br>`messageheight` | Specifies the pixel height for trace events. | `int` | 20 |
| `jack.tracing.idisplay.headheight` | Specifies the pixel height for the source side indicator. | `int` | 20 |
| `jack.tracing.idisplay.details` | Traces all events, not just the top level ones. | `boolean` | |

**Table A-5:** Interaction Diagram Properties

# Plan Tracing Tool Properties

The following properties can be used to alter the default behaviour of the Plan Tracing Tool. To use these properties, the Plan Tracing Tool must first be enabled in the JDE or by setting the `jack.plan.tracing.enabled` property to `true`.

| Property | Description | Type | Default |
|---|---|---|---|
| `jack.plan.tracing.alwaysraise` | Always bring the current task window of the Plan Tracing Tool to the front of all windows whenever any node in a graph it shows is traced. | `boolean` | `false` |
| `jack.plan.tracing.alwaysrestore` | Always restore a minimised Plan Tracing Tool task window when any node in a graph it shows is traced. | `boolean` | `true` |
| `jack.plan.tracing.config` | Specifies the configuration file to be used by the Plan Tracing Tool. | `String` | `null` |
| `jack.plan.tracing.descriptivemode` | Starts the Plan Tracing Tool in Descriptive Mode. | `boolean` | `false` |
| `jack.plan.tracing.enabled` | Enables the Plan Tracing Tool. | `boolean` | `false` |
| `jack.plan.tracing.listallfacts` | Displays all the facts that are contained in the beliefsets that appear in the Variables section of a Plan Tracing Tool task window. If this property is set to `false`, only the number of facts is displayed. | `boolean` | `false` |
| `jack.plan.tracing.runmode` | Puts the Plan Tracing Tool into Run mode whenever a new agent is created. | `boolean` | `false` |

| Property | Description | Type | Default |
|---|---|---|---|
| `jack.plan.tracing.tracetextual` | Traces textual reasoning methods as well as graphical reasoning methods. | `boolean` | `true` |
| `jack.plan.tracing.zoomcombo.show` | Shows the zoom combo box in task windows of the Plan Tracing Tool. | `boolean` | `false` |
| `jack.plan.tracing.zoomslider.show` | Shows the zoom slider in task windows of the Plan Tracing Tool. | `boolean` | `true` |
| `jack.plan.tracing.zoomslider.showtickmarks` | Shows tick marks on the zoom slider. | `boolean` | `true` |

**Table A-6:** Plan Tracing Tool Properties

# Index


**A**
agent behaviour debugging 11, 71
agent debugging
  running 74
Agent Interaction Diagram 59
Agent Manual 11, 37, 71
agent tracing controller 52
agent tracing controller window 41
AgentDebuggerCommand interface 71, 72
applicable plans 37, 44
apply design tracing configuration 21

**B**
buttons
  add design trace row 18
  add trace row 20
  apply 21
  compile 30
  load design tracing configuration 20
  remove trace row 18
  reset links count 26, 27
  run tracing 23
  save design tracing configuration 20
  select agent type 19
  select design diagram 19
  step tracing 23
  stop tracing 23
  stop/continue design tracing 23, 26

**C**
close portal 23
command line arguments
  DCI 13, 14
compile application tab 30
compile button 30
compiler utility 30
configure design tracing 17
configure design tracing option 16
configuring an interaction diagram 62
connect to nameserver option 15
connect to portal 12
connect to portal option 16, 31
control design tracing 21
control design tracing option 16

**D**
DCI 12
DCI command line arguments 13, 14
debug object 73
  user defined 73
debug.setError 78
debug.setInput 78
debug.setOutput 78
debugging
  agent 71
  generic 71
  running 74
delete trace row 18
design and plan tracing 12
design graph 26
design trace control 22
design tracing
  add trace row 18
  agent types 19
  all agents 20
  close portal 23
  configuration error 21
  configure 17, 32
  control 21, 22, 23, 33
  delay transitions 22
  descriptive mode 22, 26
  edit trace row 18
  example 27
  global trace settings 22
  individual agents 20
  load configuration 20
  no windows shown 27
  reconfigure 24
  relevant agents 26
  relevant tasks 27, 35
  remove trace row 18
  reset links count 26, 27
  run 23
  save configuration 20

select agent type 19
select design diagram 19
step 23
stop 23
stop/continue 26
TDW control bar 26
TDW tool bar 26
transitions 27
turn on 22
visualisation 22
visualisation error 27
zoom in/out 26
design tracing configuration window 17
design tracing controller 16, 21
design tracing java argument 13
design tracing tool 11
Development Environment 11
development environment 12
Development Environment Manual 11, 37
DTT 11
DumpState 72, 73

**E**
enabling the agent interaction diagram 59
event tracing 37, 40

**F**
failed plans 37, 44
fields
Java Args 30

**G**
generic debugging 71
running 74
global trace settings 22
Graphical Plan Editor 37
Graphical Plan Editor Manual 37

**I**
interaction diagram
configuring 62
enabling 59

**J**
JACK kernel 71
jack.args 76
jack.compiler.emit.imports 76
jack.compiler.errors 76
jack.debugger.command 71
jack.debugger.commands 78
jack.debugger.port 71, 78
jack.plan.tracing.alwaysraise 80
jack.plan.tracing.alwaysrestore 80
jack.plan.tracing.config 80
jack.plan.tracing.descriptivemode 80
jack.plan.tracing.enabled 80
jack.plan.tracing.listallfacts 80
jack.plan.tracing.runmode 80
jack.plan.tracing.tracetextual 81
jack.plan.tracing.zoomcombo.show 81
jack.plan.tracing.zoomslider.show 81
jack.plan.tracing.zoomslider.showtick-marks 81
jack.portal.host 76
jack.portal.name 76
jack.portal.port 76
jack.property.file 77
jack.run.debug.agents 78
jack.run.debug.options 78
jack.run.debug.show.count 78
jack.run.nthreads 77
jack.run.repeatable 77
jack.run.timeslice 77
jack.tracing.enabled 78
JACOB dictionary file 71, 73
JACOB Manual 71, 74
Java Args field 30
Java arguments
design tracing tool 13
plan tracing tool 55
Java portal properties 12, 13, 30
JDE 11, 12, 15, 37

**L**
load design tracing configuration 20

**M**
menus
trace 15, 17, 31
message 60
method
process 71, 72

**O**
options
configure design tracing 16
connect to nameserver 15
connect to portal 16, 31
control design tracing 16
create project from example 38, 44
ping agent 17
quit tracing on portal 17

**P**
ping agent option 17
plan and event tracing
configuration file 53
plan and event tracing configuration file 44
default 54
plan tracing
applicable plans 37, 44
configuration file 44, 52
default 54
events 37, 40
failed plans 37, 44
variables 42
plan tracing configuration file 52, 53
plan tracing tool 12, 37
runtime options 55
process method 71, 72

**R**
reconfigure design tracing 24
relevant agents 26
relevant tasks 27
run application tab 30
runtime options 55

**S**
save design tracing configuration 20

**T**
tabs
compile application 30
run application 30
task window 42
TDW 11, 24
trace designs 11
trace menu 15, 17, 31
traced design window 11, 24
tracing designs and plans 12
tracing events 40
tracing portal window 17, 31
transition delay 52

**V**
viewing documentation 26

**W**
windows
compile 30
design tracing configuration 17
error applying configuration 21
tracing 17, 31