Reference Manual 3.0 Beta

Nasa Ames Research Center
Moffett Field, CA 94035

© 2005 NASA Ames Research Center

All trademarks are the property of their respective owners.

Printed in the United States of America.
Edition: April 2005

NASA Ames Research Center
Moffett Field, CA 94035
(650) 604-5000

apexhelp@eos.arc.nasa.gov

http://human-factors.arc.nasa.gov/apex

Table of contents

1 Introduction ... 6
1.1 Obtaining more information ... 6
1.2 Getting the latest information ... 7
1.3 Using this manual ... 7
1.4 Conventions ... 8
2 Using Apex ... 9
2.1 Introduction to the Sherpa workspace ... 9
2.1.1 The object tree ... 9
2.1.2 The view panel ... 10
2.2 Starting Apex ... 10
2.3 Loading an application ... 11
2.4 Running an application ... 11
2.5 Working with event traces ... 11
2.5.1 Filtering an event trace ... 13
2.5.2 Other event trace features ... 14
2.6 Inspecting objects ... 14
2.7 Viewing agendas ... 15
2.8 Viewing diagrams ... 16
2.9 Viewing monitors ... 16
2.10 Viewing state variables ... 17
2.10.1 Exporting state variable information ... 18
2.11 Using the PDL view ... 18
2.12 Using primary views ... 19
2.13 Moving back and forth through the view history ... 19
2.14 Setting general preferences ... 19
2.14.1 Setting the look and feel ... 19
2.14.2 Setting the font ... 19
2.15 Searching ... 19
2.16 Sherpa troubleshooting ... 20
2.17 System patches ... 20
2.18 Getting help ... 20
3 Creating Apex Applications ... 22
3.1 Lisp programming and Emacs ... 22
3.2 Application Definition File ... 22
3.3 Application files ... 23
3.4 Libraries ... 23
3.4.1 Using libraries ... 23
3.4.2 Creating libraries ... 24
3.4.3 Finding libraries ... 24
3.4.4 Provided libraries ... 24
4 Procedure Description Language ... 25
4.1 Introduction ... 25
4.1.1 Action Selection Architecture ... 26
4.2 Basics ... 29
4.2.1 Introduction to PDL variables ... 29
4.2.2 Introduction to PDL time representation ... 29
4.2.3 Notations and conventions ... 30

4.3 Procedures ... 31
4.3.1 index ... 32
4.3.2 proctype ... 33
4.3.3 profile ... 33
4.3.4 log ... 35
4.3.5 expected-duration ... 36
4.3.6 terminate/when ... 36
4.3.7 interrupt-cost ... 36
4.3.8 on-start ... 36
4.3.9 on-end ... 37
4.4 Steps ... 37
4.5 Monitoring overview ... 37
4.5.1 waitfor ... 38
4.5.2 terminate/when ... 38
4.5.3 suspend/when/until ... 39
4.5.4 restart/when ... 39
4.5.5 resume/when ... 39
4.5.6 select ... 40
4.6 Task control overview ... 40
4.6.1 Priority ... 40
4.6.2 Rank ... 41
4.6.3 interrupt-cost ... 41
4.7 Repetition, policies, and iteration ... 42
4.7.1 repeating ... 42
4.7.2 responding ... 43
4.7.3 forall ... 43
4.7.4 period ... 44
4.8 Conditions ... 44
4.8.1 Monitoring state variables ... 45
4.8.2 Times associated with monitors ... 46
4.8.3 :measurement ... 47
4.8.4 :estimation ... 48
4.8.5 :episode ... 50
4.8.6 :and ... 54
4.8.7 :or ... 54
4.8.8 :not ... 54
4.8.9 :in-order ... 54
4.8.10 Allen monitors ... 55
4.8.11 :delay ... 55
4.8.12 :timestamp ... 55
4.9 Primitive procedures ... 57
4.9.1 primitive ... 57
4.9.2 duration ... 57
4.9.3 update ... 58
4.9.4 on-start ... 58
4.9.5 on-completion ... 58
4.9.6 locals ... 58
4.9.7 profile ... 58
4.9.8 return ... 59
4.10 Miscellaneous features ... 59
4.10.1 Agent's initial task ... 59
4.10.2 PDL Partitions (Bundles) ... 59
5 Apex Programming Reference ... 61
5.1 defapplication ... 61
5.2 Application interface ... 62
5.3 Time and Scheduled Events ... 63

5.4 Agents ... 63
5.5 Diagrams ... 65
5.5.1 Hierarchical diagrams ... 66
5.5.2 Wireframe diagrams ... 67
5.5.3 File-based SVG diagrams ... 68
Glossary ... 70
Appendix A: Getting Started with Apex ... 71
Overview of third party software packages ... 71
Java Runtime Environment ® (JRE) ... 71
Graphviz® ... 71
Emacs ... 71
Getting started on Macintosh ... 72
Installing the software on Macintosh ... 72
Starting Apex on Macintosh ... 72
Getting started on Windows ... 72
Installing the software on Windows ... 73
Starting Apex on Windows ... 73
Getting started on Linux ... 73
Installing the software on Linux ... 73
Starting Apex on Linux ... 74
Appendix B: Using the Lisp Listener ... 75
Loading an application ... 75
Running an application ... 76
Working with event traces ... 76
Using the prompt-based interface ... 77
Appendix C: Event Traces ... 78
Predefined show levels ... 78
Lisp commands for controlling trace output ... 78
Trace constraint syntax ... 78
Appendix D: PDL Syntax ... 80
Procedure level clauses ... 80
Step level clauses ... 80
Primitive Procedures ... 81
Monitor conditions ... 81
Measurement conditions ... 81
Estimated conditions ... 82
Simple episodic conditions ... 82
Complex conditions ... 83
Time-based conditions ... 83
Atomic episode conditions ... 84
Appendix E: Diagram Programming Reference ... 85
Labeling graphical objects ... 85
Example ... 85
Wireframe diagrams ... 86
SVG diagrams ... 86
Appendix F: Time and Date Formats ... 87
Appendix G: Troubleshooting Tips ... 89
Appendix H: Pattern Matching ... 90
Appendix I: Application Definition File Example ... 93
Appendix J: Starting Apex within Allegro Common Lisp ... 95

1 Introduction

Apex is a toolkit for constructing software that behaves intelligently and responsively in demanding task environments. Reflecting its origin at NASA where Apex continues to be developed, current applications include:

- Providing autonomous mission management and tactical control capabilities for unmanned aerial vehicles including an autonomous surveillance helicopter and a simulation prototype of an unmanned fixed-wing aircraft to be used for wildfire mapping

- Simulating human air traffic controllers, pilots and astronauts to help predict how people might respond to changes in equipment or procedures

- Predicting the precise duration and sequence of routine human behaviors based on a human-computer interaction engineering technique called CPM-GOMS

Among Apex's components are a set of implemented reasoning services, such as those for reactive planning and temporal pattern recognition; a software architecture that embeds and integrates these services and allows additional reasoning elements to be added as extensions; a formal language for specifying agent knowledge; a simulation environment to facilitate prototyping and analysis; and Sherpa, a set of tools for visualizing autonomy logic and runtime behavior. In combination, these are meant to provide a flexible and usable framework for creating, testing, and deploying intelligent agent software.

Overall, our goal in developing Apex is to lower economic barriers to developing intelligent software agents. New ideas about how to extend or modify the system are evaluated in terms of their impact in reducing the *time, expertise, and inventiveness* required to build and maintain applications. For example, potential enhancements to the AI reasoning capabilities in the system are reviewed not only for usefulness and distinctiveness, but also for their impact on the readability and general usability of Apex's behavior representation language (PDL) and on the transparency of resulting behavior.

A second central part of our approach is to iteratively refine Apex based on lessons learned from as diverse a set of applications as possible. Many applications have been developed by users outside the core development team including engineers, researchers, and students. Usability is thus a central concern for every aspect of Apex visible to a user, including PDL, Sherpa, the Apex installation process, APIs, and user documentation.

Apex users vary in their areas of expertise and in their familiarity with autonomy technology. Focusing on usability, a development philosophy summarized by the project motto "Usable Autonomy," has been important part of enabling diverse users to employ Apex successfully and to provide feedback needed to guide iterative, user-centered refinement.

1.1 Obtaining more information

More information about Apex is available on the Apex website, http://human-factors.arc.nasa.gov/apex. The website includes links to papers which describe many aspects of Apex.

Extending and developing applications in Apex requires programming in Common Lisp. For more information about Common Lisp, consult a reference book, such as Common Lisp by Guy Steele. The complete text of Common Lisp is available online at the following URL:

http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/ai/html/cltl/cltl2.html

1.2 Getting the latest information

For the current version of Apex and this document, visit http://human-factors.arc.nasa.gov/apex. To check whether you have the latest version of this document, check the publication date on the copyrights page.

To report a bug or consult on a technical problem, contact the Apex development team at apexhelp@eos.arc.nasa.gov.

For information related to the development of the Apex system, send email to mfreed@arc.nasa.gov.

1.3 Using this manual

This manual provides information about using Apex and building Apex applications.

- Appendix A describes how to get started with Apex including how to download and install the system.

- Chapter 2 covers the basics of using Apex via its graphical user interface Sherpa.

- Chapters 3 and 4 describe how to build an application with Chapter 4 focused specifically on defining the behavior of intelligent software agents using Apex's PDL notation. A brief tutorial on building applications called "Paper Covers Rock" is also available and may be downloaded from the Apex website along with technical papers, presentations, and other material.

- Chapter 5 describes Apex software in more detail, as needed to modify and extend system behavior or cause Apex to interoperate with other systems.

1.4 Conventions

To make this manual easier to read, the following typography conventions have been adopted.

Code examples appear in 9 pt Courier. For example,

```
(procedure
(index (start-engine))
(step s1 (turn-key))
```

Code of particular importance appears in bold 9 pt Courier. The `defapplication` form is the main focus of this example:

```
(defapplication "Hello World"
     init-sim (hello-world))
```

User-specified entries in code appear in chevrons (< >) and sometimes have a qualifying statement following. For example,

```
(defapplication <application-name>
    :init-sim (<initialization-function>))
```

where `<application-name>` is a string and `<initialization-function>` is the name of the function to run to start the simulation.

In this case, the actual code a user enters would look something like the following:

```
(defapplication "Hello World"
    :init-sim (<hello-world>))
```

2 Using Apex

This chapter describes how to use Apex via its graphical user interface, Sherpa. Sherpa provides support for creating, debugging, demonstrating, and analyzing Apex applications. Sherpa also allows you to obtain graphical output from application runs, such as event traces, diagrams, and PERT charts.

Apex can instead be used through a Lisp interactive window, known as a Listener. See Appendix B for information on using the Listener.

For requirements and instructions for installing and setting up Apex and Sherpa, see Appendix A.

2.1 Introduction to the Sherpa workspace

The Sherpa workspace contains tools for controlling Apex applications, changing between views, and navigating through the view history, as well as a status bar that displays information about the application state.

The Sherpa main window is composed of two panels: the *object tree* and the *view panel*.

- The object tree lists the scenario objects from an application in a collapsible hierarchical fashion.
- The view panel provides access to a number of views to help in understanding Apex applications.

2.1.1 The object tree

The object tree lists all objects associated with an application in an object hierarchy. The object tree root node is the application object. The immediate children of the application object are locales representing the places where application agents reside. The following top-level entities are included in locales:

- Agents—entities that use the Apex action selection architecture to generate behavior. Each agent has an agenda, a monitor array, a procedure set, and a set of resources.
- Non-agent entities—these entities contain their parts.
- Interface objects— entities that mediate information exchange within or across locales as well as between an Apex application and a foreign application.

The object tree also contains any routers used in the application.

Each item in the object tree is identified by its object type along with an ID number using the following format: *type-ID*. When you select an item from the object tree, that item becomes the *focal object*. You can also specify the focal object by typing the object ID in the Focal Object field or by clicking on a representation of an object from within a view.

The focal object provides context for information displayed in the view panels. Depending on the type of the focal object, some toolbar controls may be disabled. For example, if you select a locale as the focal

object, the Agenda and PDL toolbar buttons become disabled since those views are not meaningful in the context of a locale.

You can select multiple objects from the object tree using shift-click or control-click. Note, however, that some views do not support multiple foci.

2.1.2 The view panel

The view panel provides a number of views for examining the application focal object. From the main window, you can launch multiple views at a given time by floating the view panel. Each of these views shares a common focal object; changing the focal object in any view causes it to change in all others.

- The Trace View displays the activities of Apex agents and other entities (if any) as they occur while the application runs
- The Inspect View provides easy access to internal representations of all objects and is useful for debugging.
- The Agenda View provides detailed information about tasks in the agent's currently active tasks, including task states, monitor states, resources used, as well as time information.
- The PDL View lists the procedures for an agent. You can view this list in a variety of ways, such as alphabetically or by bundle. The PDL view also allows you to examine the code for each procedure.
- The Diagram View allows you to examine diagrams associated with agents or objects. By default, Apex displays a tree diagram for an object; however, other types such as imaged maps diagrams are also supported.
- The Monitor View provides a graphical display of the status of a monitor during an application run: whether it is satisfied, partially satisfied, unsatisfied, or unchecked.
- The State Variable View allows you to examine the values of state variables.

These views are described in greater detail later in this chapter.

2.2 Starting Apex

Instructions for starting and stopping Apex and Sherpa vary by operating system and can be found in Appendix A.

After you start Apex and Sherpa, Sherpa attempts to connect to an Apex server running on the same computer or to the Apex server that Sherpa was last connected to. If you want to connect to a different Apex server:

1. Select Set Server from the File menu.
2. Enter the address of the computer, and click OK.
3. Select Reset Sherpa from the File menu.

2.3 Loading an application

Before you can run an Apex application, you must first load its Application Definition File. You can load an application using one of the following methods:

- **Select from a list of recently loaded applications**—From the File menu, select Recent Applications, then choose the application you want to open.

- **Browse files and select an application from your local file system**—From the File menu, select Load Application, navigate to the appropriate folder, select the file, then click Open.[1]

2.4 Running an application

Once an application is loaded, you can manipulate it in the following ways:

- **Start the application**—Click the Run button. The application runs to completion. If the Trace View is open, events are seen as they occur in the running application (other views do not dynamically update).

- **Pause a running application**—Click the Pause button, if it is selectable (pausing is not supported in all applications). There are other ways to pause an application – see the Preprogrammed Pause section below.

- **Step the application**—Click the Step button, if it is selectable (not all applications support stepping). By default, a step is the advancing of the application by one *cycle*, as defined by the application. In a native simulation application, this is one simulation engine cycle. In a realtime application, a step is defined by the `:step` clause of its `defapplication` form. However, if settings are made in the Step Dialog (brought up by clicking on the triangle icon next to the Step button), then Step will effect a *preprogrammed pause* (see next paragraph).

- **Preprogrammed Pause (Step Dialog).** It is possible to have an application pause automatically at a specific timepoint, pause at regular time intervals, or pause at regular cycles. These settings are made using the Step Dialog, which is brought up by clicking on the triangle icon next to the Step button. Only one setting can be made at a time. When a setting is entered, the Step button will cause the application to run until the next specified pause point.

- **Reset the application**—Click the Reset button to restore the application to its initial state.

- **Reload the application**—Click the Reload button to reload the application.

2.5 Working with event traces

An event trace displays the activities of Apex agents and other entities (if any) as they occur while the application runs and also allows users to examine these events retrospectively. Event traces are available from the Trace View. To access the Trace View, select one or more objects in the object tree and click the Trace button. Alternately, you can right-click on an object in the tree or within a view and select Trace Object from the context menu. Figure 1 shows an example event trace.

---

[1] Currently this feature does not work if Sherpa and Apex are running on different computers.

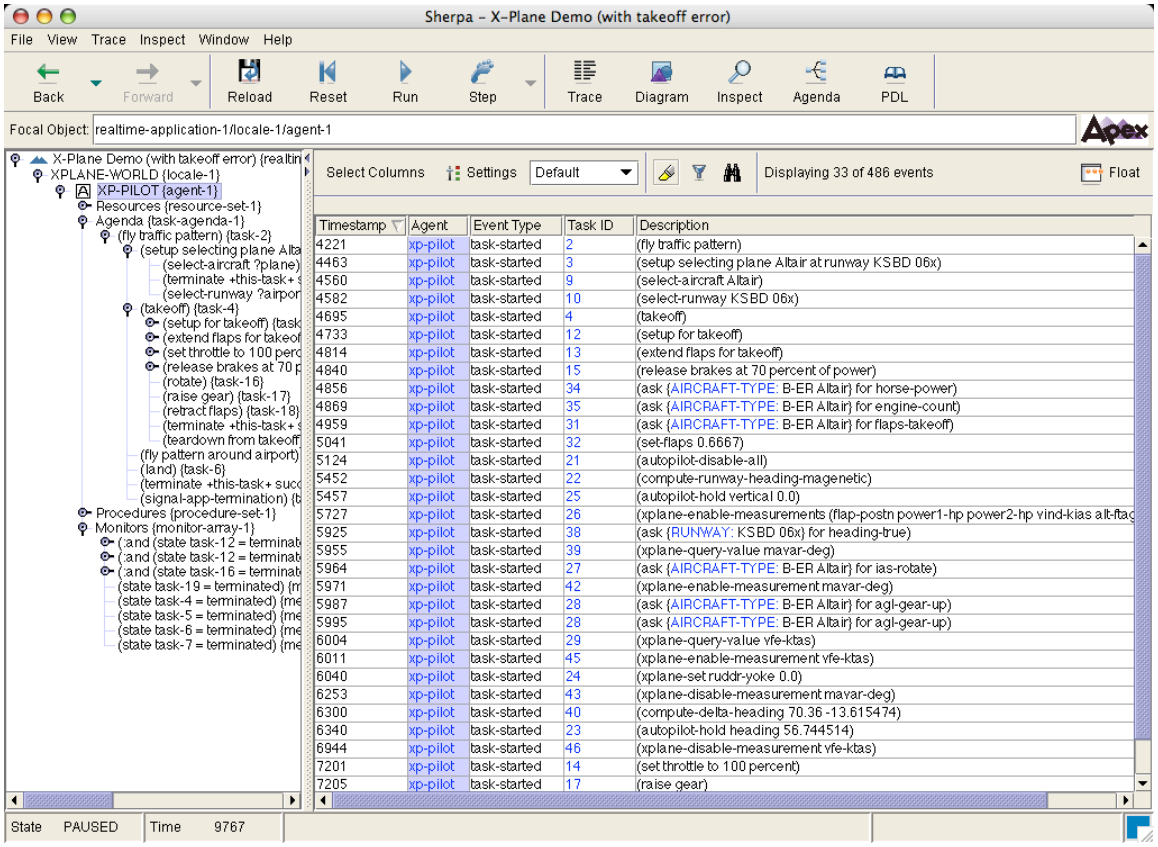


Figure 1: Trace View

The Trace View shows event information in which the focal object is mentioned. For instance, the objects task-25, hand-1, and rock-1 are mentioned in the following event:

```
(interrupted (task-25 (grasp rock-1 with hand-1))
```

An object is considered indirectly mentioned in any event that directly mentions one of its descendants in the object tree. For instance, if the focal object is agent-1 and hand-1 is a component of agent-1, then an event that directly mentions hand-1 is considered to have indirectly mentioned agent-1.

Each row in the event trace represents an event. For example, the following row:

```
4235  Fred  task-started  10  (sign-in)
```

represents that at time 4235 the agent Fred began task number 10 to “sign in.” By default, time is measured in milliseconds after the start of the Apex application run. In simulations, this time indicates the simulated time (that is, the time in the chronological frame of the simulation not in the real world).

An Apex application typically generates a large event trace. You can filter the data in the event trace to obtain a smaller, meaningful subset.

2.5.1 Filtering an event trace

You can specify filter criteria to reduce the amount of trace information displayed. You can apply filtering criteria to trace data displayed at runtime as well as traces derived from the stored event history. If events are not displayed during a run due to filtering, you can still view them after the run or while the application is paused.

You can apply filters to event traces from the Trace Filter dialog box. To open the dialog box, click Settings to open the Trace View.

Events are most often filtered by event type. You can filter event traces by event type in the following ways:

- **Specifying a show level in the pull-down menu**—A show level is a predefined collection of event types to be shown. Predefined show levels are described in Appendix C. By default, the Trace View shows a subset of the events for an application run. You can view a trace in its entirety by selecting All in the Show Level menu. Note, however, that this trace may contain thousands of events or more.

- **Selecting specific event types of interest**—Event types associated with the currently loaded application are displayed next to checkboxes in the Trace Filter dialog box. Click the checkboxes to toggle whether or not to include a particular event type. Note that selecting or deselecting event types modifies the choices associated with the previous show-level, though that show-level is still displayed on the interface. Predefined event types are listed in Appendix C.

Traces can also be filtered by other criteria, such as time range, object, and procedure, in the Trace Filter dialog box. The filtering criteria are cumulative.

- **Time range**—Select a time format, then enter the start and end times.

- **Object**—Enter the object ID next to Filter Objects and click Add. You can also include all of the object's descendants in the filer by clicking the checkbox next to the object ID. You can remove an object from a filter by selecting the object ID in the Filter Objects list and pressing the Delete key. For example, to add

- **Procedure**—Under Filter by Procedure, enter the first word of the index clause for the procedure. Separate multiple entries with blank spaces.

After specifying a filter, you can choose how it is applied.

- **Apply and Show**—Applies the filter and redisplays the event trace with the filter applied.

- **Apply and Save**—Applies the filter and saves the resulting event trace to file.

- **Apply Filter**—Displays the number of events in the event trace with the new filter applied, but does not redisplay the event trace. This feature is particularly useful when working with the large event traces. If you decide to view the event trace, click Apply and Show.

- **Set Filter**—Sets the filter for the event trace. The next time that you view an event trace the filter is applied.

To make a filter persistent, select Persist Event Filter from the Trace menu. Each time you access a new Trace View, Sherpa applies the persistent filter.

The Trace View also provides tools for highlighting, filtering by object, and finding instances of an object.

- **Highlighting all events for an object**—To highlight all events for a particular object, click the Highlight icon in the Trace View, then click the object to highlight.

- **Filtering by object**—To filter the event trace by object, click the Filter icon in the Trace View, then click on an agent, task ID, or procedure ID. The event trace is updated with events only for that object.

- **Finding next instance**—To find the next instance of an object in the event trace, click the Find icon in the Trace View, then click on the object you want to find the next instance of.

You can also filter the Trace View by right-clicking on an item in the object tree and choosing one of the following options:

- Trace Object—equivalent to filter by object

- Trace Object Subtree—equivalent to filtering by object and setting the checkbox to include descendants

- Trace Highlight—equivalent to highlight object

2.5.2 Other event trace features

Sherpa also provides the following features for working with event traces:

- **Sorting an event trace**—You can sort an event trace by clicking on a column name. The first click sorts the column in alphabetical or numerical order. Clicking the column name again sorts in the reverse order.

- **Setting event trace to verbose**—If you set the event trace to verbose, the event trace will include any print statements in the application code. Setting the event trace to verbose can help with debugging.

- **Redirecting the event trace to the Listener**—The Trace View has a limited scroll size. It is possible to redirect trace output to the Listener by selecting Trace To Listener from the Trace menu.

- **Saving the event trace to a file**—You can also save trace output to a file by selecting Trace to File from the Trace menu. Navigate to the desired location, enter a file name, then click Save.

2.6 Inspecting objects

You view information about objects from the Inspect View. To access the Inspect View, select one or more objects in the object tree, then click the Inspect button. Alternately, you can right-click on an object in the tree or within a view and select Inspect from the context menu.

The Inspect View provides easy access to internal representations of objects and is useful for debugging. You can simplify the display by hiding null values or get detailed information by selecting the verbose option.

Within the Inspect View, you can click on any object representation (in blue text) to change the focal object.

2.7 Viewing agendas

The Agenda View provides detailed information about tasks in the agent's agenda, including task states, monitor states, resources used, as well as time information. You can view the agenda for an agent during an application run by selecting an agent or one of its children in the object tree, then clicking the Agenda button. Alternately, you can right-click on an agent or one of its children in the tree or within a view and select Agenda from the context menu. Figure 2 shows an example of the Agenda View.

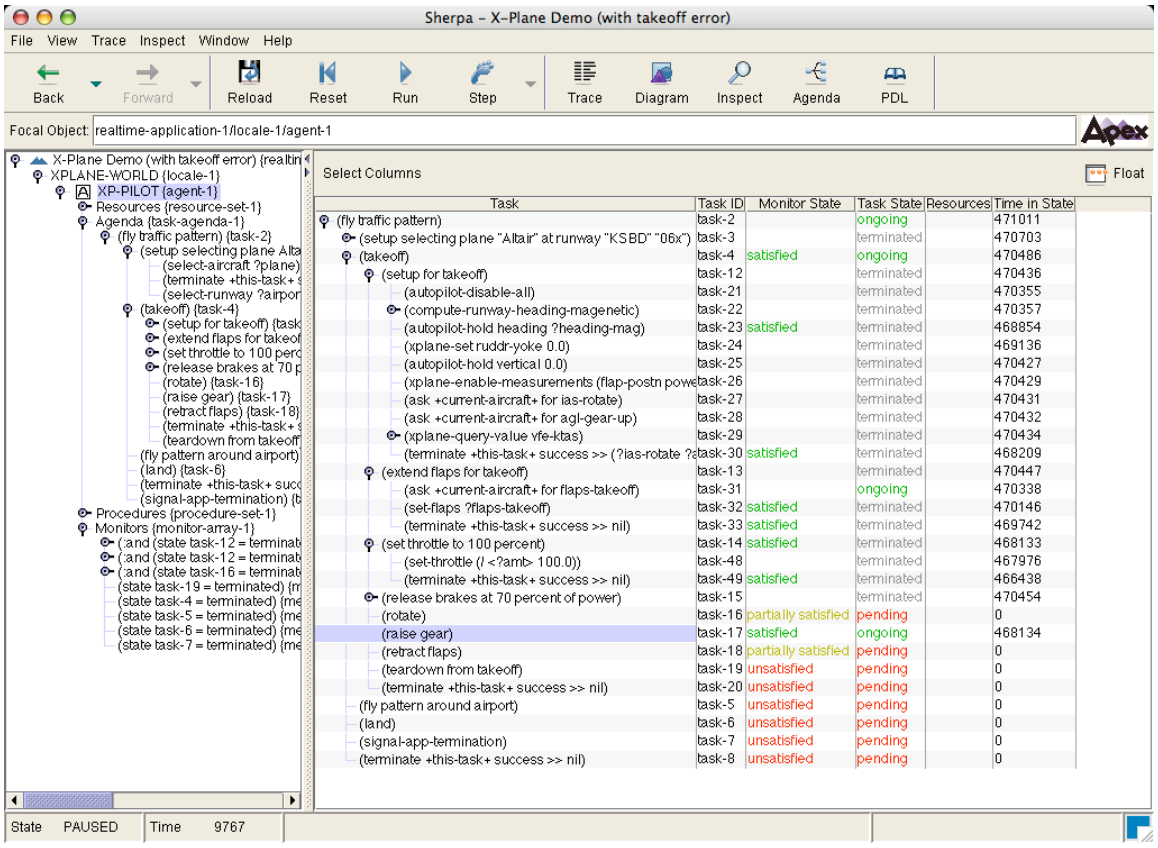


Figure 2: Agenda View

As you single step an application, you can watch the task states and monitor states update.

- Each task in the agenda is in one of the following states: ongoing, terminated, and pending
- Each monitor is in a satisfied, partially satisfied, or unsatisfied state

You can access monitor information from the Agenda view in the following ways:

- Mousing over a monitor status within the Agenda View displays a summary of that monitor.
- Clicking on a monitor status opens the Monitor View for that monitor.
- Right-clicking on any monitor state in the agenda to view more information about the monitor from one of the following views: Monitor, Diagram, Inspect, or Trace.

You can customize the agenda view by choosing which columns to display: click Select Columns and choose the appropriate columns.

2.8 Viewing diagrams

You can view diagrams associated with agents or objects from the Diagram View. You can access the Diagram View by selecting an item in the object tree and clicking the Diagram button.

Apex supports the following types of diagrams.

- **Tree diagram**—Displays the focal object and its descendents. The Tree Diagram is displayed by default.
- **SVG diagram**—Displays an image-mapped scalable vector graphic if available.
- **XY, YZ, and ZX diagrams**—Display an image-mapped two-dimensional drawing if available.

The SVG and two-dimensional diagrams are image mapped to objects that facilitate demonstration and exploration of application models. You can zoom in and out on these diagrams as well as print them.

If you click on an image hotspot, the Diagram View is updated with a new focal object. If you right-click on an image hotspot, you can choose to access other views with the new focal object.

For information about adding diagrams to applications, see Chapter 5.

2.9 Viewing monitors

You can view monitor information for an agent in a diagram format from the Monitor View. To access the Monitor View, right-click on a monitor in the object tree during an application run and select Monitor Diagram. You can also view a monitor diagram from the Agenda View, by right-clicking on a monitor status and selecting Monitor Diagram from the context menu.

The Monitor View provides information about the status of a monitor: whether is satisfied, unsatisfied, or unchecked. In the case of complex monitors, monitors can also be partially satisfied. If you step an application from the Monitor View, you can watch monitor status change as the application proceeds. Figure 3 shows an example of the Monitor View.

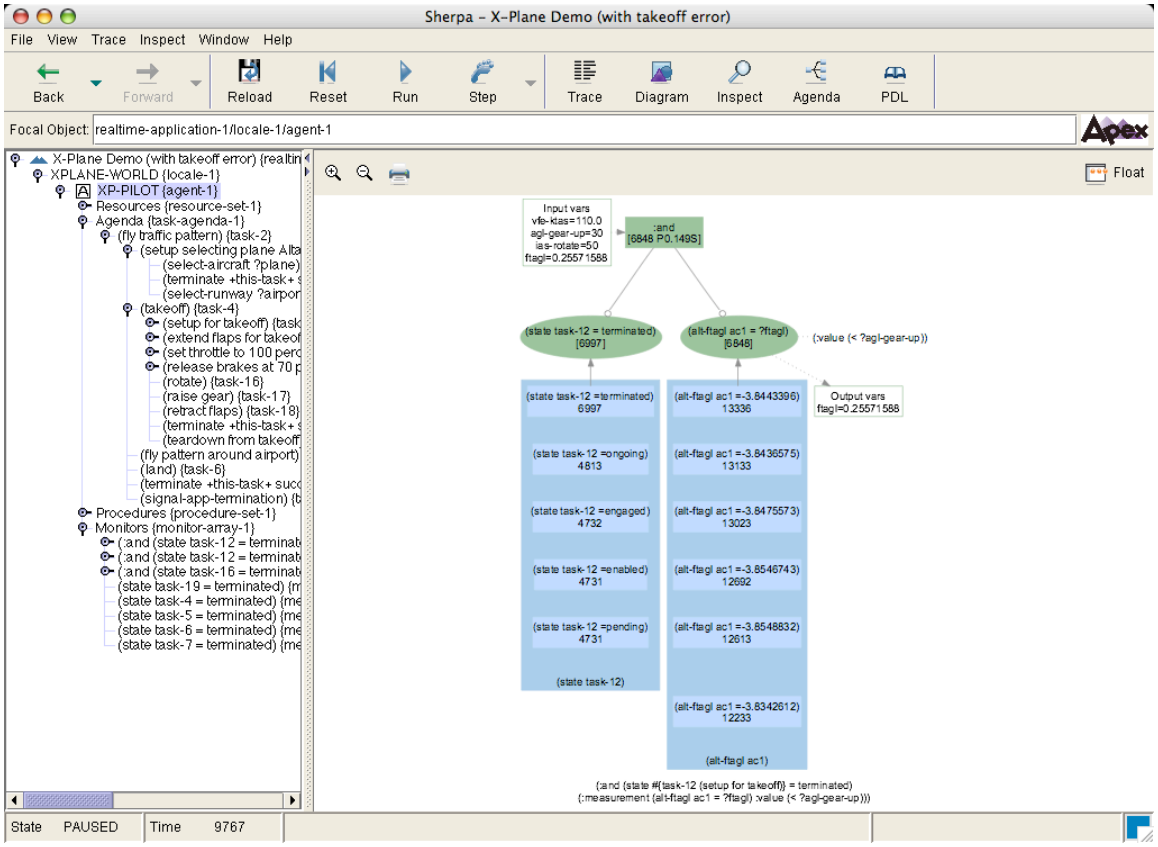


Figure 3: Monitor View

Monitors are color coded as follows:

- Unsatisfied—red
- Partially satisfied—yellow
- Satisfied—red
- Unchecked—grey

You can click a monitor to view detailed information about that monitor in the Inspect View.

2.10 Viewing state variables

You can view the values of state variables from the State Variables View. To access the State Variables View, right-click on an agent or any of its descendants in the object tree and select State Variables.

A state variable is an attribute of an object (often the agent, or a component of the agent), which may change over time, for example, the altitude, longitude, and latitude of an aircraft.

You can choose whether or not to include state variables for tasks, if you choose not to only object state variables, if any, are displayed.

2.10.1 Exporting state variable information

You can export state variable information, by clicking Export to CSV, choosing a location and name for the file, then clicking Save

2.11 Using the PDL view

The PDL View lists the procedures for an agent. You can access the PDL View by selecting an agent in the object tree and clicking the PDL button.

From the PDL View, you can view procedures in the following ways:

- **Call graph**—Lists the procedures for an agent in the order they are called within the application
- **Alphabetical list**—Lists the procedures for an agent in alphabetical order
- **Bundle list**—Lists the procedures for an agent by bundle in alphabetical order

In the alphabetical list and the bundle list, a blue icon indicates a primitive, while an orange icon indicates a procedure. Figure 4 shows an example of the PDL View.

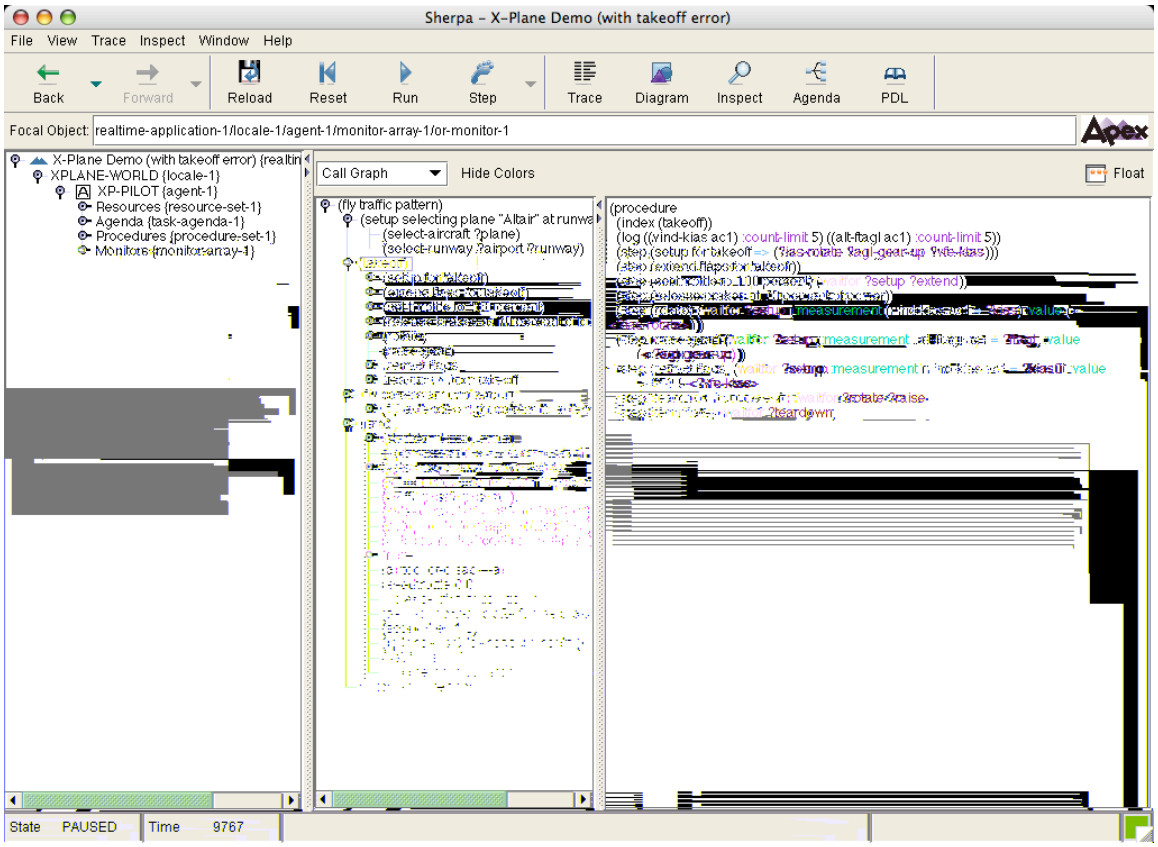


Figure 4: PDL View

To view the code for a primitive or procedure, click the name in the list in the left panel, the code appears in the right panel. You can choose whether to highlight PDL constructs in the code, such as variables and keywords, by toggling Show Colors and Hide Colors.

2.12 Using primary views

Unlike floated views, a primary view is a main window that has its own focal object. Each primary view can launch any number of floated views. Each floated view shares the focal object of the primary view from which it was launched.

Note, however, while different primary views may have different focal objects, all primary views share the same underlying application state. In other words, pausing an application in one primary view causes the application to pause in all other primary views.

You can open a new primary view by typing Control-n. Note that the new primary view contains the view history of its parent.

2.13 Moving back and forth through the view history

You can use the navigation tools to move back and forth through the view history. The view history includes events such view changes and filtering. Note that the navigation tools do not apply to the application state, in other words clicking the Back button does not allow you to go back a step in an application run.

2.14 Setting general preferences

You can set general preferences, such as setting the look and feel and setting the font, from the View menu. You can reload the default general preferences by selecting Reload Preferences from the View menu.

2.14.1 Setting the look and feel

Sherpa allows you to set the look and feel by selecting from a list of styles. A style does not change Sherpa functionality, but rather defines how the user interface looks.

To set the look and feel, select Look & Feel from the View menu, then choose a style.

2.14.2 Setting the font

You can change the font for text displayed within the View panel.

Select Font Size from the View menu, then select a font, the size, and whether the font should be bold or italic, then click OK

2.15 Searching

Sherpa allows you to search for text in the view panel. Note that not all views support searching.

To search for text in a view, select Find from the View menu. Enter the text you would like to find, specify a search direction (up or down), and click Find Next. You can optionally specify if you want to match the whole word or if you would like to make the search case sensitive.

2.16 Sherpa troubleshooting

The Debug window provides detailed information useful for debugging problems in Sherpa. You can access the Debug window from the Window menu.

The Debug window displays the raw data sent between Apex and Sherpa. You can set the level of detail for this data by selecting a Debug Level; however, this manual does not discuss the details of these levels since this functionality is mainly intended for the Apex support staff.

The Debug window also allows you to instruct the Lisp system to perform garbage collection. Garbage collection is the process of reclaiming memory that is no longer accessible, thereby improving system performance.

You can view details about the software systems underlying Sherpa, such as Java, by clicking the System Properties button.

You can clear the content of the Debug window by clicking the Clear Output button.

2.17 System patches

Patches provide extensions, modifications or fixes to the existing Apex software without requiring reinstallation. You can obtain patches from the Apex web site:

http://human-factors.arc.nasa.gov/apex

The exact URL for patches is not known at the time of this writing, but you will be able to find it easily. Instructions for downloading and installing patches will be available on the website, but the following is a synopsis of the process.

- Download all of the .lisp files available and put them in your apex:patches directory.
- Delete any patch files with the same name, including any compiled versions (for example, those ending in .fasl). Newly installed patches will automatically be in effect the next time you start Apex.

If you want to install the patches without restarting Apex, type `(load-apex-patches)` at the Listener prompt. A brief description of each patch is found in the file.

2.18 Getting help

If you experience problems with Apex, please consult the Troubleshooting sections in this manual and the Known Issues section of the Release Notes. If you still need help, you can contact the Apex development team by sending email to:

apexhelp@eos.arc.nasa.gov

Sending email to this address is the fastest way to receive help. If you are reporting what appears to be a bug, first see if you can reproduce it. Please include the following information in your email:

- Your operating platform, including the following:
  - type of computer and operating system
  - the version of Apex (available from the Sherpa Help menu)
  - the version of *Allegro Common Lisp* (if applicable).
- Detailed description of the problem, including the following:
  - any error messages that appeared (in their entirety, cut and pasted if possible)
  - the last thing you tried before the problem occurred
  - whether you could reproduce the problem

3 Creating Apex Applications

Apex supports three kinds of applications: *native simulation applications*, *real-time applications*, and *foreign simulation applications*. Apex also supports user-defined application types – see chapter 6 for details.

- *Native simulation applications*, also called *simworlds*, are fully contained in Apex. These applications use the Apex *simulation engine* (simengine) to model agents interacting with other agents, an external environment, or both. The simengine is an event-driven simulator, where events are discrete actions on ongoing *activities*. All components of a simworld are defined using Apex. In a simworld, time is simulated and controlled by the Apex simengine. This simulated time has no relation to real time and may pass slower or faster, depending on many factors including the complexity of the application and the resources of the computer.

- *Real-time applications* do not use the Apex simulation engine. Instead, agents operate in real time, as measured by the system clock in the computer. Apex has been used to control autonomous vehicles through real-time applications that interface Apex with the physical system.

- *Foreign simulation applications* are those in which Apex is connected to an external simulation system, with Apex agents interacting with the externally simulated world. The interface for this type of application is still under development, and at present foreign simulation applications are handled as real-time applications. The X-Plane® sample application provided with Apex is such an example.

3.1 Lisp programming and Emacs

You can create Apex applications with a text editor using the Apex *Procedure Description Language* (PDL) along with Common Lisp (the programming language upon which PDL is based and Apex is implemented). While a basic working knowledge of Lisp is necessary to create Apex applications, you can learn Lisp as you learn Apex. This manual does not cover Lisp, but you can learn it from a variety of books and tutorials.

The Apex team recommends the Emacs text editor for use with Apex due to its strong support for Lisp programming and because you can run Apex "inside" Emacs (see Appendix J). Currently, there are two popular versions of Emacs: Gnu Emacs and Xemacs. While both work well as text editors, only Gnu Emacs has been tested for use with Apex.

Emacs is freely available on every computer platform that supports Apex. A good way to learn Emacs is from a tutorial accessible through the Emacs Help menu.

3.2 Application Definition File

Every Apex application requires an Application Definition File (ADF). An ADF is a Lisp file that contains a `defapplication` form. The `defapplication` form (see Chapter 5) names the application, specifies libraries and other files that comprise the application, and defines the interface for the application.

Along with the `defapplication` form, an ADF may contain PDL and Lisp code. In some cases an ADF contains all application code, such as the Hello World sample application provided with Apex.

An example of the `defapplication` form follows:

```
(defapplication "My application"
  :libraries ("human")
  :files ("file1.lisp" "file2.lisp")
  :init-sim (init-my-app))
```

3.3 Application files

An Apex application can consist of any number of files in addition to the Application Definition File. You can add these files to the application in either of the following ways:

- Using the `:files` clause in the `defapplication` form.
- Using the `require-apex-file` form. This form allows files to arbitrarily load other files, thus allowing an application to be a hierarchy of files.

These application files are typically Lisp files[2], but they may also be other file types, such as binary files used via Lisp's foreign function interface. An important rule is that Lisp source files *must* have a Lisp extension (.lisp, .cl, or .lsp) and non-Lisp files must *not* have a Lisp extension.

3.4 Libraries

A library is a collection of related definitions. Libraries allow you to share a body of Apex code (for example, PDL procedures) across different applications. A library might consist of one file or many files, but this difference is transparent to the users.

3.4.1 Using libraries

You can include an existing library in an Apex application in one of the following ways:

- Including its name in the `:libraries` clause of `defapplication`, for example:

```
(defapplication "My World"
  :libraries ("human" "Boeing757-cockpit")
 …)
```

- Loading it directly (on demand) with the `require-apex-library` form, for example:

```
(require-apex-library "human")
```

[2] Lisp files may be loaded into Apex in either source or compiled form, but at this time compilation of Lisp source is not performed automatically by Apex, and the Apex standard distribution does not include a Lisp compiler.

3.4.2 Creating libraries

Like an Apex application, a library can consist of one or more files. The top-level file of the library is called the *library file* and may contain Lisp code. This file may constitute the entire library, or it may include other libraries (using `require-apex-library`) or other files (using `require-apex-file`).

A library filename must include the suffix –apexlib (for example, `human-apexlib.lisp`). You can store a library anywhere. However, if the library consists of several files, you can store the files in a directory named after the library "base name". For example, Apex can find the human library if it is filed as either `human-apexlib.lisp` or `human/human-apexlib.lisp`).

3.4.3 Finding libraries

The Lisp global variable `*apex-library-path*` specifies where libraries are located. This variable contains list of directories that Apex searches in the given order. The default value of `*apex-library-path*` is:

```
(:application "apex:apexlib" "apex:examples:apexlib")
```

The special symbol `:application` indicates that the application directory itself is first searched for libraries. The following two strings in the list use the Common Lisp *logical pathname* syntax. However, you can use any valid filename syntax for your computer platform.

You can modify the search path as needed. For example, to have Apex first look in its provided libraries directory and then in the directory `C:/me/apexlib`, enter the following form in the Listener:

```
(setq *apex-library-path* '("apex:apexlib" "C:/me/apexlib"))
```

To set the path across Apex sessions, you may also add this form to the user preferences file.

3.4.4 Provided libraries

Apex is shipped with two sets of libraries:

- `apex:apexlib` contains components useful for a wide range of applications.
- `apex:examples:apexlib` contains libraries used by the example applications provided with Apex. (Note that these libraries are provided for convenience and illustration, but are not supported).

See the comments in the library files for a description of the libraries.

4 Procedure Description Language

4.1 Introduction

Procedure Description Language (PDL) is a formal language used to specify the behavior of Apex agents. PDL can be seen as a means of representing particular kinds of content – for example, mission plans for an autonomous robot, partial plans or SOPs defining how to achieve goals that might arise in carrying out a mission; a task analysis describing observed or expected human behavior; a human cognitive model reflecting procedural and declarative memory. However, making effective use of PDL requires also understanding it as a programming language for invoking the capabilities of the Apex Action Selection Architecture. This section describes the syntax of PDL following a brief overview of the workings of the Action Selection Architecture.

The central language construct in PDL is a procedure, which contains at least an `index` clause and one or more `step` clauses. The `index` uniquely identifies the procedure and typically describes what kind of task the procedure is used to accomplish. Each `step` clause describes a subtask or auxiliary activity prescribed by the procedure.

```
(procedure
   (index (get-hires-image ?target))
   (profile camera-1)
   (step s1 (move-to-standoff ?target => ?loc))
   (step s2 (power-up camera-1))
   (step s3 (orient-camera camera-1 to ?target)
      (waitfor (:and (terminated ?s1) (terminated ?s2))))
   (step s4 (take-picture camera-1)
      (waitfor (terminated ?s3)))
   (restart-when (resumed +this-task+))
   (terminate (when (image-in-mem ?target))))
```

The example procedure above, describing how to obtain an image using a fixed mounted aerial camera, illustrates several important aspects of PDL. First, a PDL procedure's steps are not necessarily carried out in the order listed or even in a sequence. Instead, steps are assumed to be concurrently executable unless otherwise specified. If step ordering is desired, a `waitfor` clause is used to specify that the completion (termination) of one step is a precondition for the start (enablement) of another. In the example, steps `s1` and `s2` do not contain `waitfor` clauses and thus have no preconditions. These steps can begin execution as soon as the procedure is invoked and can run concurrently. Step `s3`, in contrast, cannot begin until the conditions named in its associated `waitfor` clause have occurred -- namely that the first two steps have terminated.

Procedures are invoked to carry out an agent's active tasks. Tasks, which can be thought of as agent goals[3], are stored on a structure called the agenda internal to the Action Selection Architecture. When a task on the

[3] The term task generalizes the concept of a classical goal – i.e. a well-defined state, expressible as a proposition, that the agent can be seen as desiring and intending to bring about (e.g. "be at home"). Tasks can also, e.g., encompass multiple goals ("be in car seat with engine started and seatbelt fastened"), specify

agenda becomes enabled (eligible for immediate execution), what happens next depends on whether or not the task corresponds to a primitive or non-primitive action. If the task is non-primitive (i.e. it specifies an action for which no primitive procedure or built-in action type has been defined), the specified action is carried out and then the task is terminated. There are a limited number of built-in action types, each with a distinct effect.

If the task is not a primitive or built-in, the Action Selection Architecture retrieves a procedure whose index clause matches the task. For example, a task of the form `(get-hires-image target1)` matches the `index` clause of the procedure above and would thus be retrieved once the task became enabled. `step` clauses in the selected procedure are then used as templates to generate new tasks, which are then added to the agenda. It is conventional to refer to these tasks as subtasks of the original and, more generally, to use genealogical terms such as child and parent to describe task relationships. The process of decomposing a task into subtasks on the basis of a stored procedure is called task refinement. Since some of the tasks generated through this process may themselves be non-primitive, refinement can be carried out recursively. This results in the creation of a task hierarchy.

An Apex agent initially has on its agenda a single task specified by the user, which defaults to the form `(do-domain)`. All agent behavior results from tasks descending hierarchically from this initial task. Thus, the specification of agent behavior for a given application (model) must include either a procedure with the `index` clause

```
(index (do-domain))
```

or one whose `index` clause matches the specified initial task. Steps of this procedure should specify not only the main "foreground" activities of the agent, but also any appropriate background activities (e.g. low priority maintenance of situation awareness) and even reflexes (e.g. pupil response to light).

4.1.1 Action Selection Architecture

The Action Selection Architecture[4] integrates a set of algorithms that Apex uses to generate behavior. Input to the algorithm consists of events that the agent might respond to and a set of predefined PDL procedures. For example, the architecture outputs commands to resources. For mobile robots or UAVs, resources are effectors such as manipulator arms and aircraft flight surfaces. When used to generate behavior for a simulated human agent, resources are representations of cognitive, perceptual and motor faculties such as hands and eyes. Since the Action Selection Architecture could be used to model other entities with complex behavior such as robots and autopiloted aircraft, resources could correspond to, e.g. robotic arms or flight control surfaces. The Action Selection Architecture incorporates a range of functional capabilities accessible through PDL. These functions fall into four categories:

- Hierarchical action selection
- Reactive control

---

goals with indefinite state ("finish chores"), specify goals of action rather than state ("scan security perimeter "), and couple goals to arbitrary constraints ("be at home by 6pm").

[4] Designated the Action Selection Architecture in other documents. To some, this term implies that the architecture performs AI planning tasks, but not scheduling or control. The term Action Selection Architecture was chosen to be happily ambiguous about the underlying technology.

- Resource scheduling
- General programming language functions

Hierarchical action selection refers to the process of recursively decomposing a high-level task into subtasks, down to the level of primitive actions. The basic process of selecting action by hierarchical task decomposition is simple. Tasks become enabled when their associated preconditions have been satisfied. If the task is not a primitive or built-in, a procedure whose index clause matches the task is retrieved and one new task (subtask) is created for each step of the selected procedure. If the enabled task is a primitive or built-in, the specified action is executed and the task is terminated.

PDL provides flexibility in controlling how and when task decomposition takes place. The issue of how to decompose a task arises because there are sometimes alternative ways to achieve a goal, but which is best will vary with circumstance. Criteria for selecting between different procedures are represented in the `index` clause and the `select` clause. The issue of *when* to decompose a task is equally crucial since an agent will often lack information needed to select the appropriate procedure until a task is in progress. The ability to specify what needs to be known in order to select a procedure (informational preconditions) is provided by the `waitfor` clause.

Reactive control refers to a set of abilities for interacting in a dynamic task environment. As noted above, the ability to cope with uncertainty in the environment sometimes depends on being able to delay commitment to action; when crucial information becomes available, the agent can select a response. Another aspect of reactivity is the ability to handle a range of contingencies such as failure, interruption, unexpected side effects, unexpectedly early success and so on. An important part of reactive control is recognizing goal-relevant conditions, especially including conditions indicating how best to achieve intended goals or signifying a special contingency. Apex includes capabilities for detecting complex conditions of interest including temporal event patterns such as trends in sensor data.

Reactive mechanisms combined with looping and branching allow closed-loop control – i.e. the ability to manage a continuous process based on feedback. The combination of discrete control mechanisms such as hierarchical action selection with continuous control mechanisms allows PDL to model a wide range of behaviors.

Resource scheduling refers to the ability to select execution times that meet specified constraints for a set of planned actions. Typically, an overriding goal is to make good (possibly optimal) use of limited resources. Actions can be scheduled to run concurrently unless they conflict over the need for a non-sharable resource (e.g. a hand) or are otherwise constrained. For example, an eye-movement and an unguided hand movement such as pulling a grasped lever could proceed in parallel. PDL includes numerous clauses and built-in action types for dynamically asserting, retracting and parameterizing scheduling constraints.

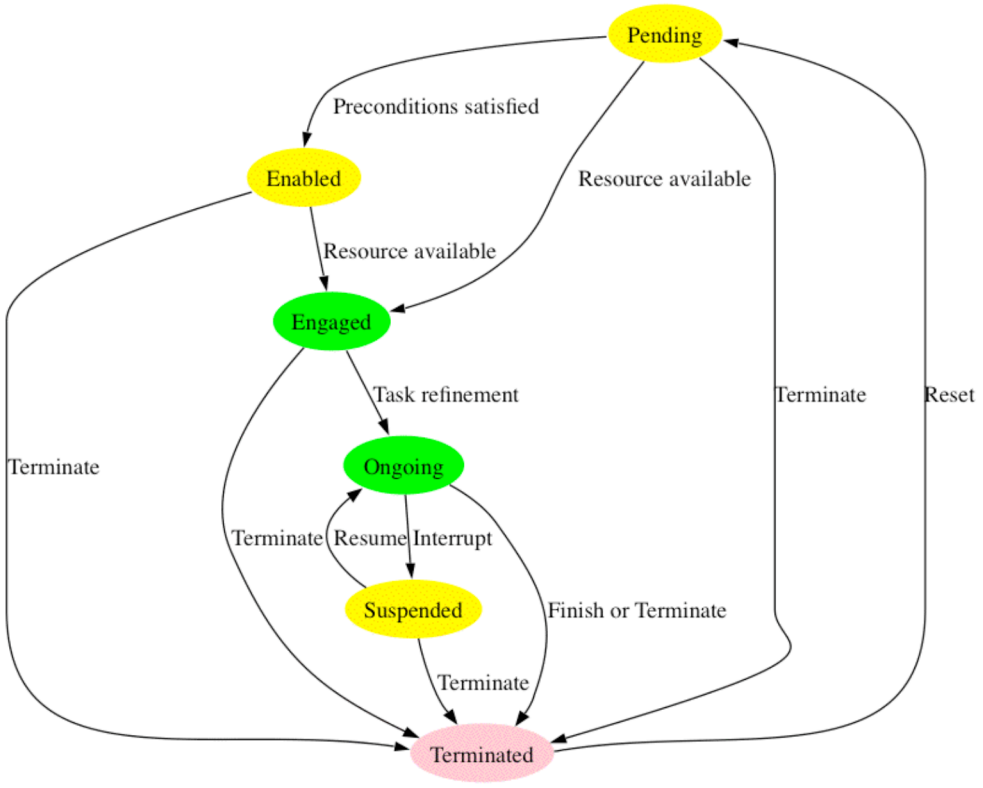
Scheduling is tightly integrated with reactive control and hierarchical planning. In a less tightly integrated approach, these functions might be assigned to modular elements of the architecture and carried out in distinct phases of its action decision process. In Apex, these activities are carried out opportunistically. For example, when the information to correctly decompose a task into subtasks becomes available, the architecture invokes hierarchical planning functions. Similarly, when there are a set of well-specified tasks and scheduling constraints on the agenda, Apex invokes scheduling functions.

This has two important implications for the role of scheduling in Apex. First, scheduling applies uniformly to all levels in a task hierarchy. In contrast, many approaches assume that scheduling occurs at a fixed level – usually at the "top" where a schedule constitutes input to a planner. Second, the tasks and constraints that form input to the scheduler must be generated dynamically by hierarchical planning and reactive control

mechanisms, or inferred from local (procedure-specific) constraints, evolving resource requirements, and changes in the execution state of current tasks. Basic scheduling capabilities can be employed without a detailed understanding of the architecture. For more advanced uses of these capabilities, it is hoped that the PDL construct descriptions will prove helpful. Further information can be found in Freed (1998a, 1998b).

General programming language functions such as looping and branching are included in PDL language constructs. However, the user will sometimes wish to access data or functions not directly supported in PDL but available in the underlying Lisp language. PDL supports callouts to Lisp that apply to different aspects of task execution including: precondition handling, action selection and specification of execution parameters.

Task state. Procedures describe the parameterized methods for carrying out an agent's goals; primitives describe the parameterized, specific "effector" actions an agent can take. The Action Selection Architecture instantiates these as tasks, as described above. Tasks start in a *pending* state; when a task's preconditions are met, it passes into an *enabled* state. If the task's required resources are available, it becomes *engaged* and then immediately *ongoing*[5]. The task may be interrupted (and thus pass to a *suspended* state) and resumed again (back to an *ongoing* state). On completion, the task enters a *terminated* state. Task may become *terminated* from any of its task states (for example, if its parent task is terminated). The following transition diagram shows possible task state transitions. The Sherpa tool can also show task state transitions in a number of ways: via the Trace view, the Agenda view, or the State Variable view.



Task State Transitions
In Apex

[5] We mention *engaged* for completeness; this is essentially internal bookkeeping required by the Action Selection Architecture.

4.2 Basics

4.2.1 Introduction to PDL variables

In order to be able to distinguish literals (such as `throttle`) from variables (such as `?units`), the Apex system requires that all variables have prefixes. Variables in PDL are typically prefixed with a question mark (?). Thus, `?age` is a PDL variable, as are `?altitude` and `?rate`. These are called *free variables*.

Variables occur in PDL patterns, such as in the index clause for a procedure or in the conditions for a monitor. For example, the index clause for a procedure to increase throttle speed to a certain rate is `(increase throttle speed to ?rate ?units per ?timeunit)`. A call to this procedure in another procedure might look like `(increase throttle speed to 40 meters per second)`. Apex uses pattern matching (see Appendix H) to find the corresponding procedure; for example, the pattern `(increase cabin pressure to ?amt)` matches `(increase cabin pressure to 33)`, but does not match `(increase throttle speed to 40 meters per second)`. The judicious use of patterns can make PDL very easy to read and understand.

In addition to free variables, PDL supports another variable type: *indexical variables*. Indexical variables, indicated by symbols starting and ending with + signs, define roles that may be filled by different values over time, for example, `+the-car-in-front-of-me+` and `+most-recently-used-visible-light-camera+`. Such variables are defined by functions that are called whenever the variable value is needed. PDL includes some predefined indexical variables including `+self+` (the current agent), `+this-task+`, (the currently executing task) and `+now+`. (the current time). Appendix H lists the standard indexical variables provided by Apex, and how to define new indexical variables.

Also, free variables can be annotated to indicated that they should have a value when they are used. The annotation consists of surrounding the variable with angle brackets, for example, `<?altitude>` and `<?rate>`. Consider the following snippet:

```
(step (increase speed of ?aircraft to ?rate meters per second)
  (waitfor (rate request <?aircraft> ?rate))
```

The condition `(rate request <?aircraft> ?rate)` enables this step; when an event occurs that matches `(rate request <?aircraft> ?rate)`, the `?rate` variable becomes bound. Since this condition enables the step, it is guaranteed that `?rate` will be bound before the step's action is taken. By annotating the variable `?aircraft` in the waitfor clause, we suggest that there should already be a binding for `?aircraft` when condition matching begins. Apex issues a warning if a variable annotated in this way is unbound during pattern matching.

4.2.2 Introduction to PDL time representation

Durations and points in time are used for a number of reasons when writing PDL. One can describe the duration and expected duration of primitives and procedures, for example, or schedule an event to occur after a certain amount of time, or constrain condition monitors to occur at a certain time.

The duration syntax is described more completely in Appendix F. Durations come in one of two forms. The *list-based form* describes a duration as a list of amount/unit pairs; for example, `(1 hr 3 min 4 sec 500 ms)`. The *symbol-based form* is a shorthand symbolic form, `PwWdDhHmMsS`, with amounts and units combined into the symbol starting with `P`; for example, `P1H3M4.5S` is the symbolic form for the list form just given. The list-based form is somewhat more readable, and allows for pattern matching. The symbol-based form is usually more compact. Again, details are in Appendix F. When a duration is required (for

example, in a `DURATION` clause), the duration must be in either a list-based or symbol-based form, except for the special case of a zero duration; in which case the numeral `0` can be used.

Points in time, when required, are represented as a Lisp expression; for example, `(+ +now+ P3s)` refers to the point in time three seconds from now. The Lisp functions `start-of` and `end-of` provide the start point and end point, respectively, for most Apex objects. So, for example, the following snippet contains a condition monitor waits for a measurement to occur *n* seconds after the start of the current task:

```
(step (release brakes)
  (waitfor (:measurement (speed engine > 100)
              :timestamp (>= (+ (start-of +this-task+) (<?n> seconds)))))
```

4.2.3 Notations and conventions

This manual uses the following conventions to describe PDL syntax:

- ( ) parentheses enclose all PDL constructs
- [ ] square-brackets enclose optional parameters
- < > angle-brackets enclose types rather than a literal values
- | vertical bars separate alternative values
- { } curly brackets enclose alternatives unless otherwise enclosed
- X+ means that 1 or more instances of X are required
- X* means that 0 or more instances of X are required

In addition, the following terminology is used.

- A *procedure-level clause* is a language construct that is embedded directly in a PDL procedure. Examples include index clauses and step clauses.
- A *step-level clause* is a language construct, such as `waitfor`, that is embedded directly in a step clause.
- A *procedure* is a first-class construct—it is not embedded in any other language element.
- A *pattern parameter* is a parenthesized expression that may contain variables.
- *Patterns* are matched against each other by the pattern matcher (see Appendix H) and can appear in several PDL clauses.
- A Lisp *symbol* is a sequence of characters that that may include alphanumeric characters, dashes, and some other characters.
- A Lisp *symbolic expression*, or *s-expression*, is either a Lisp symbol or a list of symbols and Lisp expressions enclosed by parentheses. Symbols and s-expressions in PDL clauses may contain Apex variables

4.3 Procedures

Type: first-class construct

Syntax:

```
(PROCEDURE
 [ (PROCTYPE :conc[urrent] | :seq[uential] | :ranked) ]
   (INDEX (<name:non-variable-symbol> . <parameters>) )
 [ (PROFILE <resource>+) ]*
 [ (LOG <log-policy>+) ]*
 [ (EXPECTED-DURATION <duration>) ]
 [ (TERMINATE (WHEN <monitor-condition>)) ]
 [ (INTERRUPT-COST <lisp-expression>:real) ]
 [ (STEP [<step-tag:symbol>] <task-description>
     [=> <var>] <step-clause>* ) ]*
)
```

There are three syntactic forms for procedures: *concurrent* (the default), *sequential* and *ranked*. In a concurrent procedure, all tasks generated from the procedure's steps are assumed to be concurrently executable, except where ordering, constraints, or other conditions are specified by `waitfor` clauses. A concurrent procedure usually includes an explicit termination step. For example, the procedure below might be invoked to carry out a task such as *(task-15 (open door))* when the precondition for step `s4` is satisfied, for example, when step `s3` is complete, `s4` will cause *(task-15 (open door))* to terminate.

```
(procedure
   (index (open door))
   (step s1 (grasp door-handle))
   (step s2 (turn door-handle) (waitfor ?s1))
   (step s3 (push) (waitfor ?s2))
   (step s4 (terminate (waitfor ?s3))))
```

It is quite common to define procedures consisting of a totally ordered set of steps. Such procedures can be conveniently represented using the sequential procedure syntax. The following example is equivalent to the concurrent procedure above.

```
(procedure :sequential
   (index (open door))
   (step (grasp door-handle))
   (step (turn door-handle))
   (step (push)))
```

A sequential procedure includes only an index clause and a list of steps to be carried out in the listed order. No terminate clause is required. Sequential procedures are not really a separate type from concurrent procedures, but merely an alternative syntax. PDL mechanisms automatically translate them into equivalent concurrent procedures by adding a `terminate` step and `waitfor` clauses as needed to specify step order.

Ranked procedures abbreviate a concurrent procedure form in which `rank` clauses are added to each step. Rank values in these procedures are in ascending order of appearance. Thus, the `:ranked` procedure below is equivalent to the procedure with explicitly ranked steps:

```
(procedure
   (index (open door))
   (step s1 (grasp door-handle)       (rank 1))
   (step s2 (turn door-handle)        (rank 2))
   (step s3 (push)      (rank 3))
   (step s4 (terminated) (waitfor ?s1 ?s2 ?s3)))

(procedure :ranked
   (step (index (open door)))
   (step (grasp door-handle))
   (step (turn door-handle))
   (step (push)))
```

4.3.1 index

Type: procedure-level clause

Syntax: `(index <pattern>)`

Each procedure must include a single `index` clause. The index pattern uniquely identifies a procedure and, when matched to a task descriptor, indicates that the procedure is appropriate for carrying out the task. The pattern parameter is a parenthesized expression that can include constants and variables in any combination. The following are all valid `index` clauses:

```
(index (press button ?button))
(index (press button ?power-button))
(index (press button ?button with hand))
(index (press button ?button with foot))
```

Since index patterns are meant to uniquely identify a procedure, it is an error to have procedures with non-distinct indices. Distinctiveness arises from the length and constant elements in the index pattern. For example, the first and second `index` clauses above are not distinct since the only difference is the name of a variable. In contrast, the 3rd and 4th `index` clauses are distinct since they differ by a constant element.

The pattern matcher provides a great deal of flexibility in specifying a pattern. For example, the following `index` clause includes a constraint that the pattern should not be considered a match if the value of the variable is self-destruct-button.

```
(index (press button ?button
   (?if (not (eql ?button 'self-destruct-button)))))
```

In the next example, the variable `?*.button-list` matches to an arbitrary number of pattern elements. This provides the flexibility to create a procedure that presses a list of buttons without advance specification of how many buttons will be pressed.

```
(index (press buttons (?* button-list)))
```

See Appendix H for more information on the pattern matcher.

Although an `index` clause is required, it can be shortened to just its pattern. So for example, the following procedures are the same:

```
(procedure (index (press button ?button)) … )

(procedure (press button ?button) …)
```

If the short form of an `index` clause is used, it must be the first full clause after `procedure`. (It may follow a keyword indicating the procedure type).

4.3.2 proctype

Type: procedure-level clause

Syntax: `(PROCTYPE :conc[urrent] | :seq[uential] | :ranked)`

The `proctype` clause is optional. Procedures default to `:concurrent`. The `proctype` clause can be shortened to just the procedure type (that is, `:sequential` or `:concurrent` or `:ranked`); in this case, the procedure type must appear immediately after the procedure form. Thus, the following groups of forms are equivalent to one another:

```
(procedure :seq  (do something) …)
(procedure (index (do something)) (proctype :seq) …)

(procedure :conc  (do something) …)
(procedure (index do something) (proctype :conc) …)
(procedure (index (do something) …)

(procedure :ranked  (do something) …)
(procedure (index (do something))
  (proctype :ranked) …)
```

4.3.3 profile

Type: procedure-level clause

Syntax: `(profile <resource-requirement>+)`

```
resource-requirement:
  resource:<symbol>
```

The `profile` clause lists discrete resources required for using a procedure.[6] More than one `profile` clause may be specified. Whenever the procedure is selected for a task, the resource requirements become additional preconditions (beyond those prescribed by `waitfor` clauses) for beginning execution of the

[6] The profile clause is only used for "blocking" resources such as hands and eyes that can only be allocated to one task at a time, but are not consumed by use. There are currently no mechanisms to support reasoning about depletable resources such as fuel or money.

task. For example, the following procedure declares that if selected as a method for carrying out a task, that task cannot begin execution until the Action Selection Architecture allocates to it a resource named `right-hand`.

```
(procedure
  (index (shift manual-transmission to ?gear))
  (profile right-hand)
  (step s1 (grasp stick with right-hand))
  (step s2 (determine-gear-position ?gear => ?pos))
  (step (move right-hand to <?pos>)
    (waitfor ?s1 ?s2))
  (step s4 (terminate) (waitfor ?s3)))
```

The `profile` may specify resources as variables as long as these are specified in the `index` clause. For example, the procedure above could be specified as follows:

```
(procedure
   (shift manual-transmission to ?gear using ?hand))
   (profile ?hand)
   …)
```

Resource preconditions are determined when a procedure is selected, rather than after all `waitfor` preconditions have been satisfied. Thus, the architecture only makes allocation decisions for tasks that are enabled or already ongoing. The architecture allocates resources to tasks based on the following rules:

1. A task is competing for the resources listed in its `profile` if it is either enabled (all `waitfor` preconditions satisfied) or already ongoing .

2. If only one task competes for a resource, it is allocated to that task.

3. If multiple tasks compete for a resource, allocation is awarded to the task with highest `priority`.

4. If one of the tasks competing for a resource is already ongoing (and thus has already been allocated the resource), its `priority` is increased by its `interrupt-cost`. By default, the interrupt cost is slightly positive, producing a weak tendency to persist in a task, rather than interrupt it.

5. Tasks at any level in a task hierarchy may require and be allocated resources. However, a task never competes with its own ancestor for a resource.

6. If a `profile` lists multiple resources, it is allocated all of them or none. If there is a resource for which it is not the highest `priority` competitor, then it does not compete for the other resources and any resources already allocated become deallocated. This rule takes precedence over rules 2 and 3.

Resources listed in a `profile` clause do not necessarily correspond to components of the agent resource architecture, the collection of modules that either provide information to the Action Selection Architecture or can be commanded by it using the primitive actions. Resources named in a `profile` clause that do not correspond to an element of the resource architecture are *virtual resources*.

4.3.4 log

Type: procedure-level clause

Syntax: `(log <log-policy>+)`

```
log-policy: (attr obj) |((attr obj)
[:frequency-limit <duration>]
[:time-limit <duration>] [:count-limit <real>])
```

The `log` clause specifies logging policies for the recording of state variable measurements. More than one `log` clause may be present. The policy can be in one of two forms. The short form is to list a state variable; this specifies that measurements of the state variable will be recorded for the duration of the task. The long form allows greater control over logging. The frequency limit places an upper limit on the frequency of logging. The time limit places an upper limit on how long measurements will be logged. The count limit places an upper limit on the number of measurements that will be logged. When logging policies are in conflict, the least restrictive policy is used.

The following procedures show two concurrent tasks: driving and making a rest stop when a driver gets hungry. Unless the driver agent remembers signs for rest stops, it won't know that it has passed a rest stop to which it can return. Thus a logging policy is set to remember signs, as measured as values of a `(message-read vision)` state variable.

```
(procedure :seq
    (index (make rest stop))
  (step (use resource (car) for 5 +/- 1 min)
   (waitfor
   (:measurement (message-read vision = |rest stop|)))
  (step (cogevent (hunger diminished) +self+)))


(procedure
    (index (do-domain))
  (log (message-read vision))
  (step s1 (drive)
   (priority 1))
(step s4 (make rest stop)
   (priority 9)
   (waitfor (hunger increased)))
  (step term (terminate)
   (waitfor (message-read vision = |end of road|)))))
```

4.3.5 duration

Type: procedure-level clause

Syntax: `(duration <duration>)`

The `duration` clause specifies the actual (prescribed) duration for a primitive. The primitive will terminate when this time has transpired.

4.3.6 expected-duration

Type: procedure-level clause

Syntax: `(expected-duration <duration>)`

The `expected-duration` clause annotates the procedure with its expected duration. The `expected-duration` is an estimate that helps reasoning mechanisms to make good guesses about how to use resources efficiently. In certain conditions, Apex uses this value to decide whether to start a task or wait for a better window of opportunity. Specifically, it is used to determine whether the task is brief enough to be treated as an inconsequential interruption of a higher priority task.

4.3.7 terminate/when

Type: procedure-level clause

Syntax: `(terminate (when <condition>))`

The `terminate` clause describes a condition under which the task described by the procedure should be terminated. It is effectively the same as adding a `terminate` step; the following two procedures are effectively equivalent.

```
(procedure (howl on back fence)
(terminate (when (shoe thrown))
(step (caterwaul)
  (repeating :with-min-interval '(10 secs))))

(procedure (howl on back fence)
(step (caterwaul)
  (repeating :with-min-interval '(10 secs))))
(step (terminate (waitfor (shoe thrown))
```

4.3.8 interrupt-cost

Type: procedure-level clause

Syntax: `(interrupt-cost <integer>|<s-expression> )`

The `interrupt-cost` specifies a degree of interrupt inhibition for an ongoing task. The `interrupt-cost` is computed whenever the task is ongoing and competing for resources – i.e. resources it has already been allocated and is "defending." The value is added to the task's local priority. Note: this is also a step-level clause.

4.3.9 on-start

Type: procedure-level clause

Syntax: `(on-start < s-expression>)`

When a task is created from the procedure, the Lisp form is executed.

4.3.10 on-end

Type: procedure-level clause

Syntax: `(on-end  < s-expression>)`

When the task is terminated, the Lisp form is executed.

4.4 Steps

The `step` clause describes a task to be undertaken during a procedure. In addition to specifying the task to be undertaken, step-level clauses are provided to describe when a task should be enabled, terminated, etc.; whether a step should be repeated, and how; specifying priority, interrupt cost and rank; and Lisp functions to execute on the start and completion of tasks. The following is an overview of the step clause:

```
(STEP [<step-tag:symbol>] <task-description> [=> <var>]
 [ (WAITFOR <monitor-condition> )
 [ (INTERRUPT-COST <lisp-expression>->real) ]
 [ (PRIORITY  <lisp-expression>->real) ]
 [ (RANK  <lisp-expression>->real) ]
 [ (REPEATING <repeat-spec>) ]
 [ (RESPONDING <respond-spec>) ]
 [ (SELECT <select-spec>) ]
 [ (FORALL <forall-spec>) ]
 [ (ON-START <lisp-expression>*) ]*
 [ (ON-END <lisp-expression>*) ]*
 [ (TERMINATE (when <monitor-condition>)) ]
 [ (RESTART (when <monitor-condition>)) ]
 [ (RESUME (when <monitor-condition>) ]
 [ (SUSPEND (when <monitor-condition>) (until <monitor-condition>)) ]
 )
```

4.5 Monitoring overview

A task has a life-cycle: typically it is created, waits until its preconditions are met, becomes on-going, and eventually terminates. It may be suspended and resumed, or restarted. The step level clauses `waitfor`, `terminate/when`, `suspend/when/until`, `resume/when`, and `restart/when` all provide programmer control over task transitions. Each of these clauses require a *condition* specification for which the Apex system monitors: when the condition is met, the task transition occurs (assuming all other conditions for the transition are also met). The syntax for conditions is described in Section 4.8 below.

4.5.1 waitfor

Type: step-level clause

Syntax: `(waitfor <condition>+)`

A `waitfor` clause defines a precondition that must be satisfied for the task to become enabled, that is, eligible for execution.

`Waitfor` clauses are useful for specifying execution order for steps of a procedure. This is accomplished by making the termination of one step a precondition for the enablement of another. The Action Selection Architecture records task state transitions events of the form (state <*task*> = terminated) when a task is terminated, so a clause such as (waitfor (state **?s3** = terminated)) will impose order with respect to the task bound to the task-tag-variable ?s3. Termination preconditions can be expressed using an abbreviated form: (waitfor <*task-tag-var*>)) == (waitfor (state <*task-tag-var*> = terminated)). Thus, the expression (waitfor ?s3) is equivalent to (waitfor (state ?s3 = terminated)).

More than one precondition can be specified in a waitfor clause; this is the same as the conjunction of the conditions. For example, the expression (waitfor ?s1 ?s2 ?s3) is equivalent to (waitfor (:and ?s1 ?s2 ?s3)); these are expanded in the way described in the previous paragraph. conjunctive; all must be satisfied for the task to become enabled.

Correctly specifying `waitfor` preconditions is perhaps the trickiest part of PDL. For example, one might want to express a behavior that becomes enabled in response to a red light, representing this with:

```
(waitfor
  (:and (color ?object red)
        (luminance ?object = high)).
```

Given the semantics of conditions, this condition will be met only if an event of the form `(color ?object red)` and a measurement of the form `(luminance ?object = high)` occur after the enclosing step is created.

4.5.2 terminate/when

Type: step-level clause

Syntax: `(terminate (when <condition>))`

A `terminate/when` clause defines a condition that, when satisfied, will cause the current step to be terminated. For example, the following step will be terminated if a `(stop doing that)` event occurs:

```
(step (do long involved task)
      (terminate (when (stop doing that)))
```

Note that termination of repeating and responding policies are handled in a different way; the `terminate/when` clause refers only to the current execution of the task.

4.5.3 suspend/when/until

Type: step-level clause

Syntax: `(suspend (when <condition1>) (until <condition2>)`

A `suspend/when/until` clause defines a condition that, when satisfied, will cause the current step to be suspended, and will re-enable the task when a second condition is met. For example, the following step will be suspend if a `(stop doing that)` event occurs, but will be re-enabled when a `(resume doing that)` event occurs :

```
(step (do long involved task)
  (suspend (when (stop doing that))
           (until (resume doing that))))
```

Note that termination of repeating and responding policies are handled in a different way; the `suspend/when/until` clause refers only to the current execution of the task.

4.5.4 restart/when

Type: step-level clause

Syntax: `(restart (when <condition>))`

A `restart/when` clause defines a condition that, when satisfied, will cause the current step to be restarted; i.e., placed back into a pending state. For example, the following step will be restarted if a `(try again)` event occurs:

```
(step (do long involved task)
  (restart (when (try again)))
```

Note that termination of repeating and responding policies are handled in a different way; the `restart/when` clause refers only to the current execution of the task.

4.5.5 resume/when

Type: step-level clause

Syntax: `(resume (when <condition>))`

A `resume/when` clause defines a condition that, when satisfied while the task is in an interrupted state, will cause the current step to be resumed; i.e., placed back into a pending state. For example, the following step will be restarted if a `(try again)` event occurs:

```
(step (do long involved task)
  (resume (when (try again)))
```

Note that termination of repeating and responding policies are handled in a different way; the `resume/when` clause refers only to the current execution of the task.

4.5.6 select

Type: step-level clause

Syntax: `(select <variable> <s-expression>)`

The `select` clause is used to choose between alternative procedures for carrying out a task. Its influence on selection is indirect. The direct effect of a `select` clause is to bind the specified variable to the evaluation of the Lisp-expression argument. This occurs as the task becomes enabled, just prior to selecting a procedure for the associated task, so instances of the variable in the task description will be replaced by the new value and may affect procedure selection.

```
(step (press ?button with ?extremity)
  (select ?extremity
   (if (> (height ?button) .5) 'hand 'foot)))
```

In the example above, the value of the variable `?extremity` is set to `hand` if the button is more than `.5` meters off the ground, otherwise it is set to `foot`. Assuming procedures with `index` clauses `(index (press ?button with hand))` and `(index (press ?button with foot))`, the effect of the `selection` clause is to decide between the procedures.

Only one `select` clause can be defined per step.

4.6 Task control overview

Programmers have control over the priority, rank and interrupt cost of tasks. The following clauses describe how these are set.

4.6.1 Priority

Type: step-level clause

Syntax: `(priority {<integer>|<variable>|<s-expression>})`

A `priority` clause specifies how to assign a priority value to a task in order to determine the outcome of competition for resources—higher priority tasks are assigned resources before lower priority tasks. The assigned value is a unitless integer. It can be specified as a fixed value, as a variable that evaluates to an integer, or as an arbitrary Lisp s-expression.

A task's `priority` is first computed when it becomes enabled, is matched to a procedure that requires a resource (i.e. includes a profile clause), and is found to conflict with at least one other task requiring the same resource. If the task is not allocated a needed resource, then it remains in a pending state until one of several conditions arises causing it to again compete for the resource. These conditions are: (1) the resource is deallocated from a task that currently owns it, possibly because that task terminated; (2) new competition for that resource is initiated for any task; (3) the primitive action `reprioritize` is executed on the task. Whenever a task begins a new resource competition, its `priority` is recomputed.

A `step` may have multiple `priority` clauses, in which case, the `priority` value from each clause is computed separately. The associated task is assigned whichever value is the highest. This value is the local

priority value. Tasks may also inherit priority from ancestor tasks. A task could have one or more inherited priorities but no local priority. Alternately, it may have no inherited priorities but a local priority. In all cases, task priority equals the maximum of all local and inherited values.

Note: In some cases, a task will become interrupted but one or more of its descendant tasks will become or remain ongoing. These descendants no not inherit priority from the suspended ancestor.

4.6.2 Rank

Type: step-level clause

Syntax: `(rank {<integer>|<variable>|<s-expression>})`

Like a `priority` clause, a `rank` clause specifies how to determine the outcome of competition for resources—lower ranked tasks are assigned resources before higher ranked ones. The assigned value is a unitless integer. It can be specified as a fixed value, as a variable that evaluates to an integer or as an arbitrary Lisp s-expression. `Rank` values are computed whenever `priority` values are computed.

Though also used to resolve resource conflicts, `rank` is very different form `priority`. Whereas a task's priority is an intrinsic (globally scoped) property, its rank depends on what task it is being compared to. For example, consider the procedure below:

```
(procedure
   (index (record phone number of ?person))
   (step s1 (determine phone-number of ?person) (rank 1))
   (step s2 (write down phone-number of ?person) (rank 2))
   (step s3 (terminate) (waitfor ?s1 ?s2)))
```

This procedure specifies that activities related to determining a specified person's phone number can be carried out in parallel with activities for writing the number down – i.e. the latter task and all of its descendant subtasks (e.g. `(task-25 (grasp pencil))`) do not have to wait for the former task to complete. However, resource conflicts will automatically be resolved in favor of the better-ranked task – i.e. the one with the lower `rank` value. Thus, if `(task-25 (grasp pencil))` and `(task-22 (grasp phone book)`) both need the right hand, the latter task will be favored since it descends from a task with superior rank.

To determine rank for two conflicting tasks *A* and *B*, the architecture locates a pair of tasks *A'* and *B'* for *A'* is an ancestor of *A*, *B'* is an ancestor of *B*, and *A'* and *B'* are siblings – i.e. derived from the same procedure. If no rank is specified for *A'* and *B'*, then *A* and *B* have no rank relative to one another. Resource conflict is then resolved based on `priority`. Otherwise, rank values for *A'* and *B'* are inherited and used to resolve the conflict.

4.6.3 interrupt-cost

Type: step-level clause

Syntax: `(interrupt-cost {<integer>|<variable>|<s-expression>})`

The `interrupt-cost` specifies a degree of interrupt-inhibition for an ongoing task. The `interrupt-cost` is computed whenever the task is ongoing and competing for resources – i.e. resources it has already been allocated and is "defending." The value is added to the task's local priority.

4.7 Repetition, policies, and iteration

The PDL clauses `repeating`, `responding`, and `forall` provide support for repetition, response policies, and iteration, `respectively`.

4.7.1 repeating

Type: step-level-clause

Syntax:

```
(repeating  [:enabled [<test>]]
            [:while <test>]
            [:until <test>]
            [:with-recovery <duration>]
            [:with-min-interval <duration>]
            [:for-new <var1> ..<varn>] )
```

The `repeating` clause creates and controls repetition. The simplest form of the clause, `(repeating)`, declares that the task should restart immediately after it terminates and repeat continuously. In this case, repetition ceases only when the parent task terminates or the task is explicitly terminated (by a `terminate` clause).

To be eligible for repetition, a task must be blocked from immediate or effortless re-enablement, thus preventing the system from generating endless zero-duration repetitions and thereby hanging. This condition is satisfied if either (a) the task has preconditions (not suppressed by the `:enabled` argument described below), (b) it has non-empty resource requirements, (c) decomposes into subtasks, at least one of which has preconditions or resource requirements, or (d) it's a primitive task with either a specified duration or update interval.

All arguments of the `repeating` clause are optional. By default, any `waitfor` preconditions associated with a recurrent task are reset to their initial unsatisfied state when the task restarts.

The `:enabled` argument causes the task to restart in an enabled state, that is, with preconditions satisfied. An optional test for enablement evaluates at restart-time; if the test evaluates to nil, the task restarts with all preconditions unsatisfied, as in the default case.

The `:while` argument specifies a test (a Lisp expression) that causes repetition to continue while the test holds.

Similarly, the `:until` argument specifies a test (a Lisp expression) that causes repetition to continue *until* the test succeeds.

The `:with-recovery` argument temporarily reduces a repeating task's priority in proportion to the amount of time since the task was last executed. This reflects a reduction in the importance or urgency of re-executing the task. For example, after checking a car's fuel gauge, there is no reason to do so again soon afterwards since little is likely to have changed. In the following example, the priority of the task for repeatedly monitoring the fuel gauge is reduced to 0 immediately after performing the task and gradually rises to its full normal value over a period of 30 minutes.

```
(step s5 (monitor fuel-gauge)
   (period :recurrent :recovery (30 minutes)))
```

The `:with-min-interval` argument specifies a minimum time between repetitions of the task.

When a `waitfor` is satisfied, any free variables it contains are bound. By default, the values bound on the first satisfaction of a `waitfor` are used in subsequent matches. For example, the following code repeatedly swats only the first fly seen.

```
(step (swat ?fly)
      (waitfor (seen ?fly))
      (repeating))
```

On the other hand, the `:for-new` argument allows variables in a `waitfor` to take on different values at each repetition (in this case swat any fly that appears). The `:for-new` argument lists the variables to be bound "anew":

```
(step (swat ?fly)
      (waitfor (seen ?fly))
      (repeating :for-new  ?fly))
```

The variables given to `:for-new` must only be free variables that appear in the step's `waitfor`.

4.7.2 responding

Type: step-level-clause

Syntax: `(responding [:while <test>] [:until <test>][:for-new <var`$_1$`> ..<var`$_n$`>] )`

The `responding` clause provides a way to specify a *response policy*, which means that a fresh instance of the task (for which this step applies) is created as a response to the class of events specified in the step's `waitfor`.[7]

```
(step(shift-gaze ?visob)
   (waitfor (new (visual-object ?visob)))
   (responding :for-new ?visob))
```

For example, the step above represents a policy of shifting gaze to any newly appearing object, even if the new object appears during the process of shifting gaze to a previously appearing object. If we used `repeating` instead of `responding`, objects appearing during a previous gaze-shift response would be ignored. The optional arguments `:while`, `:until`, and `:for-new` are defined as they are for `repeating`, specified in the previous section.

4.7.3 forall

Type: step-level-clause

Syntax: `(forall <var> in {<var>|<list>})`

[7] A step that includes a `responding` clause requires a `waitfor`.

The `forall` clause is used to repeat an action for each item in a list. For example, the following step prescribes eating everything in the picnic basket.

```
(step s3  (eat ?food)
   (forall ?food in ?basket-contents)
   (waitfor ?s2 (contents picnic-basket ?basket-contents))
```

The effect of a `forall` clause is to cause a task to decompose into a set of subtasks, one for each item in the list parameter. Thus, if the step above generates `(task-12 (eat ?food))` and the cogevent `(contents picnic-basket (sandwich cheese cookies))` occurs, the variable `?basket-contents` will become bound to the list `(sandwich cheese cookies)`.

Later, when the task bound to `?s2` is terminated, `task-12` becomes enabled. Normally, the Action Selection Architecture would then select a procedure for `task-12`. The `forall` clause takes effect just prior to procedure selection, creating a set of new tasks for each item in the `forall` list. Each of these is a subtask of the original. In this example, the `forall` clause would result in subtasks of `task-12` such as `(task-13 (eat sandwich))`, `(task-14 (eat cheese))` and `(task-15 (eat cookies))`. Procedures would then be selected for each of the new tasks.

```
(step (examine indicator ?indicator)
   (forall ?instrument in (fuel-pressure air-pressure temperature))
   (repeating))
```

Note that `forall` can be combined with `repeating`. In the example above, the step prescribes repeatedly examining a set of instruments.

4.7.4 period

The `period` clause is still supported in Apex 3.0, but it is deprecated. For syntax, see the Apex 2.4 Reference manual.

4.8 Conditions

A crucial aspect of an effective task execution system is the ability to recognize quickly conditions that require action. This can be as simple as enabling a task when another task has completed or as complicated as resetting a task when one of several complicated conditions are met.

Conditions are based on either the measurements of monitor *state variables* or the recognition of *atomic episodes*. A state variable is an attribute of some object, such as the temperature of water, the altitude of an airplane, or the current state of a task. Atomic episodes are atomic events or signals that arrive "whole" to the monitoring subsystem. An atomic episode refers to some behavior or state over time which comes externally to the Apex monitoring system without knowledge of the actual interval or semantics of the form.

In particular, Apex can monitor for the following:

- measured values of state variables (*measurement* monitors)
- estimated values of state variables (*estimation* monitors)
- behavior of a state variable over an interval (*simple episode* monitors)

- signals or events which are generated internally or externally from other systems which are not related to state variables *per se* (*atomic episode* monitors)

- logical and temporal combinations of the above (*complex episode* monitors)

In the following example, the step is enabled when a temperature gauge goes above 100:

```
(step s1 (stop waiting for water to boil)
  (waitfor (:measurement  (temperature  water = ?val) :value (> 100))
```

The step could also be enabled when a vision system sends an event indicating the presence of bubbles:

```
(step s1 (stop waiting for water to boil)
  (waitfor (bubbles in pot seen))
```

Or the step could be enabled when either of the conditions is met:

```
(step  s1 (stop waiting for water to boil)
  (waitfor
    (:or (:measurement  (temperature  water = ?val) :value (> 100))
         (bubbles in pot seen)))
```

Perhaps the most common action and condition found in Apex code is enablement on the completion of another step:

```
(step s2 (pour water into teapot)
  (waitfor  ?s2))
```

4.8.1 Monitoring state variables

In Apex, you can define state variables for object attributes. Simply put, a state variable is a time-varying attribute of an object. Syntactically, you describe a state variable using (`<attr> <obj/v>`), where `<attr>` is an attribute (such as altitude or position) and `<obj/v>` is an object reference or variable. Thus, the state variable referring to the altitude of aircraft-1 is `(altitude aircraft-1)`, and any state variable on altitude (with the value the object bound to variable `?aircraft`) is `(altitude ?aircraft)`.

A measurement is pairing of a state variable and its value at a particular point in time. Syntactically, a measurement looks like the following (taking particular note of the equals sign):

```
(<attr> <obj/v> = <val/v>)
```

where `<attr>` and `<obj/v>` are as described above and `<val/v>` is either the value or a variable to be bound to the value. State variable measurements are signaled by the Apex or Lisp form `INFORM`; in these cases the object and value must be bound.

Monitors allow constraints to be set for measurements, times, time intervals, and behavior. In general, a constraint looks syntactically like the following:

```
(<op> <value>)
```

where `<op>` is a Boolean function name and `<value>` is a value or value expression. For example, a constraint on the altitude of an aircraft measurement might look like the following:

```
(:measurement (alt ac1 = ?val)  :value (> 32000))
```

The `<value>` can be any Lisp expression such as:

```
(:measurement (alt ac1 = ?val) :value (> (calculate-minimum-altitude)))
```

This is useful, for example, for setting timestamp constraints based on the time of the current task:

```
(:measurement (alt ac1 = ?val)
 :timestamp (= (+ (start-of +this-task+) P45S)))
```

The operator can also be any Lisp Boolean function, although the following operators (>, <, =, <=, >=) allow the monitoring system to set constraints on intervals and values more efficiently.

More than one constraint can be specified as shown in the following example:

```
(:measurement (alt ac1 = ?val) :value (>= 32000) (<= 34000))
```

The previous example can be rewritten in shorter syntactic form, which may be more readable as well:

```
(:measurement (alt ac1 = ?val)     :value (:range 32000 34000))
```

An even shorter syntactic form for this example is as follows, although the variable is not captured:

```
(:measurement (alt ac1 = 33000 +/- 1000)
```

The special syntactic forms are as follows:

```
(:range <min> <max>) == (>= <min>) (<= <max>))
(:min <min>)         == (>= <min>)
(:max <max>)         == (<= <max>))
```

4.8.2 Times associated with monitors

The following different timestamps are associated with monitors:

- the time a monitor is started
- the time a monitor *succeeds*
- the time(s) a monitor is checked

Different intervals are also associated with monitors:

- the constraints on the interval for data collection on linear regression
- the constraints on the timestamp of a measurement or estimation monitor
- the interval of an episode
- the constraints on the start time of the episode, the end time of the episode, and the duration of the episode

4.8.3 :measurement

Type: condition clause

Syntax (long version):

```
(:measurement [<tag>] (<attr> <obj/v> = <val/v> [+/- <val/v>])
     [:timestamp <constraint>*]*
     [:value     <constraint>*]*
     [:object    <constraint>*]* ) |
```

Syntax (short version): `(<attr> <obj/v> = <val/v>)`

A measurement monitor looks for the value of a state variable at some point in time (a measurement). The measurement can be described in one of the following ways:

- *value* constraints where the value of the state variable is constrained (for example, has a particular value, is within a range, etc.)
- *object* constraints where the object of the state variable is constrained
- *timestamp* constraints where the point in time of the behavior is constrained

The state variable history is a time series, that is an ordered set, of measurements on the same state variable. No assumption is made that the measurements are equally spaced in time. Nor are assumptions made that measurement values change between measurements. Note that logging of state variables is required.

The state variable memory is a collection of state variable histories. Each agent has its own state variable memory.

The timestamp defaults to the start of the current step.

4.8.3.1 Measurement examples: Monitoring for future conditions

The following measurement monitor looks for a measurement in the future where the altitude of an aircraft is 32000:

```
(:measurement m1 (alt ac1 = 32000))
```

The timestamp is assumed to occur after the start of the current task, so the above is equivalent to:

```
(:measurement m2 (alt ac1 = 32000)
  :timestamp (>= (start-of +this-task+))) ;; not necessary
```

4.8.3.2 Monitoring for conditions with variable binding

The following measurement monitor looks for any future altitude report:

```
(:measurement m3 (alt ac1 = ?value))
```

The following monitor looks for a measurement with a value between [30000,40000] without binding the variable.

```
(:measurement m4 (alt ac1 = ?)
  :value (:range 30000 40000))
```

Here is yet another way to write m1 from above:

```
(:measurement m5 (alt ac1 = ?)
  :value (= 32000))
```

4.8.3.3 Monitoring for past conditions

By setting the timestamp constraints, the following monitor becomes a query for past conditions:

```
(:measurement q1 (alt ac1 = ?val)
  :value (< 32000)
  :timestamp
   (:range (- (start-of +this-task+) P1M) (start-of +this-task+)))
```

You can also set the range to monitor for past or present conditions, as in the following example:

```
(:measurement q1 (alt ac1 = ?val)
  :value (< 32000)
  :timestamp
   (:range (- (start-of +this-task+) P1M) (+ (start-of +this-task+) P1m)))
```

4.8.4 :estimation

Type: condition clause

Syntax:

The full syntax for estimation monitors is as follows:

```
(:measurement [<tag>] (<attr> <obj/v> = <val/v>)
      :estimation <estimator>
     [:timestamp <constraint>*]*
     [:value     <constraint>*]*
     [:object    <constraint>*]* )
```

where `<estimator>` is either a `<persist-estimator>` or a `<linear-regression-estimator>`.

A `<persist-estimator>` uses the form:

```
(:persist [:with-timeout <duration>:+pos-infinity+])
```

and a `<linear-regression-estimator>` uses the form:

```
(:linear-regression
    [:minimum-points <int>:2]
    [:maximum-error <real>:infinity]
    [:maximum-difference <real>:infinity]
    [:minimum-frequency <real>:infinity]
    [:start <time-point>:+beginning-of-time+]
    [:end <time-point>:+current-time+] )
```

The `:start` and `:end` parameters for `:linear-regression` describe the interval from which the linear regression equation is generated. The timestamp constraint defaults to `(>= (start-of +this-task+))`.

An estimation monitor has the essential form of a measurement monitor, but state variable values are estimated from observed values. Thus, the same constraints of measurement monitors are relevant to estimation monitors:

- *value* constraints where the value of the state variable is constrained (for example, has a particular value, is within a range, etc.)
- *object* constraints where the object of the state variable can be constrained
- *timestamp* constraints where the point in time of the behavior is constrained

In addition, one of the following estimation methods is required:

- A *persistence estimator* estimates values by assuming that observed values remain valid for a duration of time. For example, one might assume that once a light switch has been turned on it remains on, or one might assume that the measured altitude of an aircraft is valid for a period of time.
- A *linear regression estimator* estimates values based on linear regression of observed values.

Note that the only syntactic difference between measurement monitors and estimation monitors is the presence of an `:ESTIMATION` form.

4.8.4.1 Monitoring for future conditions, based on persistence

This example looks for an observation in the future where the altitude of an aircraft is 32000. The main difference between this example and the analogous non-estimated monitor is that if the altitude has just been measured at 32000 prior to the start of the monitor, the monitor will succeed.

```
(:observation e1 (alt ac1 = 32000)
  :estimation (:persist))
```

4.8.4.2 Monitoring with variable binding, with a persistence monitor

In this example, if a value has not been seen in the past second, the monitor looks for a future altitude report.

```
(:observation e2 (alt ac1 = ?value)
  :estimation (:persist :with-timeout P1S))
```

4.8.4.3 Monitor for an observation

This example monitors for an observation with a value between [30000,40000] using linear regression of values from the last 5 minutes to estimate it.

```
(:observation e3 (alt ac1 = ?val)
  :value (:range 30000 40000)
  :estimation (:linear-regression
                :start (- (start-of +this-task+) P5M)))
```

4.8.4.4 Estimates into the future

Estimation functions can also estimate into the future. For example, in the following monitor, assume that the task starts at 0 seconds and a measurement on (`alt ac1`) occurs every 5 seconds. Since the `:PERSIST` estimation rule, by default, states that values persist until contradicted, if any measurement comes in between 0 and 10 seconds (even before 5 seconds), this monitor will succeed.

```
(waitfor
  (:measurement
    o1.6b
    (alt ac1 = ?var)
    :estimation (:persist)
    :timestamp (= (+ (start-of +this-task+) P10s))))
```

4.8.5 :episode

Type: condition clause

Syntax:

The full syntax for episode monitors is as follows:

```
(:episode [<tag>] (<attr> <obj/v>)
    :quality (:no-constaints) | (:minimum-sample-interval <msi-constraint> )
              | (:msi <msi-constraint> )
    [:timing <timing-constraint>*]*
    [:value <constraint>*]*
    [:first-value <constraint>*]*
```

```
[:last-value <constraint>*]*
[:object <constraint>*]*
[:stats <stat-constraint>*]*
[:trend <trend-constraint>*]* )
```

where `<msi-constraint>` is a `<constraint>` or a `<duration>` (`<duration>` is equivalent to `<= <duration>`)

`<timing-constraint>` can take one of the following forms:

- `(:start <constraint>*)`
- `(:end  <constraint>*)`
- `(:earliest-start <time-spec>)`
- `(:es <time-spec>)`
- `(:latest-start  <time-spec>)`
- `(:ls <time-spec>)`
- `(:earliest-end  <time-spec>)`
- `(:ee <time-spec>)`
- `(:latest-end  <time-spec>)`
- `(:le <time-spec>)`
- `(:duration <constraint>*)`

`<stat-constraint>` can take one of the following forms:

- `(:mean <constraint>*)`
- `(:count <constraint>*)`
- `(:stddev <constraint>*)`
- `(:variance <constraint>*)`
- `(:sum <constraint>*)`
- `(:difference <constraint>*)`

`<trend-constraint>` can take one of the following forms:

- `(:rate <constraint>*)` for units/time, e.g. `(> (20 P1s))`
- `(:rate :increasing |`
  `:decreasing |`
  `:non-increasing |`
  `:non-decreasing)`
- `(:step <constraint>*)` for constraints on differences in contiguous values

- `(:step :increasing |`
  `:decreasing |`
  `:non-decreasing |`
  `:non-decreasing )`

- `(:frequency <constraint>*)` for units/time

- `(:custom <interval-fn>)` which is a Lisp function of `history+interval+bindings` that returns a Boolean value.

An episode monitor looks at the behavior of a state variable over some interval. The behavior can be described as one of the following:

- *value* constraints where the value of the state variable is constrained at each point in the interval (for example, remains constantly at some value, is within a range, etc.)

- *first value* and *last value* constraints where constraints can be placed on the first and last values

- *timing* constraints where the interval of the behavior is constrained. The interval can be constrained with respect to its start time, its end time, or its duration; or some combination

- *statistical* constraints where the values of the state variable within some interval, considered as a set, are constrained (for example, the average value must be above some number)

- *trend* constraints where the values of the state variables within some interval, considered as a time series, are constrained. These include constraints on rate of change, frequency, and step (differences between two observation values)

- *object* constraints where the object of the state variable can be constrained at each point in the interval

4.8.5.1 Holding monitor

This example looks for an interval in the future in which the airplane is holding at an altitude of 32,000 feet for at least 10 seconds. The sampling rate must be at least 1 observation per second.

```
(:episode h1 (alt ac1)
  :quality (:msi P1s)
  :value  (= 32000)
  :timing (:start (> (start-of +this-task+))) ;; not necessary
          (:duration (>= P10s)))
```

In this holding example, a light switch is in the on position throughout a particular interval; persistence is assumed by setting constraints on the start time.

```
(:episode h3 (position ?switch)
  :quality (:no-constraints)
  :timing (:es <t_1>)
          (:ls <t_2>)
          (:end   (= <t_3>))
  :value  (eql on))
```

4.8.5.2 Holding within a range

This example is similar to the holding example h1, but in this case the value can range between (30000, 34000] with any sample interval.

```
(:episode h2 (alt ac1)
  :quality (:no-constraints)
  :value  (>= 30000) (< 34000)
  :timing (:start (> (start-of +this-task+))) ;; not necessary
          (:duration (>= P10s)))
```

4.8.5.3 Holds monitor

This example is similar to the holding monitor h3 but with appropriate timing constraints. In this case, the light switch must be on for more than 5 minutes, ending some time after the start of the current task.

```
(:episode h4 (position ?switch)
  :quality (:msi P1s)
  :timing (:start (> +beginning-of-time+)) ;; maybe...
          (:end   (> (start-of +this-task+)))
          (:duration (> P5m))
  :value  (eql on))
```

4.8.5.4 Increasing monitor

In this example, the monitor looks for an interval in the future in which the airplane is rising for at least 10 seconds at an altitude greater than 32,000 feet. The sampling rate must be at least 1 observation per second.

```
(:episode i1 (alt ac1)
 :quality (:msi P1s)
 :timing (:duration (>= P10s))
 :trend (:rate :increasing)
 :value (> 32000))
```

This example is similar to the previous one (i1), but in this case the *average* value in the interval should be greater than 32000 feet.

```
(:episode i2 (alt ac1)
 :quality (:msi P1s)
 :timing (:duration (>= P10s))
 :trend (:rate :increasing)
 :stats (:mean (> 32000)))
```

4.8.5.5 Monotonically increasing monitor

This example is similar to the increasing monitor i1, but in this case the values must be rising monotonically.

```
(:episode i1 (alt ac1)
 :quality (:msi P1s)
 :timing (:duration (>= P10s))
 :trend (:step :increasing)
 :value (> 32000))
```

4.8.6 :and

Type: condition clause

Syntax: `(:and [<tag> <condition>+  [:constraint <episode-constraint>]*)`

An :and monitor succeeds when all of its sub-monitors succeed; that is, when each of its conditions are met. Optionally, any additional constraints must be met. Conditions are checked in left-to-right order; if a condition is not met, conditions to the right are not checked.

4.8.7 :or

Type: condition clause

Syntax: `(:or [<tag> <condition>+  [:constraint <episode-constraint>]*)`

An :or monitor succeeds when at least one of its sub-monitors succeeds; that is, when each of its conditions are met. Conditions are checked in left-to-right order; if a condition is met, conditions to the right are not checked Optionally, any additional constraints must be met.

4.8.8 :not

Type: condition clause

Syntax: `(:not [<tag> <condition>  [:constraint <episode-constraint>]*)`

A :not monitor succeeds when its monitor fails; that is, its condition is not met. Note that conditions are check at the time the monitor is created.

4.8.9 :in-order

Type: condition clause

Syntax: `(:in-order [<tag> <condition>+  [:constraint <episode-constraint>]*)`

An :in-order monitor succeeds when all of its sub-monitors succeed; that is, when each of its conditions are met. In addition, the conditions must succeed in the order specified. More precisely, monitor M succeeds only if for every m_i and m_j which are adjacent sub-monitors of M, m_i succeeds and m_j succeeds, and the end

of m_i < start of m_j) Optionally, any additional constraints must be met. Conditions are checked in left-to-right order; if a condition is not met, conditions to the right are not checked. In order monitors can have delay monitors internally; that is, between two other (non-delay) monitors.

4.8.10 Allen monitors

Type: condition clause

Syntax: `(<allen> [<tag> <condition>+  [:constraint <episode-constraint>]*)`

Where `<allen>` one of the following: `:contains, :finishes, :starts, :before, :meets, :overlaps, :cotemporal, :during, :finished, :started, :after, :met,` or `:overlapped.`

An Allen monitor succeeds when all of its sub-monitors succeed; that is, when each of its conditions are met. In addition, the conditions must succeed in the order specified. More precisely, monitor *M* succeeds only if for every m_i and m_j which are adjacent sub-monitors of *M*, m_i succeeds over interval I_i and and m_j succeeds over interval I_j. I_i <op> I_j, where <op> is the Allen operator associated with the monitor type. Optionally, any additional constraints must be met. Conditions are checked in left-to-right order; if a condition is not met, conditions to the right are not checked.

4.8.11 :delay

Type: condition clause

Syntax: `(:delay <duration-spec-1> [<duration-spec-2>])`

A delay monitor waits for a delay of at least `<duration-spec-1>` time units, and at most `<duration-spec-2>` time units, if specified. Delay monitors may not appear in :and, :or, or Allen monitors. They may appear in in-order monitors as described above.

4.8.12 :timestamp

Type: condition clause

Syntax: `(:timestamp <time-spec-1> [<time-spec-2>])`

A delay monitor waits until least `<time-spec-1>` time occurs, and at most until `<time-spec-2>` time occurs, if specified. Timestamp monitors may not appear in :and, :or, or Allen monitors, or in-order monitors.

4.8.12.1 :in-order monitor examples

The following monitor specifies an order for submonitors.

```
(waitfor
  (:in-order s1
    (says john = take)
    (says john = the)
    (says john = cannoli)))
```

The monitor succeeds with the following event stream:

```
(says john = take)
(says john = the)
(says john = yummy)
(says john = cannoli)
```

The following monitor waits 2 seconds after the light turns green.

```
(wait-for
  (:in-order c2
   (color light-1 = green)
   (:delay P2s)))
```

In this example, the monitor looks for an aircraft that rises, holds steady for a while, then descends.

```
(waitfor
  (:in-order i1
   ;; increase for a while
    (:episode e1 (alt ac-1)
     :timing (:duration (>= <?a-while>))
     :trend (:rate :increasing))
   ;; hold steady for a while
    (:episode e2 (alt ac-1)
     :timing (:duration (> <?a-while>))
     :stat (:difference 30))
   ;; decrease for a while
    (:episode e3 (alt ac-1)
     :timing (:duration (>= <?a-while>))
     :trend (:rate :decreasing))))
```

4.8.12.2 :and monitor examples

The following `and` monitor waits for four subsystems to be in a “safed” state.

```
(waitfor
  (:and (status ro = safed)
        (status pps = safed)
       (status pbbwp = safed)
       (status aes = safed)))
```

4.9 Primitive procedures

A *primitive procedure* (commonly called a *primitive*) is a special kind of PDL procedure. It defines a fundamental task that cannot be decomposed into other tasks. Primitives are used to represent low-level behaviors or operations of an agent. Like non-primitive procedures, they have an *index* clause, which is used as the basis for selection for a task.

The term *operator* is used for behaviors that are low-level from the point of view of a particular domain or task model. For example, in some models of human-computer interaction, behaviors such as pushing a button and moving a mouse to a target location might be considered operators. Operators are generally represented as primitive procedures in PDL.

Primitives also serve as a bridge between PDL and the underlying Common Lisp language – arbitrary Lisp code can be specified in most clauses of a primitive. This bridge serves to provide an interface between Apex and foreign applications, or any other software whose interface is accessible in Lisp.

4.9.1 primitive

Syntax: `(primitive {<index>} <primitive-clause>*)`

As in non-primitive procedures, the first argument to a primitive can be its index, for example:

```
(primitive (reach ?object with ?hand)
  …)
```

or, alternately, the index can be specified with an index clause:

```
(primitive
  (index (reach ?object with ?hand))
 …)
```

The difference is only one of style. The index is the only required argument or clause in a primitive. The other primitive clauses are described below.

4.9.2 duration

Type: primitive-clause

Syntax: `(duration <time-expression>)`

This clause specifies the actual (prescribed) duration for a primitive. The primitive will terminate when this time has transpired.

4.9.3 update

Type: primitive-clause

Syntax: `(update <duration-expression> <lisp-form>+)`

While a primitive is executing it may be *updated* at regular time intervals as specified by arbitrary Lisp code. The first argument of this clause specifies the time interval. The remaining arguments are Lisp forms that will be evaluated in the order listed when the primitive is updated.

4.9.4 on-start

Type: primitive-clause

Syntax: `(on-start <lisp-form>)`

This clause specifies arbitrary Lisp forms that will be evaluated in the given order upon initiation of the primitive.

4.9.5 on-completion

Type: primitive-clause

Syntax: `(on-completion <lisp-form>)`

This clause specifies arbitrary Lisp forms that will be evaluated in the given order upon termination of the primitive.

4.9.6 locals

Type: primitive-clause

Syntax: `(locals (<var1> <exp1>) … (<varn> <expn>))`

The `locals` clause specifies variables that are bound locally within the primitive procedure. Each entry in the clause is a list consisting of a variable name followed by a Lisp expression denoting its initial value. This expression is evaluated when the primitive begins execution. The variable may be reassigned in any other clause within the primitive procedure.

4.9.7 profile

Type: primitive-clause

The syntax and semantics of this clause is identical with that for non-primitive procedures and is described in section 4.3.3.

4.9.8 return

Type: primitive-clause

Syntax: `(return <lisp-form>)`

This clause specifies arbitrary Lisp forms that will be evaluated in the given order upon termination of the primitive.

4.10 Miscellaneous features

4.10.1 Agent's initial task

You can specify an agent's initial task with the `:initial-task` initarg, whose value is a procedure invocation that defaults to `'(do-domain)`. For example,

```
(let ((jack (make-instance 'human :name 'jack
              :initial-task '(play roshambo 3 times)))))
```

If you do not specify an `:initial-task` when creating an agent instance, you must define a PDL procedure with the index `(do-domain)`.

4.10.2 PDL Partitions (Bundles)

A PDL procedure can be associated with a named category (called a *partition* or *bundle*) that can be referred to during agent creation. This allows different agents to have different skill sets. PDL procedures can be optionally assigned to a partition using the `in-apex-bundle` form, which appears at the top level of an application file and looks like:

```
(in-apex-bundle <name>)
```

where `<name>` is a keyword symbol by convention, for example:

```
(in-apex-bundle :best-strategy)
```

After such a form appears, all subsequent procedures defined until either the end of file or the next `in-apex-bundle` form, will be assigned to the specified bundle.

You can assign a given bundle to an agent using the optional `:use-bundles` argument in the `make-instance` form for the agent. For example, let's give Fred the best strategy procedures:

```
(make-instance 'agent :name "Fred" :use-bundles '(:best-strategy))
```

This agent will have a *procedure library* consisting only of the procedures in the :best-strategy bundle. An agent may use any number of bundles. If no bundles are specified, the agent will have all procedures defined in the application.

NOTE: If an agent uses more than one bundle, and two or more contain procedures with the same index, only one of the procedures will be placed in the agent's procedure library; which one is unspecified. You should partition your bundles so that this won't happen.

See the Roshambo simworld (`<apex-installation-directory>/examples/roshambo.lisp`) for a good example of the use of bundles.

5 Apex Programming Reference

This chapter of the manual, still under development, contains detailed descriptions the Apex Application Programming Interface (API), aside from PDL. For information on PDL, see Chapter 4.

Currently, this chapter contains information on using the `defapplication` form to define Apex applications, creating agents, and associating diagrams with objects in Apex applications.

5.1 defapplication

The `defapplication` form defines and creates an Apex application. It specifies the name and type of the application, its files and libraries, and its operational interface. The format of `defapplication` is as follows.

```
(defapplication <name>
  [:type <type> ]
  :libraries (<name1> .. <nameN>)
  :files (<file1> .. <fileN>)
  :init <form>
  :reset <form>
  :start <form>
  :stop <form>
  :step <form>
  :restart <form>
  :time-function <function>
  :time-setter <function>
  :time-resetter <function>
  :scheduler <function>
  :pause-time-setter  <function>
  :pause-cycle-setter <function>
  :pause-interval-setter  <function>)
```

The first argument `<name>` is a string that names the application.

The :type clause, which is optional, specifies the kind of application to create. It has four legal values:

7. `:native-sim`, which creates a native simulation application.

8. `:realtime`, which creates a realtime application.

9. `:foreign-sim`, which creates a foreign simulation application.

10. A symbol (without quote) naming a defined class. This is how a user-defined application type is specified. The specified class should be a subclass of `application`.

The <name> arguments for the :libraries are strings naming libraries. These libraries are loaded in the order specified.

The <file> arguments of the :files clause are strings representing either relative or absolute pathnames. If the file's extension is omitted, a Lisp extension is assumed (.lisp, .lsp, or .cl). The files are loaded in the order specified. Note that files and libraries may themselves load other files or libraries, so the lists specified in :libraries and :files do not necessary name all the files and libraries that comprise the application.

The <form> arguments of the other clauses are Lisp expressions that will be the bodies of the respective interface functions. When the function is called, the form is evaluated in the context of the current top-level Lisp environment (that is, defapplication does not create a lexical scope).

The following four clauses specify the time and scheduling mechanisms for the application. Please note that nspecified behavior will result if they are used with native simulation or realtime applications, since these applications have their own time and scheduling mechanisms. Each clause takes a symbol naming a function.

- The :time-function clause specifies a function of no arguments that returns the current time for the application, when current-time is called.
- The :time-setter clause specifies a function of one argument, an integer representing milliseconds, that is used to set the application's time, when set-clock is called.
- The :time-resetter clause specifies a function of no arguments that resets the application time, when reset-clock is called.
- The :scheduler clause specifies a function, of one or more arguments (the first is a duration expression, and the rest are Lisp forms), that is used to schedule events for the application, when schedule is called.

The :pause-time-setter, :pause-cycle-setter, and :pause-interval-setter clauses specify functions of one argument that set a pause time, number of cycles to execute before pausing, and a time interval to transpire between automatic pauses, respectively. The <function> argument of these clauses is a symbol naming this function.

The clauses in defapplication may appear in any order. While all are optional, :init is required for a native simulation application and :start is required to make a real-time application invokable from Sherpa.

5.2 Application interface

The Lisp interface for manipulating Apex applications, and its correspondence to Sherpa, is described as follows. It's important to note that all of these operations except for initialization are predefined (and cannot be redefined) for native simulation applications. For other application types, these operations have no predefined semantics. They behave as specified in the application's defapplication form, and are optional.

(initapp) initializes the application, as specified in the :init clause of its defapplication form. If there is no :init clause, calling initapp does nothing. Sherpa does not provide a means to initialize an application explicitly. In native simulation applications, initialization is performed automatically as needed when the application is started, and generally means creating all the objects in the application.

(resetapp) initializes the application, as specified in the :reset clause of its defapplication form. If there is no :reset clause, calling resetapp does nothing. Clicking the Reset button in Sherpa calls this function.

(startapp) starts the application, as specified in the :start clause of its defapplication form. If there is no :start clause, calling startapp does nothing. Clicking the Start button in Sherpa calls this function.

(stopapp) stops the application, as specified in the :stop clause of its defapplication form. If there is no :stop clause, calling stopapp does nothing. Clicking the Stop button in Sherpa calls this function.

(stepapp) advances the application one step or operating cycle, as specified in the :step clause of its defapplication form. If there is no :step clause, calling stepapp does nothing. Clicking the Step button in Sherpa calls this function, *unless* the application has a pause time, cycle, or interval set (as specified by the :pause-time-setter, :pause-cycle-setter, and :pause-interval-setter clauses of defapplication, respectively) – in this case Sherpa provides no interface to stepapp.

(stepapp/pause) advances the application to its (next) pause point, as specified by the :pause-time-setter, :pause-cycle-setter, and :pause-interval-setter clauses of its defapplication form. If none of these clauses are specified, or if no subsequent pause point exists, calling stepapp/pause does nothing and the Step button in Sherpa calls stepapp. Otherwise, the Step button in Sherpa calls stepapp/pause.

(reloadapp) reloads the current application. It is invoked by the Reload button in Sherpa.

5.3 Time and Scheduled Events

Apex applications have the following Lisp interface for time-related behaviors.

(current-time) returns the current application time in milliseconds. In native simulation applications, this comes from the simengine's clock. In realtime applications, this is the system time. In other kinds of applications, it is the value returned by the :time-function clause of defapplication.

(set-clock <milliseconds>) updates the application's time, as specified by the :time-setter clause in defapplication. Unspecified behavior will result if you call this function in a native simulation or realtime application.

(reset-clock) resets the application's time, as specified by the :time-resetter clause in defapplication. Unspecified behavior will result if you call this function in a native simulation or realtime application.

(schedule <delay> <form1> .. <formN>) schedules the specifed Lisp forms to be evaluated, in the order given, after passage of the time in <delay>, which must be a duration expression.

5.4 Agents

In Apex applications, an *agent* is the autonomous entity whose behavior is specified by PDL. An application may define any number of agents, though at least one is usually needed to model anything interesting.

You create agents with make-instance. For example,

```
(make-instance 'agent :name "Fred" :initial-task '(drive to work))
```

All keyword arguments are optional, so you can create an agent with simply `(make-instance 'agent)`. The available keyword arguments are defined as follows.

- `:name` is a string or symbol that names the agent.
- `:initial-task` is a list expression specifying the first procedure the agent executes. For example, the initial task `'(drive to work)` might cause the procedure whose index is `(drive to ?destination)` to be selected.
- `:use-bundles` is a list of symbols (usually keyword symbols) that specifies which procedure bundles the agent will use.
- `:routers` is a list of publish/subscribe routers to which the agent will subscribe.

Agents may be created at any time in an application, though in general they are created in the initialization function . See `roshambo.lisp` and other sample applications for examples of agent instantiation.

Other functions related to agents:

- `(find-agent <name>)` takes a symbol or string and returns the agent having that name, or `nil` if no such agent exists.

5.5 Inform

In general there are three kinds of communication in Apex.

1. Within agents. This is provided by cogevent, described in 4.3.4.
2. Within in application. This is provided by the publish-subscribe mechanism, described in this section.
3. Between an Apex application and an external application. This is provided by the narration facility described in 4.2.15.

Apex provides a simple publish-subscribe mechanism as one means for entities to communicate with each other and to receive information from the external world. This mechanism consists of a *router* to which an entity may *publish* information and to which an entity may *subscribe*. This information is treated as an event and may appear in event traces (see 3.7.1 for more information on event tracing). When an event is published to a router, it is immediately sent to all subscribers of the router (but not back to the publisher if it happens to subscribe to the router). For convenience, there is a default global router[8] to which all agents are automatically subscribed.

Subscribers must be instances of a class that has *ps-mixin* as a superclass. This class must specialize the `deliver` method used to deliver events to an entity:

[8] NOTE: there is is one other default router, reserved for use by the narration facility described in section 4.2.14.

```
(defmethod deliver ((recipient ps-mixin) message &key trigger-asa)
```

This default method, defined in Apex, generates an error (i.e. there is no default delivery method for subclasses of *ps-mixin*). The Boolean valued `trigger-asa` keyword argument is agent-specific and controls whether or not the agent's ASA cycle is activated upon message delivery.

The publish-subscribe mechanism is provided by the class ps-router, whose interface is defined as follows.

Create a router (example):

```
(defparameter *my-router* (make-instance 'ps-router))
```

Subscribe to a router:

```
(subscribe <ps-mixin> <ps-router>)
```

Publish to a router:

```
(publish <event> [:router <router>] [:author <ps-mixin>]
                 [:trigger-asa nil])
```

If the `:router` keyword argument is omitted, the event will be published to the default router. The `:author` keyword argument exists to prevent the authoring entity from receiving the event it publishes when it subscribes to the same router. The `:trigger-asa` keyword argument is agent-specific and determines if an ASA cycle is triggered in subscribers that are agents.

Reset a router (clear its subscribers list):

```
(reset <ps-router>)
```

Reset all existing routers:

```
(reset-ps-routers)
```

5.6 Diagrams

Any object in Apex that is a subclass of `appob` may have a diagram view associated with it. The default diagram view for an object in Sherpa is a box-and-arrow diagram showing the object as a box with arrows to boxes representing the children of the object.

A diagram for an object can be defined several different ways. They can be complex SVG diagrams, simple wireframe diagrams, or something in the middle composed hierarchically using polygons, circles, labels, etc.

You see examples of the different types of diagrams in the following sample applications:

- simple wireframe diagram: `goms/atm/cpm.lisp`
- hierarchical diagram with colored rectangles, circles, labels: `goms/atm/cpm-hier.lisp`
- complex SVG diagram: `goms/atm/cpm-photo.lisp`

To see the diagram, run the application, then right-click on the ATM object in the Object Tree and select Diagram.

5.6.1 Hierarchical diagrams

You can build a diagram in Apex by setting the graphical-object slot of an object instance that is a subclass of appob. A full diagram for an object consists of the set of graphical objects for the object and its descendants. For example, the diagram for an ATM consists of an ATM graphical object, a keypad graphical object, a set of button graphical objects for keys in the keypad, etc.

The basic graphical objects available for constructing a hierarchical diagram in Apex include rectangles, circles, polygons, lines, text, paths, and images. These objects are modeled after the corresponding SVG elements. More information about SVG elements is available at http://www.w3.org/TR/SVG11/eltindex.html.

Slot values are the same as those in the SVG spec except as noted below (some esoteric slots are omitted from the Lisp implementation). Initargs and accessors are identical to slot names.

The following function constructs objects for a yes/no keypad and creates a diagram that has a grey keypad with a green "YES" button and a red "NO" button.

```
(defun create-yes-no-keypad (locale)
  (let ((keypad (make-instance 'interface-object :name "keypad"
                               :locale locale))
    (yes (make-instance 'interface-object :name "yes" :locale locale))
    (no (make-instance 'interface-object :name "no" :locale locale)))
     (assemble keypad)
     (assemble yes :component-of keypad)
     (assemble no :component-of keypad)

     ;;grey rectangle for keypad
     (setf (graphical-object keypad)
       (make-instance 'rect
                   :x 0 :y 0 :width 350 :height 50
                   :fill "grey" :fill-opacity 1 :stroke "black"))
     ;;green rectangle for yes
     (setf (graphical-object yes)
       (make-instance 'rect
                   :x 50 :y 10 :width 100 :height 30
```

```
                :fill "green" :fill-opacity 1 :stroke "black"))
    ;;place a "YES" label in the center of the rectangle
    (auto-label (graphical-object yes) "YES")

    ;;red rectangle for no
    (setf (graphical-object no)
     (make-instance 'rect
                :x 200 :y 10 :width 100 :height 30
                :fill "red" :fill-opacity 1 :stroke "black"))
    ;;place a "NO" label in the center of the rectangle
    (auto-label (graphical-object no) "NO")
    ))
```

5.6.2 Wireframe diagrams

There are convenience functions available for building a simple wireframe diagram containing either rectangles or polygons. These methods are available for objects that are subclasses of visob or interface-object. For interface-object subclasses, the resulting diagram will depict the two-dimensional rectangles described by the pos and dimensions of the interface-object. For visob subclases, the faces slot defines the three dimensional wireframe. From Sherpa, you can display either the XY, XZ or ZY view of the three dimensional wireframe.

The following function constructs objects for a yes/no keypad and creates a 2D wireframe diagram for the keypad and the yes/no buttons.

```
(defun create-yes-no-keypad (locale)
  (let ((keypad (make-instance 'interface-object :name "keypad" :pos '(0 0)
:dimensions '(350 50) :locale locale))
    (yes (make-instance 'interface-object :name "yes" :pos '(50 10)
:dimensions '(100 30) :locale locale))
    (no (make-instance 'interface-object :name "no" :pos '(200 10)
:dimensions '(100 30) :locale locale)))
    (assemble keypad)
    (assemble yes :component-of keypad)
    (assemble no :component-of keypad)

    ;;simple wireframe diagram (uses pos and dimensions for sizing wireframe
      rectangles)
    ;;rectangle for keypad with 2 inner rectangles for yes and no buttons
    (create-wireframe-graphical-object keypad)
    (create-wireframe-graphical-object yes)
    (create-wireframe-graphical-object no)))
```

The following function constructs a 50x50x50 cube object and creates a three-dimensional wireframe diagram for it.

Note the use of `:auto-update t` in this example. The `:auto-update` keyword causes the diagram object to update its size and position when ever the faces slot of the cube is modified. Redisplaying the diagram from Sherpa will show the new cube with updated coordinates and size.

```
(defun create-moving-cube (locale)
```

```
  ;;creates 3d cube 50x50 centered at 0,0,0
  (let ((cube (make-instance 'physob :name "cube"  :locale locale
          :faces '(((0 0 0) (0 50 0) (50 50 0) (50 0 0))  ;;bottom face
                  ((0 0 0) (0 0 50) (0 50 50) (0 50 0))  ;;back face
                  ((0 0 0) (50 0 0) (50 0 50) (0 0 50))  ;;left face
                  ((0 0 50) (50 0 50) (50 50 50) (0 50 50))  ;;top face
                  ((50 0 0) (50 50 0) (50 50 50) (50 0 50))  ;;front face
                  ((0 50 0) (0 50 50) (50 50 50) (50 50 0))  ;;right face
                             ))))
     (assemble cube)

     ;;auto-update cube (if we modify faces slot, graphical object will
update to new coords)
     (create-wireframe-graphical-object cube :auto-update t)))
```

5.6.3 File-based SVG diagrams

To show a complex diagram, you can construct an SVG file with your favorite SVG tool.

To enable clicking on portions of the diagram in Sherpa, you'll need to give the SVG objects IDs which correspond to the names of the objects in Apex.

For example, the following code creates agents named Jack and Jill and assigns an SVG file diagram to the Jill object. When viewing the diagram for Jill in Sherpa, you'll be able to click on the rectangle and inspect the object Jill, or click on the circle and inspect Jack. Note that the use of explicit pathnames in Lisp in not recommended and is only used as an example. For more elegant pathname specification, see Lisp documentation on logical pathnames.

```
(let* ((locale (make-instance 'locale :name "Room"))
       (jill (make-instance 'agent :name "Jill"
                    :use-bundles '(:jill)
                    :initial-task '(handle phone)
                    :locale locale))
       (jack (make-instance 'agent :name "Jack"
                    :use-bundles '(:jack)
                    :initial-task '(handle phone)
                    :locale locale)) )
    (setf (graphical-object jill)
     (make-instance 'file-svg-object
                 :filename "c:/tmp/sample.svg")))
```

Contents of `sample.svg`:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="4cm" height="4cm" viewBox="0 0 200 200"
     xmlns="http://www.w3.org/2000/svg" version="1.1">
```

```
  <title>Some sample svg</title>
  <rect id="Jill" x="10" y="10" width="100" height="50" fill="yellow"
stroke="green" />
  <circle id="Jack" cx="40" cy="100" r="30" fill="none" stroke="red" />
</svg>
```

Glossary

ASA—Action Selection Architecture. The algorithm Apex uses to generate behavior. Input to the algorithm consists of events that the agent might respond to along with a set of predefined PDL procedures. The ASA outputs commands to resources.

CPM—The automatic scheduling of low-level cognitive, perceptual, and motor (CPM) resources that underlie actions. Freed, Matessa, Remington, and Vera (2003).

Emacs—A text editor and software development environment with support for Lisp programming.

GOMS—A formal language for representing how human operators carry out specified routine tasks. GOMS consists of four constructs: goals, operators, methods, and selection-rules (hence the GOMS acronym). Freed and Remington (2000a).

GOMS+—A GOMS implementation that incorporates several capability extensions. As with GOMS, methods in GOMS+ are action sequences. Behaviors that are contingent or off critical-path (such as those needed to handle failure) cannot be represented. Freed and Remington (2000a).

PDL—Procedure Definition Language. A formal language used to specify the behavior of Apex agents.

PERT—The Program Evaluation and Review Technique (PERT) was developed by the US Navy to plan and control a missile program. PERT charts have a probabilistic approach that allows estimates for the duration of each activity.

RAP—Reactive Action Package. A plan and task representation language introduced by James Firby (1989).

Appendix A: Getting Started with Apex

This appendix explains how to install and start Apex. It also explains how to get and install the third party software packages required or recommended for use with Apex. Familiarity with how to use your computer is assumed. The instructions in this appendix are organized by type of operating system:

- Macintosh
- Windows
- Linux

Overview of third party software packages

Although Apex 3.0 can be started and used in its standalone form, several third party software packages are either required or recommended to utilize all the features in Apex. All are freely available and downloadable at the URLs provided. For convenience they are also available at the Apex website: http://human-factors.arc.nasa.gov/apex .

Java Runtime Environment ® (JRE)

Sherpa requires (and will not start without) JRE® version 1.4, a product of Sun Microsystems. This software is already built into the Macintosh (OS 10.3.x) and is freely available for the other platforms. It can be downloaded from Sun's website: http://java.sun.com/.

Graphviz®

To display monitor diagrams, Sherpa uses Graphviz®, a product of Pixelglow. It is freely available, and downloadable from Pixelglow's website: http://www.pixelglow.com/.

Emacs

Gnu Emacs, a product of the Free Software Foundation, is highly recommended as a text editor for writing Apex applications. It is also possible to start and run Apex inside Emacs, for those who are comfortable with or wish to learn about using Emacs as a Lisp programming environment. Various versions of Emacs have been tested with Apex on different platforms, and these versions are listed in the platform-specific sections that follow. Information about Emacs is found at its home page: http://www.gnu.org/software/emacs/.

A variant of Gnu Emacs called Xemacs is also available. Currently Apex has not been tested with Xemacs. If you try Xemacs with Apex, please let us know how it goes!

Getting started on Macintosh

This section provides information about installing and getting started with Apex on the Macintosh platform.

Installing the software on Macintosh

Installing Apex on Macintosh

To install Apex, mount the Apex disk image (.dmg file) and drag the Apex folder to the location where you want to install Apex. For quick access you can drag the Apex and Sherpa icons to the Dock.

Installing Graphviz on Macintosh

Download Graphviz from the website listed in the overview at the beginning of this appendix and follow its installation instructions.

Installing Emacs on Macintosh

Gnu Emacs comes installed with Mac OS X. Start a Terminal (found in the Applications folder) and type `emacs` at the prompt.

This built-in version of Emacs has some limitations, in particular lack of mouse support. One better version of Emacs can be downloaded from http://www.webweavertech.com/ovidiu/emacs.html.

Starting Apex on Macintosh

Start Apex and Sherpa by double clicking on their icon in the installation folder (or by clicking on the Apex icon you've put in the Dock). *Apex must be started before Sherpa*. Refer to Chapter 2 for detailed instructions on using Apex.

You must quit Apex and Sherpa separately. To quit Sherpa, select Quit from the File menu, or close the window. To quit Apex, type `(exit)` at the Apex Listener prompt.

Note: It is not necessary to quit and restart Sherpa while using Apex. This is true even if Apex is quit (or dies) and is restarted. In these cases, simply reset Sherpa by selecting Reset from the File menu.

Getting started on Windows

This section provides information about installing and getting started with Apex on Windows PC's.

Installing the software on Windows

Installing Apex on Windows

Once you have unzipped the Apex distribution, Apex is essentially installed. You may place it anywhere on your computer. For convenience you may want to create shortcuts on your desktop for the Apex and Sherpa applications.

Installing the Java JRE on Windows

Before installing Apex, check whether Sun's Java Runtime Environment (JRE) version 1.4 or later is installed. If needed, download the JRE (e.g. from the Apex website or the website listed in the overview at the beginning of this appendix) and follow its installation instructions.

Installing Graphviz on Windows

Download Graphviz from the website listed in the overview at the beginning of this appendix and follow its installation instructions.

Installing Emacs on Windows

Download Emacs (e.g. from the Apex website or another source such as http://www.gnu.org/software/emacs/windows/) and follow its installation instructions.

Starting Apex on Windows

Start Apex and Sherpa by double clicking on their icon in the installation folder, or on their shortcut icons. *Apex must be started before Sherpa*. Refer to Chapter 2 for detailed instructions on using Apex.

You must quit Apex and Sherpa separately. To quit Sherpa, select Quit from the File menu, or close the window. To quit Apex, type `(exit)` at the Apex Listener prompt.

Note: It is not necessary to quit and restart Sherpa while using Apex. This is true even if Apex is quit (or dies) and is restarted. In these cases, simply reset Sherpa by selecting Reset from the File menu.

Getting started on Linux

This section provides information about installing and getting started with Apex on Linux computers.

Installing the software on Linux

Installing Apex on Linux

After unpacking Apex, place the Apex distribution in the directory of your choice. Add this directory to your search path. For example, if your shell is csh, add the following line to your .login or .cshrc file:

```
set path = (/home/joe/apex3-0  $path)
```

Installing the Java JRE on Linux

Before installing Apex, check whether Sun's Java Runtime Environment (JRE) version 1.4 or later is installed by typing `java -version`. If needed, download the JRE (e.g. from the Apex website or the website listed in the overview at the beginning of this appendix) and follow its installation instructions.

Installing Graphviz on Linux

Download Graphviz from the website listed in the overview at the beginning of this appendix and follow its installation instructions.

Installing Emacs on Linux

You probably already have Emacs. If not, download Emacs (e.g. from the Apex website or http://www.gnu.org/software/emacs/) and follow its installation instructions.

Starting Apex on Linux

Enter `apex &` to start Apex. Enter `sherpa &` to start Sherpa. *Apex must be started before Sherpa*. Refer to Chapter 2 for detailed instructions on using Apex.

You must quit Apex and Sherpa separately. To quit Sherpa, select Quit from the File menu, or close the window. To quit Apex, type `(exit)` at the Apex Listener prompt.

Note: It is not necessary to quit and restart Sherpa while using Apex. This is true even if Apex is quit (or dies) and is restarted. In these cases, simply reset Sherpa by selecting Reset from the File menu.

Appendix B: Using the Lisp Listener

You can use the Lisp Listener, in place of Sherpa, as the primary means for interacting with Apex[9]. Listeners are interactive text windows, which are inherent to Common Lisp systems and provide capabilities for debugging Lisp code. The Lisp Listener is always present when Apex is running. Normally, the Listener is found in the `*apex*` Emacs buffer (if you started Apex through Emacs), or in the Unix shell window from which you started Apex (or that opened when the Apex startup icon was double-clicked).

In the Listener, you can directly invoke the Lisp functions that control Apex. You can also access all features of Apex, except for the graphical features of Sherpa, by invoking a prompt-driven interface.

The Listener displays debugging information and other messages generated while Apex runs. Most of the messages contain internal information and can safely be ignored. However, when an error occurs, the Apex run is interrupted, and a debugging prompt appears in the Listener along with by an error message. This can occur frequently during development or modification of a model and is usually caused by Lisp programming errors.

Loading an application

Before you can run an Apex application, you must first load the Application Definition File into Apex. You can load an application by selecting from a list of recently loaded applications or by selecting a specific application.

To select from a list of recently loaded applications:

1 Invoke the Apex prompt (if necessary) by typing `(apex)`.

2 Type `load` or lower-case letter `l`. By default, Apex displays a numbered list of the last five applications loaded.

   Note that you can change the number of applications listed in the Listener using the expression `(change-load-history-size N)`, where `N` is a natural number. To affect the current session, type the expression in the Listener. Alternately, to make the setting persistent, add the expression to your preferences file. You can also clear the load history by typing `(clear-load-history)` in the Listener.

3 Enter the number of the application you want to load.

4 To load a specific application:

5 Invoke the Apex prompt (if necessary) by typing `(apex)`.

[9] **Warning**: Interacting with Apex through the Listener while Sherpa is running may lead to unexpected behavior. Use only one method of interaction in a given Apex session.

6 Type `load` or the lower-case letter `l`, then type the number of the last menu option. The Listener then prompts you for an application file.

7 Type in the full pathname of the application file as a string, for example, "`c:/apexapps/myworld.lisp`".

Running an application

Once you load an application, you can manipulate it in the following ways:

- Start the application by typing `(startapp)`. The application runs to completion or, in the case of simulations, until a scheduled pause point. Events print in the Listener as the simulation runs.

  Note that it is not possible to interactively pause an application in the Listener, though simulations can be programmed to pause automatically.

- Reset the application by typing `(resetapp)`. This restores the application to its initial state.

- If the application supports stepping, you can single-step the application by typing `(stepapp)`. Some applications have the ability to advance one step (for example, one time unit) at a time. Native Apex applications, on the other hand, use an event-driven simulation mechanism. Thus, for these applications, a step advances the simulation to the next scheduled simulation event(s) rather than by a fixed amount of simulated time.

Working with event traces

You can generate, filter, and save event traces from the Lisp Listener.

To generate an event trace, type `(generate-trace)` in the Listener. You can view the simulation trace in its entirety, but the trace may contain thousands of events or more. You can reduce the amount of trace information by specifying filtering criteria.

You can filter event traces in the following ways:

- Specify a *show level.* The show level specifies a collection of *event types* to display. In the Listener, you can set the show levels with the `show` function using the form `(show :level <level-name>)` where `<level-name>` is a symbol *without* quotes. Predefined show levels are described in Appendix C.

- Specify particular event-types of interest. In the Lisp Listener, you can select event types with the `show` function using the form `(show <event-type>)` where `<event-type>` is a symbol *without* quotes. Event types are listed in Appendix C.

- Specify events or parameters other than, and in addition to, event types using the `show` function. The show function is described in Appendix C.

You can save traces , such as those generated by applying a specific filter, by entering `(save-trace <filename>)` where `filename` is a string, either the full pathname or simply the filename. In the latter case, the trace is saved in the directory of the current application.

Using the prompt-based interface

The following commands are available from the prompt-based interface of the Lisp Listener. You can invoke the prompt-driven interface by entering `(apex)` in the Listener.

Command and shortcut		Description
`load`	`L`	Load an application
`start`	`S`	Start the application
`reset`	`R`	Reset the application
`step`	`Ss`	Advance the application one step
`reload`	`rl`	Reload the current application
`pause`	`p`	Pause options
`trace`	`t`	Trace options
`inspect`	`I`	Object inspection options
`format`	`f`	Toggle time display format
`display`	`d`	Toggle display of trace
`eval`	`e`	Evaluate a Lisp expression
`help`	`h/?`	Print the help message
`quit/exit`	`q`	Exit to Lisp prompt
`system`	`ex`	Exit Apex and Lisp to system prompt

Appendix C: Event Traces

Predefined show levels

all	All events
none	No events
default	`task-started` events
actions	Resource-related events
asa-low	Action Selection Architecture events, low detail
asa-medium	Action Selection Architecture events, medium detail
asa-high	Action Selection Architecture events, high detail
cogevents	Cognitive events
simulation	Activity-related events

Lisp commands for controlling trace output

(show)	Query the current trace constraint (see trace constraint syntax below)
(show :runtime)	Show the event trace as the simulation runs (useful for debugging)
(show :hms)	Display time in hours/mins/secs
(show :level *level*)	Set the show-level (see predefined show levels)
(show *EventType*)	Add the event type to the trace (see event types list)
(show Constraint)	Add events matching the given trace constraint to the trace
(unshow)	Turn off the event trace
(unshow :runtime)	Suppress runtime display of the event trace
(unshow :hms)	Display time as an integer
(unshow *EventType*)	Remove the event type from trace (see event types list)
(unshow Constraint)	Remove events matching the given constraint from the trace
(generate-trace)	Generate and print the trace
(trace-size)	Query the number of events in the latest trace
(define-show-level *name TraceConstraint*)	Define a new show level (*name* is symbol)

Trace constraint syntax

TraceConstraint:

TraceParameter	{ see TraceParameter below }
| (and TraceConstraint*)	{ match events meeting all given constraints }
| (or TraceConstraint*)	{ match events meeting any given constraint }
| (not TraceConstraint)	{ match events that fail the given constraint }

TraceParameter :

(event-type <symbol>)	{ match events of the given type }
| (object-id <symbol>)	{ match events containing the given object }
| (time-range (<low> <high>))	{ match events occurring in the given time range where <low> and <high> are duration expressions, see Appendix D }
| (agent-name <symbol|string>)	{ match events containing the given agent name }

| (agent-id <symbol>) { match events containing the given agent ID }
| (task-type <symbol>) { match events that reference tasks whose procedures have an index beginning with the given symbol }

Appendix D: PDL Syntax

Procedure level clauses

```
 (PROCEDURE
 [ (PROCTYPE :conc[urrent] | :seq[uential] | :ranked) ]
   (INDEX (<name:non-variable-symbol> . <parameters>) )
 [ (PROCTYPE :conc[urrent] | :seq[uential] | :ranked) ]
 [ (PROFILE <resource> <duration> <continuity>) ]*
 [ (LOG <log-policy>+) ]*
 [ (EXPECTED-DURATION <duration>) ]
 [ (TERMINATE (WHEN <monitor-condition>)) ]
 [ (INTERRUPT-COST <lisp-expression>:real) ]
 [ (STEP [<step-tag:symbol>] <task-description> [=> <var>] <step-clause>* )
]*
)
```

The following short forms of `PROCTYPE` and `INDEX` are also available:

- Instead of `(PROCTYPE <type>)` you can use just the type.
- Instead of `(INDEX (<name> . <parameters>))` you can just use `(<name> . <parameters>)`.

If the short forms of `PROCTYPE` and `INDEX` are used, they are position dependent:

- the `PROCTYPE` must occur immediately after `PROCEDURE`
- the name+parameter list must occur immediately after the `PROCTYPE` (if present) or otherwise after the `PROCEDURE`

The shortest possible procedure is `(procedure (noop))` which is a “no-op” procedure.

Step level clauses

```
(STEP [<step-tag:symbol>] <task-description> [=> <var>]
 [ (WAITFOR <monitor-condition> )
 [ (INTERRUPT-COST <lisp-expression>->real) ]
 [ (PRIORITY  <lisp-expression>->real) ]
 [ (RANK  <lisp-expression>->real) ]
 [ (REPEATING <repeat-spec>) ]
 [ (RESPONDING <respond-spec>) ]
 [ (SELECT <select-spec>) ]
```

```
 [ (FORALL <forall-spec>) ]
 [ (ON-START <lisp-expression>*) ]*
 [ (ON-END <lisp-expression>*) ]*
 [ (TERMINATE (when <monitor-condition>)) ]
 [ (RESTART (when <monitor-condition>)) ]
 [ (RESUME (when <monitor-condition>) ]
 [ (SUSPEND (when <monitor-condition>) (until <monitor-condition>)) ]
 )
```

`PERIOD` is also supported as a deprecated step-level clause.

Primitive Procedures

```
(PRIMITIVE
   (INDEX (<name:non-variable-symbol> . <parameters>) )
 [ (PROFILE <resource> <duration> <continuity>) ]
 [ (ON-START <lisp-expression>*) ]
 [ (DURATION <lisp-expression>*) ]
 [ (ON-END <lisp-expression>*) ]
 [ (ON-COMPLETION <lisp-expression>*) ]
 [ (LOCALS (<name> <lisp-expression>)* ) ]
 [ (UPDATE [<duration>] <lisp-expression>* ) ]
 [ (RETURN <lisp-expression>) ]
```

Primitives can have the same kind of short `INDEX` clause as procedures, for example instead of `(INDEX (<name> . <parameters>))` you can use `(<name> . <parameters>)`.

Monitor conditions

Measurement conditions

Long form:

```
(:measurement [<tag>:symbol]
   (<attr> <obj/v> <relop> <lisp-expression> [+/- <expr>] )
   [:timestamp <constraint>*]*
   [:value     <constraint>*]*
   [:object    <constraint>*]* )
```

Short form:

```
(<attr> <obj/v> <relop> <lisp-expression> [+/- <expr>] )
```

Estimated conditions

```
(:measurement [<tag>]
   (<attr> <obj/v> <relop> <lisp-expression> [+/- <expr>] )
      :estimation <estimator>
     [:timestamp <constraint>*]*
     [:value     <constraint>*]*
     [:object    <constraint>*]* )
```

estimator:

```
<persist-estimator> | <linear-regression-estimator>
```

persist-estimator:

```
(:persist [:with-timeout <duration>:+pos-infinity+])
```

linear-regression-estimator:

```
(:linear-regression
    [:minimum-points <int>:2]
    [:maximum-error <real>:infinity]
    [:maximum-difference <real>:infinity]
    [:minimum-frequency <real>:infinity]
    [:start <time-point>:+beginning-of-time+]
    [:end <time-point>:+current-time+] )
```

Simple episodic conditions

```
  (:episode [<tag>] (<attr> <obj/v>)
    :quality (:no-constaints) | (:minimum-sample-interval <msi-constraint> )
                              | (:msi <msi-constraint> )
    [:timing <timing-constraint>*]*
    [:value <constraint>*]*
    [:first-value <constraint>*]*
    [:last-value <constraint>*]*
    [:object <constraint>*]*
    [:stats <stat-constraint>*]*
    [:trend <trend-constraint>*]* )
```

msi-constraint:

```
<constraint> | <duration> ;; where <duration> is sugar for (<= <duration>)
```

timing-constraint:

```
(:start <constraint>*) |
 (:end   <constraint>*) |
 (:earliest-start <time-spec>) | (:es <time-spec>) |
```

```
 (:latest-start   <time-spec>) | (:ls <time-spec>) |
 (:earliest-end   <time-spec>) | (:ee <time-spec>) |
 (:latest-end     <time-spec>) | (:le <time-spec>) |
 (:duration <constraint>*)
```

stat-constraint:

```
(:mean <constraint>*) |
 (:count <constraint>*) |
 (:stddev <constraint>*) |
 (:variance <constraint>*) |
 (:sum <constraint>*) |
 (:difference <constraint>*)
```

trend-constraint:

```
 (:rate <constraint>*) |
 (:rate :increasing |
        :decreasing |
        :non-increasing |
        :non-decreasing) |
 (:step <constraint>*) |
 (:step :increasing |
        :decreasing |
        :non-decreasing |
        :non-decreasing )
 (:frequency <constraint>*)
 (:custom <interval-fn>)
```

Complex conditions

```
(:AND [<tag>] <condition>+ [:constraint <episode-constraint>*])
(:OR  [<tag>] <condition>+ [:constraint <episode-constraint>*])
(:NOT [<tag>] <condition>  [:constraint <episode-constraint>*])
(:IN-ORDER [<tag>] <condition>+ [:constraint <episode-constraint>*])
(<allen> [<tag>] <condition_1> <condition_2> [:constraint <episode-
constraint>*])
```

where `<allen>` is one of the following: `:CONTAINS | :FINISHES | :STARTS | :BEFORE | :MEETS | :OVERLAPS | :COTEMPORAL | :DURING | :FINISHED | :STARTED | :AFTER | :MET | :OVERLAPPED`

Time-based conditions

```
(:DELAY [<tag>] <duration_1> [<duration_2>])
(:TIMESTAMP [<tag>] <timepoint-expr_1> [<timepoint-expr_2>])
```

Delay monitors can stand alone, or be part of `*internal* :IN-ORDER` conditions.

```
(:IN-ORDER <c_1> ... <delay-condition> ... <c_n>).
```

Atomic episode conditions

Any other form is treated as an atomic episode, which also has the following long form:

```
(:atomic-episode <tag> <form> [:timing <timing-constraint>*]*)
```

Appendix E: Diagram Programming Reference

Labeling graphical objects

Currently, an Apex object can have only one graphical object associated with it. To allow adding text labels, a graphical object can have a list of text labels, which are displayed when a diagram is viewed in Sherpa.

The following methods can be used to label a graphical object.

You can add a label at specified coordinates. You can use this method repeatedly to add several labels.

```
add-label ((gobj graphical-object) label x y
           &key (font-size 8) (font-family "Arial") (font-weight 'normal)
(font-style 'normal)
           (text-anchor "start") (fill "black"))
```

You can add a single label to object, automatically centered in the middle. Currently, this method is only defined for circle and rect graphical objects.

```
auto-label ((gobj graphical-object) label
           &key (font-size 8) (font-family "Arial") (font-weight 'normal)
(font-style 'normal)
           (text-anchor "middle")  (fill "black"))
```

Example

This example illustrates adding multiple labels to a graphical object.

```
(defun create-labeled-rect (locale)
  (let ((keypad (make-instance 'interface-object :name "keypad"
                               :locale locale)))
    (assemble keypad)
    (setf (graphical-object keypad)
     (make-instance 'rect
                  :x 0 :y 0 :width 100 :height 100
                  :fill "grey" :fill-opacity 1 :stroke "black"))
    ;;add 2 labels to the rectangle
    (add-label (graphical-object keypad) "Line 1" 10 20 :fill "white"
                  :font-weight "bold" :font-size 12)
    (add-label (graphical-object keypad) "Line 2" 10 40 :fill "white"
                  :font-weight "bold" :font-size 12)))
```

Wireframe diagrams

In applications where Apex objects are subclasses of visob or interface-object, it is easy to build a wireframe diagram to visualize your objects.

You can use the following syntax to create a wireframe object based on faces slot of visob.

```
create-wireframe-graphical-object ((obj visob) &key (auto-update nil)
(view-type 'xy))
```

If `auto-update` is non-nil, the diagram in Sherpa will be updated using current values of the faces slot. The `view-type` specifies which view of the three-dimensional object will be displayed: one of ('xy, 'xz, 'yz).

You can create a wireframe rectangle using the pos slot for xy, and dimensions slot for width and height.

```
create-wireframe-graphical-object ((obj interface-object) &key (auto-update
nil))
```

If `auto-update` is non-nil, the diagram in Sherpa will be updated using the current values of the pos and dimensions slots.

SVG diagrams

When building an SVG diagram with a tool, there are a few things to remember if you want to be able to click on objects when viewing the diagram in Sherpa.

- the SVG object ID must be either the name of an appob or its ID.
- IDs cannot start with numbers (part of the SVG standard). For example, to refer the appob corresponding to key 9 in a keypad, use name="key9", not name="9key".
- the SVG object ID must match the appob name exactly. The name is case sensitive, except when the Apex object name is a symbol.

Appendix F: Time and Date Formats

The maximum resolution of the Apex clock is milliseconds. Durations are an amount of time; thus, a number of milliseconds. For example, the expression `(1 min 3 sec)` resolves to the number 63000. The Lisp function `duration-read` can be used to convert duration expressions into their numeric equivalent.

A *timepoint* is a point in time; in Apex, this is the result of evaluating an expression with reference to some starting point. For example, 1 minute and 3 seconds from "now" can be written, in monitor expressions, as `(+ +now+ (1 min 3 secs))`. The functions `start-of` and `end-of` can be used to describe a starting point; so for example, `(start-of +this-task+)` is the timepoint representing the start (creation time) of the current task.

An *interval* is a pair of timepoints, representing the beginning and end of an interval of time. Its duration is the end-beginning. It is, of course, possible to have the end and the beginning be the same: this is an *instant*. Apex does not provide any special syntax for intervals; typically, the start and end of an interval are specified independently. The default interval for a monitor for a condition is over the interval that begins at `(start-of +this-task+)` and ends "at the end of time." The following standard constants are currently defined as:

```
(defconstant +end-of-time+ most-positive-single-float)

(defconstant +beginning-of-time+ 0)

(defconstant +always-dur+ (make-interval +beginning-of-time+ +end-of-time+))
```

There are two syntaxes for writing durations. One is a list-based form, which is somewhat more readable, and allows pattern matching. The other is a string/symbolic-based form, which is somewhat more compact. Also, as a special case, the numeral **0** can be used for a duration of 0 milliseconds.

List based form: `( <spec>+ )`, where spec is:

`(<number> <unit>)`, where `<number>` is any number, and `<unit>` is one of:

- `ms msecs msecs` : milliseconds
- `s sec secs second seconds`: seconds
- `min mins minute minutes`: minutes
- `h hr hrs hour hours`: hours
- `d day days`: days
- `w week weeks`: weeks

For example, `(3 wks 2 days 1 hr 3 min 4 seconds 500 ms)`

Symbol-based form: `P[wW][dD][hH][mM][sS]`, where:

- `w`—number of weeks (7 days)
- `d`—number of days (24 hours)
- `h`—number of hours
- `m`—number of minutes
- `s`—number of seconds

Thus, `P3w2d1h3m4.5s` is the same as `(3 wks 2 days 1 hr 3 min 4 seconds 500 ms)`.

Appendix G: Troubleshooting Tips

This section provides solutions to common problems.

Problem: A task that should start never does. It seems to wait forever.

Explanations/Solutions:

1. There may be a mismatch between the forms (patterns) of the event and the `waitfor` precondition.
   a. One of the patterns may contain a spelling error.
   b. There may be a difference in the order of pattern elements. For example, a perceptual event of the form `(between a b c)` will not match a precondition of the form `(between a c b)`, even though both indicate that `a` is observed to be between `b` and `c`.
   c. There may be a difference in the type of pattern elements. For example, `(distance a b 2)` will not match `(distance a b 2.0)`.
   d. The number of parameters in the events and precondition may be different.
2. The event occurs before the existence of the task whose precondition it should match. This can happen when events and preconditions are both created at the same "instant" according to the simulation clock.
3. The event occurs after the task whose precondition it should match is (prematurely) terminated.
4. The event occurs before the task whose preconditions it should match is created.

Problem: A task starts prematurely, before its `waitfor` preconditions should be satisfied.

Explanations/Solutions:

1. A precondition is less constrained than it seems to be, allowing it to match events that it should not. For example, consider a procedure consists of steps `s1` (no preconditions), `s2` (waits for `s1`; binds `?x` when it terminates) and `s3` (waits for `(color ?x red)`). The intention may be to determine an object of interest in step `s2` and then wait for it to turn red, but in this case `s3` will be enabled by observing ANY red object.
2. An event matching the precondition is generated from an unexpected source
3. There are disjoint enablement conditions (multiple `waitfor` clauses), allowing the task to become enabled for execution in an unexpected wa

Appendix H: Pattern Matching

A variety of PDL constructs, including `index`, `waitfor`, and `step`, use pattern matching. The pattern matcher is based on Peter Norvig's pattern matcher in his Source: *Paradigms of AI Programming* (1991).

A *variable* is a symbolic name for a value. There are three types of variables in PDL: *free variables*, *bound variables*, and *indexical variables*.

- Free variables allow free matching to any literal and are prefixed with a question mark.
- A bound variable must have a value by the time it is first used. Bound variables look like free variables surrounded by angle brackets, for example, `<?altitude>` and `<?rate>`. Consider the following snippet:

```
(step (increase throttle speed to <?rate> meters per second)
  (waitfor (rate request ?rate))
```

The condition `(rate request ?rate)` enables this step; when an event occurs that matches `(rate request ?rate)`, the `?rate` variable becomes bound. Since this condition enables the step, it is guaranteed that `?rate` will be bound before the step's action is taken. By annotating the variable in the action description, we provide some protection against typographical errors or dangling variable references. Apex issues a warning if a there is an attempt to bind to a bound variable that does not have a value.

- Indexical variables are, in fact, function calls. Thus the value of an indexical variable is the value of the function. Indexical variables are surrounded by plus signs, for example, `+self+`, `+this-task+`, and `+now+`. Table 1 lists the standard indexical variables.

Name	Meaning
+self+	The agent of the current task
+this-task+	The current task
+now+	The current time
+tot+	Time on the current task, in milliseconds
+met+	Time elapsed since the beginning of the application (Mission Elapsed Time), in milliseconds

Table 1 Standard indexical variables

To define a new indexical variable is easy enough: simply write a Lisp function that takes one argument, the currently executing task. Here, for example, are the definitions of `+now+` and `+tot+`:

```
(defun now (task) (declare (ignore task) (current-time))

(defun tot (task) (if task (- (current-time) (t-started task)) 0))
```

A *variable name* is the name of the variable; that is, the symbolic part of a variable. A *value* is any Lisp value, such as a symbol, a number, list or string.

A *binding* is a variable name/value pair, such as `(n . 34)`. A *binding set* is a list of bindings, such as `((n . 34)(aircraft . ac1))`. The special bindings set `no-bindings` is a binding set that contains no bindings; it is distinguished from the Lisp `nil` form; that is, `(not (null no-bindings))` is `T`.

The Lisp function `pat-match` matches a pattern to a target, and returns either a binding set (which might be the `no-bindings` binding set) if the match succeeds, or `fail` if the match fails. It takes an optional binding set as a parameter; if provided, all variables in the pattern are substituted with their values before matching; if the match succeeds, the binding set is extended with any new bindings found.

The following rules govern pattern matching:

- A free variable matches to any Lisp form, and results in a new binding of the variable and the Lisp form.
- If the pattern and the target are both EQL, they match, without any new bindings.
- If the pattern and the target are both strings, and are STRING-EQUAL, they match, without any new bindings,
- If the pattern and the target are both lists, they match if their CDRs and CARs match, respectively, producing any new bindings as a result.
- If the pattern is a “segment matcher,” its segment matching function is to perform the match.
- If the pattern is a “single pattern matcher,” its single pattern matching function is called to perform the match.
- Otherwise, the pattern and target do not match, and `fail` is returned.

The following segment matchers are provided:

- ?* : matches zero or more occurrences in the target
- ?+: matches one or more occurrences in the target
- ??: matches zero or one occurrence in the target.
- ?if: matches if Lisp code succeeds.

The following single pattern matchers are provided:

- ?is : if function defined succeeds, the variable is bound to the target,
- ?and: succeeds if all the patterns match the input,
- ?or: succeeds if one of the patterns match the input,
- ?not: succeeds if the match fails.

The following examples illustrate the behavior and capabilities of the pattern-matching algorithm.

```
> (pat-match '?any t)
;;; -> ((any . t))
> (pat-match '?any 34)
;;; -> ((any . 34))
> (pat-match '?any '(this is a list))
;;; -> ((any this is a list))
> (pat-match 10 10)
;;; -> no-bindings
> (pat-match 'a 'a)
;;; -> no-bindings
> (pat-match '(name ?first ?last) '(name pat doe))
;;; -> ((last . doe) (first . pat))
> (pat-match '(name ?first ?last) '(name "Pat" "Doe"))
;;; -> ((last . "Doe") (first . "Pat"))
> (pat-match '(name ?first ?last) '(name "Pat" "Doe") '((last . "Doe")
(first . "Pat")))
;;; -> ((last . "Doe") (first . "Pat"))
> (pat-match '(hit ?object with ?tool) '(hit nail with hammer))
;;; -> ((tool . hammer) (object . nail))
> (pat-match '(x = (?is ?n numberp)) '(x = 34))
;;; -> ((n . 34))
> (pat-match '(?x (?or < = >) ?y) '(3 < 4))
;;; -> ((y . 4) (x . 3))
> (pat-match '(x = (?and (?is ?n numberp) (?is ?n oddp))) '(x = 3))
;;; -> ((n . 3))
> (pat-match '(?x /= (?not ?x)) '(3 /= 4))
;;; -> ((x . 3))
> (pat-match '(?x > ?y (?if (> ?x ?y))) '(4 > 3))
;;; -> ((y . 3) (x . 4))
> (pat-match '(a (?* ?x) d) '(a b c d))
;;; -> ((x b c))
> (pat-match '(a (?* ?x) (?* ?y) ?x ?y) '(a b c d (b c) (d)))
;;; -> ((y d) (x b c))
> (pat-match '(?x ?op ?y is ?z (?if (eql (?op ?x ?y) ?z))) '(3 + 4 is 7))
;;; -> ((z . 7) (y . 4) (op . +) (x . 3))
> (pat-match '(?x ?op ?y (?if (?op ?x ?y))) '(3 > 4))
;;; -> fail
```

Appendix I: Application Definition File Example

The following is an example of a well-formed Application Definition File named `hello.lisp`. This file is available, along with other sample applications, in the examples directory of the Apex distribution.

```
(in-package :user)

;;; This is a very simple and introductory Apex application that
;;; simulates answering of a ringing telephone.

(defapplication "Hello World"
    :init-sim (hello-world))

;;; Jill is an agent, who knows how to answer the phone when it rings.
;;; These are her procedures.

(in-apex-bundle :jill)

(procedure :sequential (handle phone)
  (step (answer ?phone the phone)
        (waitfor (ringing ?phone))
        (on-start
         (inform '(answering the ?phone)))
        (on-end
         (inform '(answered the ?phone))))
  (step (end-application)))

(procedure :sequential (answer ?phone the phone)
  (profile right-hand)
  (step (pickup-phone ?phone))
  (step (say-hello)
        (on-start
         (inform '(saying hello into the ?phone)))
        (on-end
         (inform '(said hello into the ?phone)))))

(procedure :sequential (pickup-phone ?phone)
  (profile right-hand)
  (step (pick up ?phone with right-hand)))

(procedure :sequential (say-hello)
  (profile voice)
  (step (speak "Hello?")))

(primitive (pick up ?phone with ?hand)
 (profile ?hand)
 (duration p2s))

(primitive (speak ?utterance)
 (profile voice)
 (duration (500 ms))
 (on-completion
  (inform `(said ,?utterance) :author +self+)))
```

```
;;; The phone is an agent, which knows how ring after some time.

(in-apex-bundle :phone)

(primitive (ring sometime)
  (on-start
   (schedule 'p1.5s
     (inform `(ringing ,+self+) :author +self+))))


;;; Initialization function for this application.

(defun hello-world ()
  (let* ((locale (make-instance 'locale :name "Room"))
         (jill (make-instance 'agent :name "Jill"
                              :use-bundles '(:jill)
                              :initial-task '(handle phone)
                              :locale locale))
         (phone (make-instance 'agent :name "Phone"
                               :use-bundles '(:phone)
                               :initial-task '(ring sometime)
                               :locale locale)))

    ;; Trace settings
    (show state)
    (show saying)
    (show said)
    (show ringing)
    (show answering)
    (show answered)))
```

Appendix J: Starting Apex within Allegro Common Lisp

If you have the Franz Allegro Common Lisp (ACL) development environment, you can run Apex within ACL instead of from the Apex distribution. If you have the Emacs ACL interface, you can use Emacs to start ACL.

To run Apex within ACL:

11. If you have the ACL Emacs interface, start Emacs.

12. Start ACL.

    a. Within Emacs, type `M-x fi:common-lisp` to start ACL. Note that you can switch to a shell command line in Emacs by typing `M-x shell`.

    ```
    -1:**-Emacs  *shell*
     M-x shell
    ```

    From the command line, change to the directory containing ACL and type `alisp`.

    b. If you are not using Emacs, start the `alisp` executable of ACL from a shell command line. When ACL successfully starts, the `CL-USER(1):` Lisp prompt appears.

    ```
    [Cheetah:~] rjew% cd /Applications/acl62
    [cheetah:/Applications/acl62] rjew% alisp
    International Allegro CL Enterprise Edition
    6.2 [Mac OS X] (Jun 26, 2002 11:19)
    Copyright (C) 1985-2002, Franz Inc., Berkeley, CA, USA.  All Rights Reserved.

    This development copy of Allegro CL is licensed to:
       [TC9389] NASA Ames Research Center

    ;; Optimization settings: safety 1, space 1, speed 1, debug 2.
    ;; For a complete description of all compiler switches given the
    ;; current optimization settings evaluate (EXPLAIN-COMPILER-SETTINGS).
    CL-USER(1): :cd /
    ```

13. Load Apex.

    a. Change to the directory that contains Apex. Note that all ACL commands are preceded with a '**:**'. For example, to change to the root directory, type `:cd /`.

b. Type `(load "load")` at the Lisp prompt. Note that all Lisp commands are enclosed in **( )**'s. A status of components being loaded scrolls across the screen. When Apex successfully loads, the `Welcome to Apex!` message appears.

```
CL-USER(1): :cd /
/
CL-USER(2): :cd apex248
/apex248/
CL-USER(3): (load "load")
; Loading /apex248/load.lisp
;   Fast loading from bundle code/tester.fasl.
;   Loading apex:system;utility;debug.lisp
;      (/apex248/system/utility/debug.lisp)
;   Fast loading apex:system;loading;defsystem.fasl
;      (/apex248/system/loading/defsystem.fasl)
;   Loading apex:system;loading;intrinsic.system
;      (/apex248/system/loading/intrinsic.system)
;   Loading apex:system;loading;sim-engine.system
;      (/apex248/system/loading/sim-engine.system)
```

14. Start the Sherpa server. Enter `(sherpa)` at the Lisp prompt. Sherpa starts as a separate process so that you can still use the Apex Listener. When Sherpa successfully starts, the message `CL-USER(): started sherpa server ...` appears.

```
;   Fast loading /apex248/system/sherpa/acl.fasl
;   Fast loading /apex248/system/sherpa/pert/pert.fasl
;   Fast loading /apex248/system/sherpa/pert/cp.fasl

Welcome to Apex!

Version 2.4 Beta, Copyright (C) 2003 NASA Ames Research Center
Apex is built on Allegro Common Lisp (R), by Franz, Inc.
T
CL-USER(4): (sherpa)
#<MULTIPROCESSING:PROCESS Sherpa Server @ #x304dfeb2>
CL-USER(5): started sherpa server...
; Loading apex:apexlib;default-apexlib.lisp (/apex248/apexlib/default-
sp)
; Loading /ahello.lisp
```

15. Launch Sherpa.

- On Windows or Macintosh, double-click the Sherpa icon.
- On Linux or Solaris, change to the directory which contains the Sherpa jar file then type `java -jar <sherpa.jar>`, where `<sherpa.jar>` is the exact name of the Sherpa jar file.