

# Apex Documentation | 3\_0\_0ReferenceManual

---

## Apex 3.0 Reference Manual

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

[apexhelp@eos.arc.nasa.gov](mailto:apexhelp@eos.arc.nasa.gov)  
<http://ti.arc.nasa.gov/projects/apex/>

© 2006 NASA Ames Research Center  
All trademarks are the property of their respective owners.

### Table of contents

1. [Introduction](#)
  - a. [Obtaining more information](#)
  - b. [Getting the latest information](#)
  - c. [Using this manual](#)
2. [Using Apex](#)
  - a. [Introduction to the Sherpa workspace](#)
    - i. [The object tree](#)
    - ii. [The view panel](#)
  - b. [Starting Apex](#)
  - c. [Loading an application](#)
  - d. [Running an application](#)
  - e. [Working with event traces](#)
    - i. [Filtering an event trace](#)
    - ii. [Other event trace features](#)
  - f. [Inspecting objects](#)
  - g. [Viewing agendas](#)
  - h. [Viewing diagrams](#)
  - i. [Viewing monitors](#)
  - j. [Viewing state variables](#)
    - i. [Exporting state variable information](#)
  - k. [Using the PDL view](#)
    - l. [Using primary views](#)
  - m. [Moving back and forth through the view history](#)
  - n. [Setting general preferences](#)
    - i. [Setting the look and feel](#)
    - ii. [Setting the font](#)
  - o. [Searching](#)
  - p. [Sherpa troubleshooting](#)
  - q. [System patches](#)
  - r. [Getting help](#)
3. [Creating Apex Applications](#)
  - a. [Lisp programming and Emacs](#)
  - b. [Application Definition File](#)
  - c. [Application files](#)
  - d. [Libraries](#)
    - i. [Using libraries](#)
    - ii. [Creating libraries](#)
    - iii. [Finding libraries](#)
    - iv. [Provided libraries](#)
4. [Procedure Description Language](#)
  - a. [Introduction](#)
    - i. [Action Selection Architecture](#)
  - b. [Basics](#)

- i. [Introduction to PDL variables](#)
      - ii. [Introduction to PDL time representation](#)
      - iii. [Notations and conventions](#)
    - c. [Procedures](#)
      - i. [index](#)
      - ii. [proctype](#)
      - iii. [profile](#)
      - iv. [log](#)
      - v. [terminate/when](#)
      - vi. [interrupt-cost](#)
      - vii. [on-start](#)
      - viii. [on-end](#)
    - d. [Steps](#)
    - e. [Monitoring overview](#)
      - i. [waitfor](#)
      - ii. [terminate/when](#)
      - iii. [suspend/when/until](#)
      - iv. [resume/when](#)
      - v. [select](#)
    - f. [Task control overview](#)
      - i. [Priority](#)
      - ii. [Rank](#)
      - iii. [interrupt-cost](#)
    - g. [Repetition, policies, and iteration](#)
      - i. [repeating](#)
      - ii. [forall](#)
      - iii. [period](#)
    - h. [Conditions](#)
      - i. [Monitoring state variables](#)
      - ii. [Times associated with monitors](#)
      - iii. [:measurement](#)
      - iv. [:estimation](#)
      - v. [:episode](#)
      - vi. [:and](#)
      - vii. [:or](#)
      - viii. [:not](#)
      - ix. [:in-order](#)
      - x. [Allen monitors](#)
      - xi. [:delay](#)
      - xii. [:timestamp](#)
    - i. [Primitive procedures](#)
      - i. [primitive](#)
      - ii. [duration](#)
      - iii. [update](#)
      - iv. [on-start](#)
      - v. [on-completion](#)
      - vi. [locals](#)
      - vii. [profile](#)
      - viii. [return](#)
    - j. [Miscellaneous features](#)
      - i. [Agent's initial task](#)
      - ii. [PDL Partitions \(Bundles\)](#)
- 5. [Apex Programming Reference](#)
  - a. [defapplication](#)
  - b. [Application interface](#)
  - c. [Time and Scheduled Events](#)
  - d. [Agents](#)
  - e. [Inform](#)
  - f. [Diagrams](#)
    - i. [Hierarchical diagrams](#)

- ii. [Wireframe diagrams](#)
  - iii. [File-based SVG diagrams](#)
6. [Glossary](#)
7. [Appendix A: Getting Started with Apex](#)
  - a. [Overview of third party software packages](#)
    - i. [Java Runtime Environment ® \(JRE\)](#)
    - ii. [Graphviz®](#)
    - iii. [Emacs](#)
  - b. [Getting started on Macintosh](#)
    - i. [Installing the software on Macintosh](#)
    - ii. [Starting Apex on Macintosh](#)
  - c. [Getting started on Windows](#)
    - i. [Installing the software on Windows](#)
    - ii. [Starting Apex on Windows](#)
  - d. [Getting started on Linux](#)
    - i. [Installing the software on Linux](#)
    - ii. [Starting Apex on Linux](#)
8. [Appendix B: Using the Lisp Listener](#)
  - a. [Loading an application](#)
  - b. [Running an application](#)
  - c. [Working with event traces](#)
  - d. [Using the prompt-based interface](#)
9. [Appendix C: Event Traces](#)
  - a. [Predefined show levels](#)
  - b. [Lisp commands for controlling trace output](#)
  - c. [Trace constraint syntax](#)
10. [Appendix D: PDL Syntax](#)
  - a. [Procedure level clauses](#)
  - b. [Step level clauses](#)
  - c. [Primitive Procedures](#)
  - d. [Monitor conditions](#)
    - i. [Measurement conditions](#)
    - ii. [Estimated conditions](#)
    - iii. [Simple episodic conditions](#)
    - iv. [Complex conditions](#)
    - v. [Time-based conditions](#)
    - vi. [Atomic episode conditions](#)
11. [Appendix E: Diagram Programming Reference](#)
  - a. [Labeling graphical objects](#)
  - b. [Example](#)
  - c. [Wireframe diagrams](#)
  - d. [SVG diagrams](#)
12. [Appendix F: Time and Date Formats](#)
13. [Appendix G: Troubleshooting Tips](#)
14. [Appendix H: Pattern Matching](#)
15. [Appendix I: Application Definition File Example](#)
16. [Appendix J: Starting Apex within Allegro Common Lisp](#)

## 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.

## Obtaining more information

More information about Apex is available on the Apex website, <http://ti.arc.nasa.gov/projects/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/ctl/ctl2.html>

## Getting the latest information

For the current version of Apex and this document, visit <http://ti.arc.nasa.gov/projects/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](mailto:apexhelp@eos.arc.nasa.gov).

For information related to the development of the Apex system, send email to [mfreed@arc.nasa.gov](mailto:mfreed@arc.nasa.gov).

## 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.
- [Using Apex](#) covers the basics of using Apex via its graphical user interface Sherpa.
- [Creating Apex Applications](#) and [Procedure Description Language](#) describe how to build an application with [Procedure Description Language](#) 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.
- [Apex Programming Reference](#) describes Apex software in more detail, as needed to modify and extend system behavior or cause

Apex to interoperate with other systems.

## Using Apex

This section 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](#).

## 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.

### 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.

### 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 section.

## 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.

## 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.

## 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.

## 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.

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.

### 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 filter 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 redisplay 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

## 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.

## 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.



## 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.

## 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 [Apex Programming Reference](#).

## 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.

## 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.

## 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

## 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.

## 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.

## 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.

## 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.

## 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.

## 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

## 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.

## 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.

## 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.

## Getting help

If you experience problems with Apex, please consult the [Troubleshooting section](#) 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](mailto: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

## 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 [Apex Programming Reference](#) 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.

## 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.

## 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 [Apex Programming Reference](#)) 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))
```

## 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, 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.

## 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.

### 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")
```

### 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`.

### 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.

## 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.

# Procedure Description Language

## 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, are stored on a structure called the agenda internal to the Action Selection Architecture. When a task on the 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).

## Action Selection Architecture

The Action Selection Architecture 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
- 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. 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.

## Basics

### 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 indicate 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.

### 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))))))
```

## 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

## Procedures

**Type:** first-class construct

**Syntax:**

```
(PROCEDURE
  [ (PROCTYPE :conc[urrent] | :seq[quential] | :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)))
```

## 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).

## proctype

**Type:** procedure-level clause

**Syntax:** (PROCTYPE :conc[urrent] | :seq[uentia]l | :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) ...)
```

## profile

**Type:** procedure-level clause

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

**resource-requirement:** resource:<symbol>

The profile clause lists discrete resources required for using a procedure. 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 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.

## 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|))))
```

## 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.

## 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))
```

## 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.

## on-start

**Type:** procedure-level clause

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

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

## on-end

**Type:** procedure-level clause

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

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

## 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>)) ]
)
```

## 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).

### 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.

### 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.

### 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.

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.

## 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.

## 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.

## Task control overview

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

### 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 do not inherit priority from the suspended ancestor.

## 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 from 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.

## 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.

## Repetition, policies, and iteration

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



## 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)
  (repeating :with-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.

responding

**Type:** step-level-clause

**Syntax:** (responding [:while <test>] [:until <test>][:for-new <var1> ..<varn>])

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.

```
(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.

## forall

**Type:** step-level-clause

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

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.

## period

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

## 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))
```

## 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)/<raw>. 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): <pre width=100><!--verbatim47--></pre> where <raw><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 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>))
```

## 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

## :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.

## 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
```

## 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))
```

### 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)))
```

### :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>/<raw>. A <raw><persist-estimator> uses the form:

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

and a 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:

### 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)))
```

### 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))))
```

### :episode

**Type:** condition clause

#### Syntax:

The full syntax for episode monitors is as follows:

```
(:episode [<tag>] (<attr> <obj/v>)
  :quality (:no-constants) | (: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> )





























































