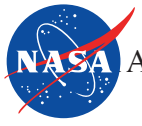
Reference Manual

Michael Freed
Eric Dahlman
Michael Dalal
Robert Harris



Ames Research Center

Printed in the United States of America.

Layout Design: Reagan Jew

Edition: May 11, 2004

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

apexhelp@eos.arc.nasa.gov

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

Contents

1.0 Introduction

1.1 What is Apex?

Apex is a computer application for generating adaptive, intelligent behavior in complex environments. It is the principal element of the Apex System that includes a range of components for modeling, simulating and analyzing human behavior. Intended uses include:

- Helping engineers evaluate and design human-machine systems
- Anticipating how newly introduced technologies will affect human operators
- Standing in for human participants in a training simulation
- Exploring or illustrating scientific theories of human performance

The Apex approach to human modeling separates aspects of behavior and performance that apply to intelligent agents in general from aspects that are particular to humans. The Action Selection Architecture (ASA) integrates AI techniques such as hierarchical planning and online-scheduling seen as useful for creating agents with human-level ability. By building capabilities into the architecture and providing a high-level language for behavior representation, Apex makes it easier to create human agent models for complex task environments. Findings from cognitive psychology and other areas concerned with human performance are incorporated into the Human Resource Architecture (HRA), which parameterizes and constrains the general agent model. A human model in Apex combines the ASA and HRA with a set of behavior representations, some specific to the task at hand, others general across many tasks.

Apex is meant to be a practical tool. It has proven successful in automating a Human-Computer Interaction analysis method called GOMS, including an especially powerful but complex variant called CPM-GOMS. The approach has also been useful for rapidly developing simulations of normative human behavior and for reconstructing incidents involving human error.

As a practical tool, one crucial consideration is to minimize the time and expertise required to build new models. This goal influences every aspect of Apex. For example, in production-system based cognitive architectures, behaviors are represented at an

"atomic" level at which the mechanisms of cognitive processing can be described in detail. In Apex, behavior is represented at a high-level, allowing modelers to ignore how behavior is generated and focus on what behaviors are desired. This can be viewed as trading usefulness at representing scientific theories of cognition for usefulness at representing complex, large-scale tasks. Similarly, Apex incorporates approaches to many high-level aspects of cognition such as selecting action under uncertainty, managing concurrent tasks, and task interleaving. These capabilities are relatively easy to invoke though a modeler is provided little flexibility in representing how they are realized.

In developing Apex, it has become clear that constructing a practical, broadly applicable human-system modeling tool is too great a job for any small team of individuals. Given the great number of issues to be addressed and the many different kinds of expertise needed, such an endeavor is most naturally carried out through a distributed development process. The design of the Apex system lends itself to distributed development. While the Action Selection Architecture is complex and its subcomponents tightly coupled, the other elements of the system are modular and thus relatively easy to extend, modify or replace. For example, cognitive, perceptual and motor faculties represented in the resource architecture are completely independent of the core action-selection mechanism, allowing modelers to "plug-in" alternative sub-models. Similarly, Apex includes a set of reusable "building blocks" for new models that can easily be modified or added to. This document is intended mainly to support the use of Apex in its current form but also provides important information for developing new Apex elements.

1.2 System Components

Software components of the Apex system fall into four categories or component layers including: the intelligent agent layer, the human/environment layer, the infrastructure layer and the user layer. The intelligent agent layer provides the ability to specify simulation entities with complex behavior reflecting goals, new events and "how to" knowledge. Its primary use in Apex is to model human operators, although it is also useful for modeling other simulation entities such as robots and aircraft autopilots. The intelligent agent layer currently includes a single component: the Action Selection Architecture (ASA), an import from the field of artificial intelligence originally designed to control mobile robots acting in complex, real-world environments. The capabilities it provides facilitate simulation of relatively sophisticated aspects of human behavior such as adapting to time-pressure, coping with uncertainty, and interleaving multiple tasks.

The human/environment layer includes a wide range of components for specifying and making inferences about humans and other entities that populate a simulation. Important subsets of these components are human resources models – representations of human cognitive, perceptual and motor faculties such as hands and eyes – which together comprise the Human Resource Architecture (HRA). Each resource model specifies performance-limiting characteristics. For example, the vision model specifies a restricted field of view, variable acuity, and a time lag between sensing and interpreting visual information. The agent and resource architectures combine to model a human agent. While the Action Selection Architecture provides the ability to engage in com-

plex behavior, the resource architecture causes this behavior to conform to human limits.

Also included in the human/environment layer are means for representing and reasoning about physical spaces (locales) and the spatial (e.g. containment, attachment, adjacency) and visual properties (e.g. color, orientation) of objects in a locale. Other components in this layer are building blocks for constructing models in human-computer interaction domains. These include representations of interface widgets (e.g. buttons, mice, keyboards) and of behaviors for using those widgets. The common theme for the components of this layer is that they are ingredients for building models of human-machine systems. Though intended to be reusable, they should not be considered core components of the Apex system. Users are encouraged to extend or replace these elements as they see fit.

The infrastructure layer provides essential services including simulation, trace event logging and mechanisms for interoperating with non-Apex processes such as an external simulation of a physical environment. The simulation component is composed of three parts: a simple language for defining "objects" to be simulated, a simulation engine whose job it is to carry out the actual simulation process, and a Lisp interface for controlling the simulation process. Some extensions to the Apex system, including development of new human resource models, require familiarity with simulation mechanisms and other components of the infrastructure layer. However, most users will probably need to know little more than how to operate the simulation engine – e.g. to begin or pause a simulation trial.

The user interface layer provides components to facilitate model construction, model debugging, and analysis and visualization of simulation output. The central element of this layer is Sherpa, a GUI that provides a range of services including buttons (shortcuts) for controlling the simulation engine, tools for handling large volumes of trace output, tools for examining simulation entities during and after a run, and a facility for automatically generating graphical representations of agent behavior.

To apply Apex in a particular domain, a user creates a simworld – a representation of a particular task and task environment. For example, to simulate people using a new automatic banking machine, an Apex user would represent the new machine's appearance and behavior, the procedural knowledge needed to operate it, and a scenario providing specifications for a particular simulation run. Together, the Apex system and user-defined simworld elements constitute an Apex application. To develop new applications, a user should be comfortable programming in Lisp and should become familiar with the contents of this manual.

1.3 Getting More Information

This document focuses on the practical aspects of using Apex. For the current version of Apex and this document, visit http://human-factors.arc.nasa.gov/apex. More information is available from several sources. Published papers describe many aspects of Apex including:

- using Apex for CPM-GOMS (John, et al. 2002)

- GOMS analyses (Freed and Remington, 2000a)
- human error prediction (Freed and Remington, 1998)
- human-system modeling methodology (Freed and Remington 2000b; Freed, Shafto and Remington 1998; Freed and Shafto 1997)
- multitask management (Freed 2000; Freed 1998a)
- detailed description of the Apex Action Selection Architecture and the modeling approach it supports (Freed 1998b)

To report a bug or consult on a technical problem, contact the Apex development team, apexhelp@eos.arc.nasa.gov. For information related to the development of the Apex system send an email to Michael Freed, mfreed@arc.nasa.gov.

Extending and developing applications in Apex may require programming in Common Lisp. The complete text of Common Lisp by Guy Steele is at: http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/ai/html/cltl/cltl2.html.

1.4 Conventions

In order to make this manual easier to read, the following typography conventions have been adopted. When code is shown, it appears in `11 pt Courier`.
For example,

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

When a section of code is of particular importance, it is in **`bold 11 pt Courier`**.
The `simob` class is the main focus of this example:

```
(defclass book (simob); BOOK is a subclass of SIMOB
```

User input is shown in chevrons (i.e. < >) and sometimes has a qualifying statement following it. For example,

`(apex-info :version <version>)`
where `<version>` is a string.
So, the actual code a user enters would look something like this:

```
(apex-info :version "2.4")
```

The syntax of programming is displayed in *italic 12 pt Times New Roman* similar to this:

(procedure [:concurrent] <index-clause> <procedure-level-clause>$^{+}$)

2.0 Getting Started

2.1. Setting up

To use Apex you'll need the following software:

1. *The Apex system*
   Using a standard web browser, Apex can be downloaded from the following web site: http://human-factors.arc.nasa.gov/apex.

   Apex is available for Macintosh, Windows, Linux, and Solaris based computers. Installation is simple as Apex comes "pre-built" and ready to start.

2. *Java Runtime Environment (JRE)*
   This is most likely already installed. If needed, the JRE may be obtained from the Apex web site along with installation instructions (see the README file).

3. *Text editor*
   Developing Apex applications requires programming in Common Lisp. By default, Apex runs inside GNU Emacs, the most popular editor for Lisp programming. However, any other text editor may be used.

2.2 Quick Tour

This section outlines the basic elements of using Apex via Sherpa, its graphical user interface. Using the attached Sherpa diagrams as a reference, follow these instructions to load, run, and inspect the results of a sample scenario modeling a person operating an automated teller machine (ATM).

1. *Start Apex*
   Directions for how to start and exit Apex, which vary depending upon operating system, are found in the GettingStarted.html document in the Apex instal-

lation folder/directory. Consulting these instructions, start Apex and its graphical user interface, Sherpa.

2. *Load an Application*

   a) Click the Start button in Sherpa. This "connects" Sherpa to the Apex system (which runs as a separate application).

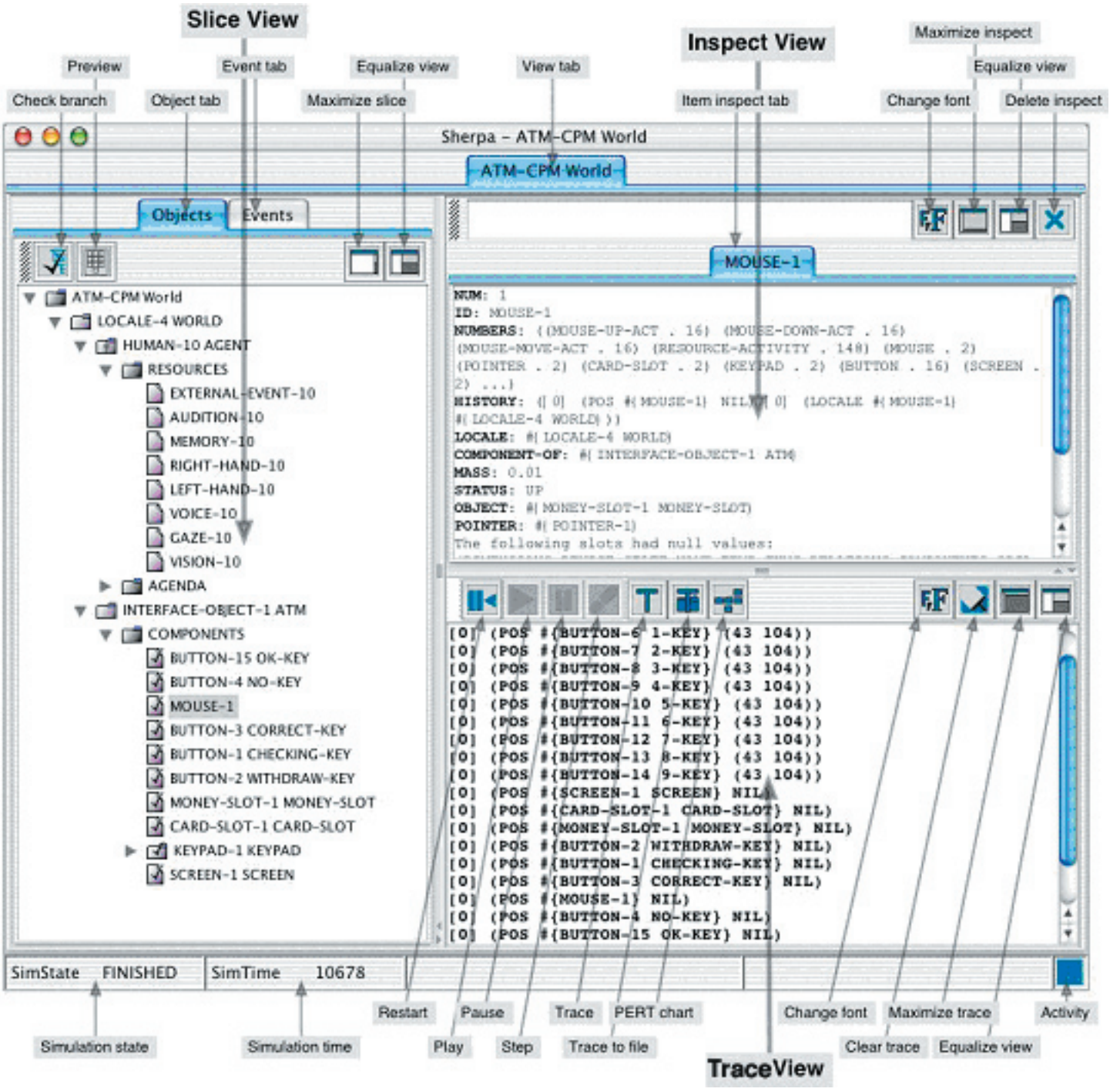


Figure 2.1 *Sherpa's user interface: Slice View, Inspect View, and Trace View*

b) Open an application. Select Load Application from the File menu. Using the directory browser, open the file <*apex*>*/examples/atm-worlds/cpm.lisp*, where <*apex*> is the Apex installation directory. Sherpa's screen will change to reveal the application control and viewing interface.

3. *Run the Application*
   Press the *Play* button. Events will print in the *Trace View* as the simulation runs.

4. *Inspect Objects*
   The *Slice View* lists the scenario objects in a collapsible hierarchical fashion. Click on the "lever" icons to expand objects. Click on objects to display information about them in *Inspect View*.

5. *View the PERT Chart*
   Click on the *PERT chart button* to generate a PERT chart for the simulation run. Inspect the chart and experiment with its manipulation controls.

   Note that the PERT chart window has become the top *view tab*. To bring up the application control interface, click on the ATM-CPM-WORLD view tab.

6. *Change Trace settings*
   Click on the *Events tab* to access the *Event View*. By default, only a small fraction of the trace data produced during simulation is shown. To see more, click on the *Set show level* drop-down menu and select *asa-low*, then click the *Trace* button to show a new (larger) subset of the events generated in the previous simulation run. See section 3.5.1 for more on controlling trace display.

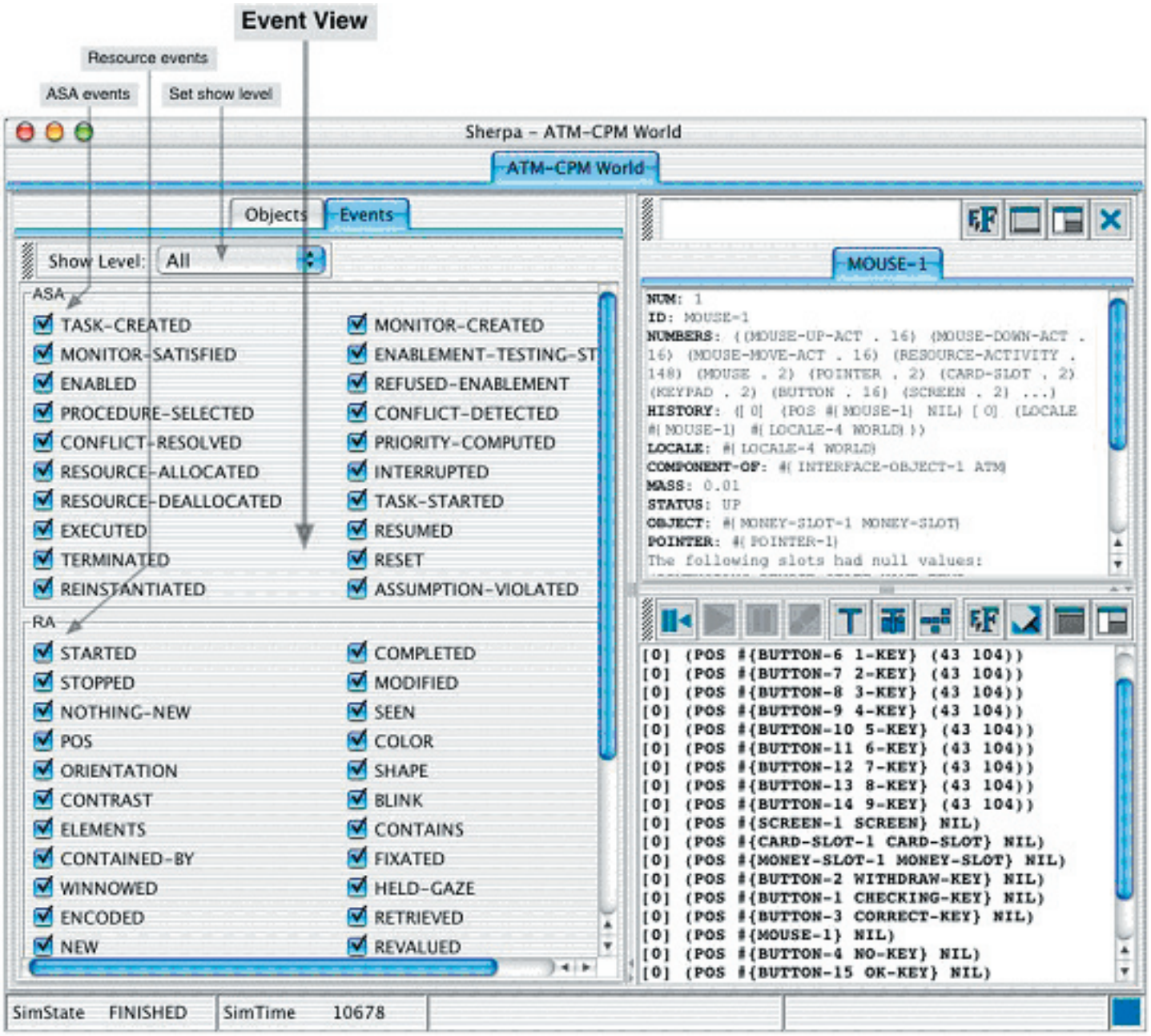


Figure 2.2 *Sherpa's user interface: Event View*

3.0 Using Apex

3.1 Interacting With Apex

Users interact with Apex mainly through three interface elements: a standard text editor such as Emacs; Apex's graphical user interface, Sherpa; and a Lisp interactive window, known as a *Listener*. In most cases, a user will wish to have all three of these elements available when building and running Apex applications.

A text editor is needed to create and modify Apex applications. Apex applications are written in the Common Lisp programming language, for which the most popular editor is Emacs. By default, Apex starts up inside an Emacs "buffer window", allowing access to the Lisp/Emacs integration provided by Apex's underlying Lisp system. If so desired, a different text editor may be used.

Sherpa is used to start Apex application runs, examine application elements, and to generate, format and display application output. It is possible to use Apex without using Sherpa. However, Sherpa provides the only means for obtaining graphical output from a simulation (e.g. PERT charts, object trees) and for pausing an application run interactively.

The Lisp Listener (or simply *Listener*) is an interactive text window always present when Apex is running. Normally, this is the `*apex*` buffer inside Emacs. Listeners are inherent to Common Lisp systems. Interacting through the Listener can be especially valuable when debugging Lisp code. A Listener can also be used in place of Sherpa as a primary means of interacting with Apex[1]. This can be done in two ways. First, the user can directly invoke Lisp functions that control Apex using functions described below (e.g. `(startapp)`). Second, a prompt-driven interface can be invoked by entering

```
(apex)
```

[1] **Warning**: using the Listener to interact with Apex while also using Sherpa may lead to unexpected behavior – only one means should be used in an Apex session.

in the Listener. This provides access to all the features of Apex, except for the graphical features of Sherpa. The prompt-driven interface is still in development; user feedback is especially encouraged.

Listeners display debugging information and other messages while Apex runs. Most of what is normally displayed is internal information that can be ignored. However, if an error occurs, the Apex run is interrupted and a debugging prompt appears, accompanied by an error message. Such occurrences are most frequent during development or modification of a model and are usually caused by Lisp programming errors.

3.2 Introduction to Apex Applications

Apex supports two classes of user applications:

Native applications – Applications that are fully contained in Apex. They use the Apex simulation engine, allowing an entire application to be a single process. For example, many Apex applications simulate one or more humans in a specified physical environment. Such applications, usually termed *simworlds*, include behavior models for all agents as well as object definitions describing the structure, appearance, and relationships between simulated physical objects.

Non-native applications – Applications in which one or more Apex components interact with an application external to Apex such as a simulation run on another computer or an embodied robot. The X-Plane® example provided with Apex is an example of a non-native application.

3.3 Loading an Application

In order to run an Apex application, it must first be loaded into Apex. There are three ways to do this.

1. Select from a list of recently loaded applications. In Sherpa, select `Recent Applications` from the File menu. In the Listener, invoke the Apex prompt (if necessary) by typing `(apex)` and enter `load` or lower case letter `l`.

   By default, Apex remembers the last five applications loaded. This value can be changed with the expression (`change-load-history-size N`) where `N` is a natural number. This can be entered in the Listener to affect the current session or be made persistent by placing it in your preferences file (3.6). If desired, the load history can be cleared by typing (`clear-load-history`) in the Listener.

2. Browse files and select an application to load from your local file system.

This is supported only in Sherpa. Select `Load Application` from the File menu.[2, 3]

3. Load a specified application from the Listener. Invoke the Apex prompt (if necessary) by typing `(apex)` and enter `load` or `l`. Enter the number of the last menu selection and you'll be prompted for an application file. Type in the full pathname of the desired file as a string, e.g.

   "`c:/apexapps/myworld.lisp`"[3]

3.4 Running an Application

Once an application is loaded, it may be manipulated in the following ways:

Starting the application - In Sherpa, click the Play button. In the Listener, type `(startapp)`. Unless there is user intervention, the application will run to completion or until a scheduled pause point (simulations only) arrives.

Pausing a running application - In Sherpa, click the Pause button. (If the Pause button is not selectable, pausing is not available for the application). It is not possible to interactively pause an application in the Listener, though simulations can be programmed to pause automatically in various ways.

Resetting the application - This restores the application to its initial state. In Sherpa, click the Reset button. In the Listener, type `(resetapp)`.

Single-stepping the application - Some applications have the ability to be advanced one step (e.g. time unit) at a time. Native Apex applications are constructed using an event driven simulation mechanism. Thus, for these applications, a step advances the simulation to the next scheduled simulation event(s) rather than by a fixed amount of simulated time. Click the Step button in Sherpa, which will be selectable if the application supports single stepping. In the Listener, type `(steppapp)` (which will have no effect if single-stepping is not supported).

---

[2] **Warning**: currently this feature will not work when Sherpa and Apex are running on different computers. In this case, you must use method (3) to load a new application (which subsequently makes the application selectable in the recent application menu).

[3] **Warning**: you must enter or select an *Application Definition File* (3.5.2). Loading any other kind of file will result in an unspecified behavior.

3.5 Creating a New Application

The information covered in this section apply to both *native* and *non-native Apex applications*.

3.5.1 Lisp Programming and Emacs

Apex applications are computer programs written in the Common Lisp language. They include code written in the Apex API, code written in PDL, and possibly arbitrary Lisp code Applications are created using a text editor. Emacs is strongly recommended because of its support for Lisp programming and convenient interface to Allegro Common Lisp®, the Lisp system upon which Apex is built. A good way to learn Emacs is from a tutorial accessible through its Help menu.

3.5.2 Application Definition File

Loading an application (3.3) causes Apex to load an Application Definition File (ADF). Every ADF contains the form:

```
(defapplication …)
```

This form names the application, specifies libraries (3.5.3) and other files that need to be loaded and defines how to initialize the application. It can also contain code that customizes the behavior of the application as described in section 3.4. See 5.4 for detailed information about this form. Many examples can be found in *<apex>/examples*, where *<apex>* is a directory name created by the user at the time of Apex's installation. A full example of an ADF is shown in Appendix E.

3.5.3 Application Files

It is acceptable for an *Application Definition File* (3.5.2) to include all the code (including PDL behavior specifications) needed for an application, but code from additional files will often be needed. This code can be made part of the application definition in either of two ways.

1. Files may be listed in the `:files` clause of `defapplication` (5.4).

2. Files may be loaded arbitrarily, anywhere in the ADF (3.5.2) or other Lisp files, using the function:

   ```
   (require-apex-file <filename>)
   ```

   where `<filename>` is a string naming the file.

The additional application files are typically Lisp files[4], but may include non-Lisp files, 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.

All Lisp files that comprise an application, including the Application Definition File and library files (discussed in the next section), are required to contain the form:

```
(apex-info :version <version>)
```

where `<version>` is a string naming the version of Apex for which the application is written. Example application files that come with Apex already contain this form. In newly created files, use "`2.4`" for `<version>`. The purpose of this form is to help flag potential incompatibilities between applications and future versions of Apex.

3.5.4 Libraries

A body of Apex code (e.g. PDL procedures, class definitions) can be shared conveniently among different applications using libraries. A library is in general a collection of related definitions that are grouped together for sharing across applications. A library might consist of one file or many files, but this difference is transparent to the users of libraries.

3.5.4.1 Using Libraries

An existing library may be included in an Apex application in either of two ways:

1. Include its name in the `:libraries` clause of `defapplication`, e.g.

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

2. Load it directly (on demand) with the `require-apex-library` form, e.g.

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

3.5.4.2 Creating Libraries

Like an Apex application, a library can be one file, or have multi-file structure. It has

---

[4] Lisp files may be loaded into Apex in either source or compiled form, but at this time compilation of Lisp source is not performed automatically by Apex.

a top-level file, called a *library file*, which may contain Lisp code. This file may constitute the entire library, or it may include other libraries (using `require-apex-library` defined in 3.5.4.1) or other files (using `require-apex-file` defined in 3.5.3). A library file's name must have the suffix `-apexlib` (e.g. `human-apexlib.lisp`). A library may be filed anywhere, though if it has several or more files, the library can be placed in a directory named after the library's "base name". For example, the human library can be found if it is filed as either `human-apexlib.lisp` or `human/human-apexlib.lisp`). Any number of libraries can exist and be available to applications.

3.5.4.3 Finding Libraries

The global variable `*apex-library-path*` specifies where libraries are found. It is a list of directories that are searched in the given order. The default value of this variable is:

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

The special symbol `:application` means that the application directory itself is first searched for libraries. The following two strings in this list use Common Lisp's *logical pathname* syntax. Any valid filename syntax for your computer platform may be used.

This search path may be modified as needed. For example, to have Apex first look in its provided libraries directory (3.5.4.4) and then in the directory `C:/me/apexlib`, enter the following form in the Listener:

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

For convenience, this form may be put in the user preferences file (3.6) and thus be in effect for all Apex sessions.

3.5.4.4 Provided Libraries

Apex comes with two sets of libraries:

1. `apex:apexlib` contains Components thought to be useful for a wide range of applications

2. `apex:examples:apexlib` contains libraries used by the example applications provided with Apex.

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

3.5.5 Worldbuilder

Worldbuilder provides a graphical interface for building physical environment models for Apex simworlds (native applications). It was created by students from Carnegie Mellon University and is available for download at the Apex web site.

3.6 User Settings and Other Files

When Apex starts, it looks for the existence of *a user settings file*, and loads it if the file exists. This is a Lisp file that users may create. It must be saved as the hidden file *~/.apexprefs* on Unix-like systems and *apex:apexprefs* in Windows. This file is for customizing the user's Lisp or Apex environment. It may contain arbitrary Lisp code, though its common function is for setting Apex parameters such as the library search path (3.5.4.3).

Apex automatically maintains other user-related information between sessions in two different files. There is the hidden file `~/.apexinfo` on Unix-like systems, `apex:apexinfo` in Windows, and apex:sherpa.ini on all platforms. *These files are generated and maintained by Apex. Do not edit them!*

3.7 Apex Output

Running an Apex application can generate two kinds of output: *event traces* and *PERT charts.*

3.7.1 Generating Event Traces

The activities of Apex agents and other entities (if any) are recorded as a chronology of events in an *event history*. Events are displayed as single lines of text specifying the time the event occurred, an associated agent (if relevant) and a description of the event. For example, the following event

```
[4235 Fred] (TASK-STARTED #{TASK-10 (SIGN-IN)})
```

represents that at time 4235 the agent Fred began a task to "sign in." By default, time is measured in milliseconds after the start of the Apex application run. If occurring in a simulation, this indicates simulated time – i.e. time in the chronological frame of the simulation, not in the real world.

Events are displayed in Sherpa's Trace View as they occur while the application runs. The trace appears in the Listener when Sherpa is not being used. Sherpa's trace view has a limited scroll size and it is possible to redirect trace output to the Listener by checking the "Trace To Listener" flag in the Trace menu. Regardless of whether events are displayed during a run, they may be viewed after a run (or while the application has been paused) by pressing the Trace button (see Figure 2.1). To request a trace in the Lis-

tener, type (`generate-trace`).

A simulation trace may be viewed in its entirety, but this may contain thousands of events or more. A user can specify filter criteria to reduce the amount of trace information displayed. Filtering criteria are applied both to trace data displayed at runtime and to trace derived from the stored event history. Events are most often filtered based on *event type* determined by the first element of an event description. For example, the types of the two example events below are *task-started* and *suspended*, respectively.

```
[12 Fred] (task-started #{task-21 (fly-to-waypoint)}
[45 Fred] (suspended #{task-19 (push-button-1)}
```

There are three basic ways to filter event traces:

1. The first is to specify a *show level.* A show level is a name that specifies a collection of *event types* to be shown. In Sherpa, click on the Event tab in the leftmost display pane; all event types associated with the currently loaded application are displayed next to checkboxes. The Show Level menu allows selection among predefined show levels. Selecting a show level causes event type checkboxes associated with that show level to become checked. In the Listener, show levels are set with the `show` function when used in the following form:

   ```
   (show :level <level-name>)
   ```

   where `<level-name>` is a symbol *without* quotes. Predefined show levels are described in Appendix A.

2. Using Sherpa, specify particular event-types of interest. Select the event tab as above, then click on checkboxes to toggle whether or not to have a particular event type shown. Note that selecting or unselecting event types modifies the choices associated with the previous show-level, though that show-level is still displayed on the interface. In the Lisp Listener, event types are selected with the `show` function when used in the following form:

   ```
   (show <event-type>)
   ```

   where `<event-type>` is a symbol *without* quotes. Event types are listed in Appendix A.

3. In the Listener (but not Sherpa), it is possible to filter events on parameters other than, and in addition to, event types. Like the previous features, this is done using the `show` function. The show function is described in Appendix A.

Traces generated with a particular filter setting may be saved to a file by typing the following form in the Listener:

```
(save-trace <filename>)
```

where `filename` is a string and may either be a full pathname, or just a filename. In the latter case, the trace is saved in the current application's directory.

3.7.2 Generating and Examining PERT Charts

A PERT chart for a specific agent in a simulation run may be generated by selecting desired agents in the Slice View, then clicking the PERT chart button located above the trace view pane. New tabs for the PERT charts are created and displayed. If no agent was selected, PERT charts for *all* agents will be generated. If there are more than 5 agents, a warning and confirmation request will appear first. PERT charts cannot be generated via the Listener. The PERT chart view can be manipulated in several ways.

- A slider bar provides zoom control
- The expand/contract buttons control distance between PERT boxes
- The timeline button toggles between a PERT view and a timeline view

3.7.3 Exporting a PERT Chart to Microsoft PowerPoint

Sherpa cannot create Microsoft PowerPoint® representations of PERT charts directly. Instead, it outputs Visual Basic® macros that can be read in from PowerPoint. PERT charts you create using the procedure below will not likely fit onto one slide, but will tend to trail off the right hand edge. You'll need to edit charts in Sherpa and/or PowerPoint to get good results.

1. Create a PERT chart in Sherpa

2. In Sherpa, press the button with the PowerPoint icon. Then select a folder and filename at the prompt. A Visual Basic macro representing the PERT chart will be written out at this location.

3. From PowerPoint select from the menu: *Tools* > *Macro* > *Visual Basic Editor*. This will open the visual basic editor.

4. From PowerPoint, load the macro created in step 2.

   On a Mac: From the Visual Basic interface, select *Insert* > *Module*. Select *Insert* > *File…* Set the *Show* field in the dialog selection box to *All Files*. Select the file you created in step 2.

   On a PC: From the Visual Basic interface, select *File* > *Import File*. Select the file you created in step 2.

5. Return to PowerPoint and click on the slide to contain the PERT chart. Select from the menu: *Tools* > *Macro* > *Macros* and run the macro "CreatePERTChart". For a large PERT chart, this may take a few moments to complete.

Note: To remove files created in step 2 (which will otherwise accumulate), go to the Visual Basic editor and select *Module*X in the Project window. From the menu, then select: *File* > *Remove Module*X.

3.8 System Patches

Patches provide extensions, modifications or fixes to the existing Apex software without requiring reinstallation. Users can acquire 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'll be able to find it easily. Instructions for downloading and installing patches will also be found there, 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 (e.g. those ending in .fasl). Newly installed patches will automatically be in effect the next time you start Apex. If you wish to install the patches without restarting Apex, type *(load-apex-patches)* at the Lisp prompt. A brief description of each patch is found in the file.

3.9 Getting Help

If you experience problems with Apex, please consult the Troubleshooting sections in this manual and in your Apex installation instructions. If necessary, contact the Apex development team by sending email to:

apexhelp@eos.arc.nasa.gov

Email is the strongly preferred means of technical support, and usually receives faster response than other means of contact. 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:

- Detailed description of the problem, including any error messages that appeared (in their entirety, cut and pasted if possible), the last thing you did before the problem occurred, and whether you could reproduce the problem.

- Your operating platform: type of computer and operating system, version of Apex (in "Help" menu of Sherpa), and version of Common Lisp (if applicable).

4.0 Procedure Description Language (PDL)

> *"By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and, in effect, increases the mental power of the race."*
>
> - Alfred North Whitehead

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 – e.g. normative behavior as defined by standard operating procedures; a task analysis describing observed or expected 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 – see Freed (1998a) for more detail.

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 (turn-on-headlights)
  (step s1 (clear-hand left-hand))
  (step s2 (determine-location headlight-ctl => ?loc)
  (step s3 (grasp knob left-hand ?loc) (waitfor ?s1 ?s2))
  (step s4 (pull knob left-hand 0.4) (waitfor ?s3))
  (step s5 (ungrasp left-hand) (waitfor ?s4))
  (step s6 (terminate) (waitfor ?s5)))
```

The procedure above represents a method for turning on the headlights in some cars and

illustrates several important aspects of PDL. One important point is that a 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 above, the steps labeled `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, includes the clause `(waitfor ?s1 ?s2)`. This means that step `s3` becomes enabled only when steps `s1` and `s2` have terminated.

Procedures are invoked to carry out an agent's active tasks. Tasks, which can be thought of as agent goals[5], 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 action. If so, the specified action is carried out and then the task is terminated. There are a limited number of primitive action types (see section 4.3), each with a distinct effect.

If the task is not a primitive, the Action Selection Architecture retrieves a procedure whose index clause matches the task. For example, a task of the form `(turn-on-headlights)` 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. In this example, there are six steps so six new tasks will be created. 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).

---

[5] The term task generalizes the concept of a classical goal – i.e. a well-defined state, expressible as a proposition, that the agent can be seen as desiring and intending to bring about (e.g. "be at home"). Tasks can also, e.g., encompass multiple goals ("be in car seat with engine started and seatbelt fastened"), specify goals with indefinite state ("finish chores"), specify goals of action rather than state ("scan security perimeter "), and couple goals to arbitrary constraints ("be at home by 6pm").

4.1 Action Selection Architecture (ASA)

The Action Selection Architecture[6] is the algorithm 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. The architecture outputs commands to resources. 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, 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, 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 (see section 4.2.2) and the `select` clause (4.2.5). 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 (4.2.4).

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. Integrating contingency-handling behav-

[6] To some, this term implies that the architecture performs AI planning tasks, but not scheduling or control. The term Action selection architecture was chosen to be happily ambiguous about the underlying technology.

ior with nominal behavior is quite challenging and benefits from building certain principles and heuristics into the architecture. For example, Apex incorporates a heuristic preference for continuing an ongoing task over allowing a new task to interrupt. The preference can be increased or negated using the interrupt-cost construct (4.2.10).

Reactive mechanisms combined with looping (4.2.6) and branching (4.2.2, 4.2.4, and 4.2.5) 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 primitive action types for dynamically asserting, retracting and parameterizing scheduling constraints (4.2.4, 4.2.8, 4.2.9, 4.2.10, 4.3.5, 4.3.6, and 4.3.7).

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 (4.2.4 and Appendix D), action selection (4.2.5), specification of execution parameters (4.2.6, 4.2.9, 4.2.10, and 4.2.11), and specification of the actions themselves (see "special procedures" in 4.2.1).

4.2 PDL Syntax

PDL syntax will be described using the following conventions:

- ( ) all PDL constructs are enclosed by parentheses
- [ ] 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 terms are used. A procedure-level clause is a language construct embedded directly in a PDL procedure. Examples include `index` clauses and `step` clauses. Step-level clauses such as `waitfor` are embedded directly in a `step` clause. The procedure construct is itself a first-class construct, meaning that it is not embedded in any other language element. A pattern parameter is a parenthesized expression that may contain variables (denoted as a symbol starting with a question-mark such as `?x`). Patterns, which are matched against each other by the pattern matcher (see Appendix D), 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. An Apex variable is a symbol whose first character is a question mark – e.g. `?x`. Symbols and s-expressions in PDL clauses may contain Apex variables.

4.2.1 `procedure`

Type: first-class construct

Syntax:*(procedure [:concurrent] <index-clause> <procedure-level-clause>$^+$)*
(procedure [:sequential|:ranked] <index-clause> <step-clause>$^+$)
(procedure :special <index-clause> <procedure-level-clause>$^+$ <s-expression>)

There are four types of procedures: concurrent, sequential, ranked and special. All types must contain an index clause. By default, procedures are of type concurrent. This means that all tasks generated from the procedure's steps are assumed to be concurrently executable, except where ordering is specified by waitfor clauses. A concurrent procedure will usually include an explicit termination step such as `s4` in the example procedure below left. In this case, the parent task *{task-15 (open door)}* will terminate when the last of its subtasks *{task-18 (push)}* terminates.

```
(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))))
```

As in this example, 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))
   (grasp door-handle)
   (turn door-handle)
   (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 specified. Only the activity-description argument of each step is specified; the symbol `step`, the step-tag argument and step-level clauses are not required or allowed. Sequential procedures are not really a separate type, but 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 (4.2.13) are added to each step. Rank values in these procedures are in ascending order of appearance. Thus, the following procedure is equivalent to the previous one:

```
(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
   (index (open door))
   (grasp door-handle)
   (turn door-handle)
   (push))
```

Special procedures are a way to call Lisp code directly during task execution. This is useful for controlling and accessing data from processes external to the Action Selection Architecture and for carrying out functions that would be awkward or impossible to represent purely in PDL. In the first example below, the procedure uses the simulation engine function `end-trial` to stop the simulation from continuing (perhaps indefinitely) past the point of interest.

```
(procedure :special
   (index (stop simulation trial))
   (end-trial))
```

In the next example, a special procedure is used to compute the distance between two points in a plane. Values returned by the Lisp body of a special procedure are bound to variables in the return value form (if any) of the calling step (see 4.2.3). Thus, executing a step such as:

```
(step s5 (compute-distance ?p1 ?p2 => ?d) (waitfor ?s4))
```

would cause the procedure to be called and its return value bound to the variable `?d`.

```
(procedure :special
   ; points are lists of the form (x y)
   (index (compute-distance ?point1 ?point2))
   (sqrt (exp (- (first ?point1) (first ?point2)) 2)
   (exp (- (second ?point1) (second ?point2)))))
```

Special procedures may include procedure-level clauses other than `index`, but may not include any `step` clauses. When a task for which a special procedure has been selected becomes enabled, that task is executed and then terminated just as if it were a primitive action.

4.2.2 `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.

Apex uses the pattern matcher from Norvig (1992), which 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` will match 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 Norvig (1992) and Appendix D for more information on the pattern matcher.

4.2.3 `step`

Type: procedure-level clause

Syntax: *(step <step-tag> <step-description [=> {var|pattern}]> [step-level-clause]*)*

`step` clauses in a procedure specify the set of tasks to be created when the procedure is invoked and may contain additional information on how the tasks should be executed (e.g. ordering constraints). Each `step` must contain a step-tag and step-description; optionally, an output variable and/or any number of step-level clauses may be added.

A step-tag can be any symbol (as defined by Lisp), although no two steps in a procedure can use the same tag. Step-tags provide a way for steps in a procedure to refer to one another. In particular, whenever a new task is created from a procedure step, the Action Selection Architecture creates a variable based on the step tag and binds that variable to the new task. For example, when `(step s4 (go west))` is used to create *{task-92 (go west)},* the variable `?s4` is created and bound to the data structure for task-92. The task refinement process also generates the variable `?self` which is bound to the task being refined – i.e. the parent to task-92 in this example. This allows subtasks to refer to their parent task.

The step-description, the part of the `step` clause that describes behavior, must be a parenthesized expression corresponding either to the index of one or more procedures in the agent's procedure library or to a PDL primitive action type (see section 4.3). It may contain variables. When a task is enabled, the value of the task description is set to equal the step description with any variables replaced by values. The task description is used to invoke a primitive action is appropriate, or if not, matched against procedure `index` clauses to select the correct procedure.

The step-description may include the special symbol `=>` followed by a variable or other pattern. This specifies one or more output variables that become a return value when the task derived from a step terminates. Thus,

```
(step s1 (find volume control => ?location))
```

would create a task such as *{task-22 (find volume control)}*. When this task terminates, it should supply a return value which will be bound to the variable `?location`. See the

description of the 'terminate' primitive (section 4.3.2) for a description of how return-values are generated.

It is an error for a task description to contain a variable whose value is undefined at the time the task is enabled. This is avoided by making task specificity a precondition using 'waitfor' clauses. Some 'waitfor' preconditions bind values directly. For example, `(waitfor (on ?object table))` not only waits for something to be on the table but also binds the variable `?object` as a side effect. Other preconditions wait for the completion of tasks that insure a variable gets bound. For example, if step `s2` waits for step `s1` above to complete, this insures that the variable `?location` will be bound when a procedure for `s2` is selected.

4.2.4 `waitfor`

Type: step-level clause

Syntax: *(waitfor {<pattern>|<step-tag-variable>}$^{+}$ [:and <test>$^{+}$])*

A `waitfor` clause defines a set of task preconditions that must be satisfied for the task to become enabled – i.e. eligible for execution. Each pattern argument defines a single precondition that is unsatisfied when the task is created. The precondition is considered satisfied when a cogevent matching the pattern is detected. Cogevents are representations of events that have become available to the Action Selection Architecture. Some cogevents are generated by the Action Selection Architecture and reflect occurrences within it (e.g. an event signaling that some task has terminated). Others cogevents are generated externally, typically by agent perceptual resources such as vision (e.g. to signal that an object has been detected).

It is important to note that `waitfor` preconditions are satisfied by events, not by states represented in memory. For example, if a task comes into existence with a precondition of the form `(on book table)` and a proposition of the same form exists in memory[7], this will not satisfy the precondition; the task will remain in a pending (non-enabled) state until matched to a corresponding cogevent. The Action Selection Architecture prescribes no particular method for detecting when preconditions are satisfied in the current state. One possibility is to include a step in the procedure to explicitly check whether a precondition is satisfied, either perceptually or by memory retrieval. Note: only allowing events to satisfy preconditions facilitated specification of reactive behavior since it will sometimes be desirable to act only in response to change.

`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 generates events of the form `(terminated` <*task*>`)` when a task is terminated, so a clause such as `(waitfor (terminated ?s3))` will impose order with respect to the task bound to the task-

[7] The Apex architecture does not include a built-in memory for world-state. Typically, this function is handled by a resource component defined to take encode and retrieve commands from the agent mechanisms.

tag-variable `?s3` (see 4.2.3 for information on task-tag-variables). Termination preconditions can be expressed using an abbreviated form: `(waitfor` <*task-tag-var*>`))` == `(waitfor (terminated` <*task-tag-var*>`))`. Thus, the expression `(waitfor ?s3)` is equivalent to `(waitfor (terminated ?s3))`.

Preconditions in a `waitfor` clause are conjunctive; all must be satisfied for the task to become enabled. Optional tests (s-expressions) following the keyword `:and` add additional conjunctive preconditions. These (special) preconditions are evaluated after all of the normal preconditions (specified before the `:and`) are satisfied. If any of these expressions evaluate to nil, the special precondition is considered unsatisfied and the task does not become enabled. Moreover, it can never become enabled since the tests are not performed again. This restricts the use of special conditions to representing conditional branches in a procedure. In the following example, the agent's behavior depends on the relative value of the variables `?my-score` and `?his-score`.

```
(step s1 (cackle with glee)
(waitfor (final-score ?my-score ?his-score :and
   (>= ?my-score ?his-score))))
(step s2 (sulk despondently)
(waitfor (final-score ?my-score ?his-score :and
   (< ?my-score ?his-score))))
```

It is possible to specify disjunctive preconditions using multiple `waitfor` clauses. For example, step `s2` prescribes terminating a hole-digging task if either the hole has been dug to the specified depth or if the shovel needed to dig breaks.

```
(step s1 (dig hole ?depth))
(step s2 (terminate)
   (waitfor ?s1)
   (waitfor (broken shovel)))
```

Correctly specifying `waitfor` preconditions is perhaps the trickiest part of PDL. One important issue arises from the fact that, in Apex, preconditions are satisfied independently, not jointly as might sometimes seem more intuitive. For example, one might want to express a behavior that becomes enabled in response to a red light, representing this with:

```
(waitfor (color ?object red) (luminance ?object high)).
```

However, vision might detect `stopsign-1` that is red but not a light and generate a co-gevent of the form `(color stopsign-1 red)`. This will satisfy the first listed precondition, binding the variable `?object` to `stopsign-1`. The second precondition will then remain unsatisfied unless `stopsign-1` becomes highly luminous. Planned improvements to PDL will provide the flexibility to express joint preconditions.

4.2.5 `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 s1 (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.

Known bug: a `step` may only contain one `select` clause.

4.2.6 `period`

Type: step-level clause

Syntax: *(period :recurrent [<test>] [:enabled [<test>]] [:reftime {enabled|terminated}] [:recovery <interval>])*

The `period` clause is used to create and control repetition. The simplest form of the clause, `(period :recurrent)` declares that the task should be restarted immediately after it terminates and repeat continuously. In this case, repetition will only cease when its parent task terminates or the task is explicitly terminated (by a `terminate` primitive action). The optional test condition is a Lisp expression that is evaluated; if nil, the task does not repeat. This makes the task behave as if in a repeat-until loop.

By default, any `waitfor` preconditions associated with a recurrent task are reset to their initial unsatisfied state when the task restarts. If present, the optional `:enabled` argument causes the task to restart in an enabled state – i.e. with preconditions satisfied. An optional test for enablement is evaluated at restart-time; if it evaluates to nil, the task is restarted with all preconditions unsatisfied as in the default case.

The optional `:reftime` argument is used to specify whether to start a new instance of the task when the old instance terminates or when the old instance becomes enabled. Restarting at termination time is the default, producing repetition in the normal sense. If the value of `reftime` equals `enabled`, the task does not repeat; instead a

whole new instance of the task is created, coexisting with the current one. This option is provided as a way to specify response policies – i.e. that a response task should be generated to a given class of events even if one or more such response tasks are already ongoing.

```
(step s5 (shift-gaze ?visob)
   (waitfor (new (visual-object ?visob)))
   (period :recurrent :reftime enabled))
```

For example, the step above represents a policy of shifting gaze to any newly appearing object, even if it appears while in the process of shifting gaze to a previously appearing object. If the task only recurred at terminate-time, objects appearing during a previous gaze-shift response would be ignored. To prevent infinite generation of new task instances, steps specified with enable-time recurrences cannot be restarted in enabled state. Thus, the enabled parameter must be nil (the default) and the `step` must include `waitfor` preconditions.

The `:recovery` argument temporarily reduces a repeating task's priority (4.2.9) 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 task for repeatedly monitoring the fuel gauge is reduced to 0 immediately after performing the task, and gradually rises to its full normal value over a period of 30 minutes.

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

4.2.7 `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 se-

lection, 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 s1 (examine indicator ?indicator)
   (forall ?instrument in
      (fuel-pressure air-pressure temperature))
   (period :recurrent))
```

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

4.2.8 `profile`

Type: procedure-level clause

Syntax: *(profile <resource>+)*

The `profile` clause lists discrete resources required for using a procedure.[8] 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-target-gear-position ?gear => ?position))
   (step s3 (move right-hand to ?position) (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
   (index (shift manual-transmission to ?gear using ?hand))
   (profile ?hand)
   ...)
```

---

[8] The profile clause is only used for "blocking" resources such as hands and eyes that can only be allocated to one task at a time, but may be reallocated freely. There are currently no mechanisms to support reasoning about "depletable" resources such as fuel or money.

Resource preconditions are not determined until a procedure is selected, and therefore not 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` (see 4.2.9)
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` (4.2.10). By default, interrupt cost is slightly positive, producing a weak tendency to persist in rather than interrupt a task.
5. Tasks at any level in a task hierarchy may require and be allocated resource. A task does not compete with its own ancestor.
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 action `start-activity` (4.3.1). Resources named in a `profile` clause that do not correspond to an element of the resource architecture are *virtual resources*.

4.2.9 `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. 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 `re-prioritize` (4.3.5) is executed on the task. Whenever a task begins a new resource competition, its `priority` is recomputed.

A `step` may have multiple `priority` clauses, in which case, the `priority` value from each clause is computed separately. The associated task is assigned whichever value is the highest. This value is the local priority value. Tasks may also inherit priority from ancestor tasks. A task could have one or more inherited priorities but no local priority. Alternately, it may have no inherited priorities but a local priority. In all cases, task priority equals the maximum of all local and inherited values.

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

4.2.10 `interrupt-cost`

Type: step-level clause

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

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

4.2.11 `assume`

Type: procedure-level clause

Syntax: *(assume <var> <proposition> <duration>)*

An `assume` clause declares that a specified proposition should be treated as an assumption. By default, the variable specified in the `assume` clause is set to *T*, indicating that the assumption has not been contradicted. If a cogevent contradiction occurs, then the value of the variable is set to nil. After an amount of time passes equal to the duration parameter, the value reverts to *T*.

The `assume` clause is meant to be used for procedure selection, allowing the architecture to select alternative means for carrying out a task in non-standard conditions. For example, the following procedure selects *route B* (rather than *route A* as usual) for getting home from work if there is accident on highway-5.

```
(procedure
   (index (get home from work))
   (assume ?clear-path (accident-on-path route-a false) (1 day))
   (step s1 (enter and start car))
   (step s2 (drive route ?selected-route)
      (select ?selected-route
         (if ?clear-path 'route-a 'route-b)))
      (waitfor ?s1))
   (step s3 (terminate) (waitfor ?s2)))
```

One very unusual aspect of the `assume` clause is that it applies not to tasks, but to procedures. In other words, the presence of the procedure in the procedure set of an agent causes the agent to track the specified assumption. If an event contradicting the assumption occurs, then this is reflected in the value of the assumed variable even if the procedure has not been selected for any current tasks. If such a task comes into existence during the interval between a detected violation of the assumption and the time when the assumption variable reverts to *T*, the assume variable will have the value nil for that task.

A cogevent is considered to violate the specified assumption if the assumption proposition ends in a Boolean value (*T*, nil, true, false) and the cogevent has the same form with the last value in the form holding the opposite value. For example, a cogevent of the form `(accident-on-path route-a true)` would violate the assumption in the example above. Assumption violation also occurs if a cogevent occurs indicating a changed value in a fluent proposition. For example, the cogevent `(color danger-indicator red)` violates an assumption proposition of the form `(color danger-indicator green)` as long as color propositions have been declared fluents (4.2.12).

To track the truth value of declared assumptions, the architecture automatically generates a procedure with `(index (monitor-assumptions))` and steps for monitoring each assumption specified in an assumption clause. The example above would cause a `step` such as the following (simplified)

```
(step g813 (set-temporary-value ?selected-route nil (1 day))
      (waitfor (accident-on-path route-a true))
      (period :recurrent))
```

to be added to the monitor assumptions procedure. This procedure is automatically selected and executed when the agent is initialized, so assumption monitoring is always active. Since the assumption variable is an Apex global variable, the value is not tied to the creation or termination of any task and is accessible to all tasks.

4.2.12 `declare-fluent`

Type: First-class construct

Syntax: *(declare-fluent <pattern> <var-list>)*

Fluents are propositions that can contradict other, similar propositions. If propositions are presented in a temporal sequence, they can make other propositions obsolete. For example, propositions *1* and *2* below are contradictory because, quantum mechanics aside, a device cannot be both on and off at the same time. If proposition *1* is presented, followed at some later time by *2*, this can be interpreted as a change of state that makes *1* obsolete. Propositions *3* and *4*, in contrast are not in contradiction because an object can be inside multiple containers.

(1) (power television-1 on)
(2) (power television-1 off)
(3) (in television-1 living-room-1)
(4) (in television-1 house-1)

The `declare-fluent` construct specifies that a given propositional form is a fluent. The pattern parameter is a list containing constants and variables. The variable-list parameter identifies pattern elements that determine whether the two propositions are potentially in conflict. Actual conflict requires some difference in value in any remaining variable element. For example,

```
(declare-fluent (power ?device ?state) (?device))
```

propositions *1* and *2* above both match the fluent pattern. Because they have the same value for `?device`, they are potentially in conflict. Because they have different values for the remaining variable specified in the fluent pattern (i.e. `?state`), they are actually in conflict. The propositions below, in contrast, do not conflict with either *1* or *2*.

(5) (power television-2 off)
(6) (weight television-1 100)

Fluent definition is used in conjunction with the `assume` clause (4.2.11) and can be used to define the information handling behavior of agent resources such as vision and memory.

4.2.13 `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. 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 (4.2.9).

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

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

This procedure specifies that activities related to determining a specified person's phone

number can be carried out in parallel with activities for writing the number down – i.e. the latter task and all of its descendant subtasks (e.g. {task-25 `(grasp pencil)`}) do not have to wait for the former task to complete. However, resource conflicts will automatically be resolved in favor of the better-ranked task – i.e. the one with the lower `priority` 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` (4.2.9). Otherwise, rank values for *A'* and *B'* are inherited and used to resolve the conflict.

4.3 PDL Primitives

Primitives are actions whose effects are defined by the Apex architecture rather than by a PDL procedure. They cannot be further decomposed into more fundamental tasks. The term **operator** is used for behaviors that are low-level from the point of view of a particular domain or task model. For example, in some models of human-computer interaction, behaviors such as pushing a button and moving a mouse to a target location might be considered operators. Operators are generally represented as PDL procedures that employ primitives, particularly `start-activity`. The full set of Apex primitives are described in the sections below.

4.3.1 `start-activity`

Type: primitive

Syntax: *(start-activity <resource> <activity-type> [:duration <time>] [<parameter-value-pair>]*)*

The `start-activity` primitive is used to initiate action in a module external to the Action Selection Architecture. Like all primitive tasks, a `start-activity` task takes zero time to execute and is terminated immediately[9]. However, an activity started by the primitive will typically go on for some non-zero time interval. To allow PDL to influence the activity during this interval and to respond when it completes, the `start-activity` returns a pointer to a representation of the activity. For example, the `start-activity` step in the following procedure signals a resource module (either left-hand or right-hand) to begin an activity of type `pressing`. A representation of the activity is

[9] The term "task" is reserved for actions and potential actions represented within the action selection architecture. "Activities" are performed outside the architecture.

returned when step `s1` terminates and is bound to the variable `?a`. The activity's completion is later (1 second later) signaled by a cogevent of the form `(completed <`*activity*`>)`, which, in this case, results in the termination of the overall task.

```
(procedure
   (index (press button ?b with ?hand))
   (profile ?hand)
   (step s1 (start-activity ?hand pressing
      :target ?b :duration (1 second) => ?a))
   (step s2 (terminate) (waitfor (completed ?a))))
```

A start-activity task essentially sends a message to a resource[10] module to begin doing something. A `start-activity` step must specify the resource that will receive the message followed by the type of activity to be initiated. Other parameters may then be specified including `:duration` and others specific to the activity type (e.g. `pressing` activities require a `:target`). If no duration parameter is specified in PDL, then the duration is determined by the resource module and/or the activity type definition.

4.3.2 `terminate`

Type: primitive

Syntax: *(terminate [<task>] [>> <return-value>])*

A `terminate` step defines conditions for stopping execution of a specified task. By default, the target task is the one whose associated procedure contains the `terminate` step. Optionally, the step can specify some other task to be terminated. For example, the procedure below specifies that the agent should whistle while it works, but stop whistling if it gets chapped lips. The gold mining task, parent of the tasks generated from steps of the procedure, terminates when the work is done.

```
(procedure
   (index (mine gold))
   (step s1 (whistle))
   (step s2 (work))
   (step s3 (terminate ?s1) (waitfor (chapped lips)))
   (step s4 (terminate) (waitfor ?s2)))
```

Terminating a task has a number of effects:

- The task's state is set to *terminated.*

[10] Only resources represented by a module external to the action selection architecture, and thus a component of the agents resource architecture, can receive start-activity messages. Resources named in profile clauses but not externally represented can still be the subject of allocation decisions. These are "virtual resources."

- The task is removed from the Action Selection Architecture's agenda.
- The architecture stops monitoring `waitfor` preconditions associated with the task.
- A cogevent of the form `(terminated` <*task*>`)` is generated (4.2.4).
- Any resources allocated to the task are deallocated.
- All of its subtasks are themselves terminated (indirect termination).
- If it is a periodic task (4.2.6) that passes its recurrence test and was not indirectly terminated, the task is restarted.

4.3.3 `reset`

Type: primitive

Syntax: *(reset <task>)*

`reset` causes the target task to terminate and then restart with all preconditions satisfied. It is generally used for trying again after a failure. For example,

```
(procedure
   (index (start-engine))
   (step s1 (turn-key))
   (step s2 (reset) (waitfor (engine-sound sputtering)))
   (step s3 (terminate) (waitfor (engine-sound turned-over))))
```

4.3.4 `cogevent`

Type: primitive

Syntax: *(cogevent <event>)*

The `cogevent` primitive generates a cogevent of the specified form, potentially matching task preconditions just as cogevents generated by resources (especially perceptual resources) or by the Action Selection Architecture. One important use of this primitive is to represent states that are inferred but not directly observed, such as hidden effects of an agent action. For example, step `s4` generates an event representing the inference that an elevator has been summoned after pressing a button for this purpose.

```
(step s3 (press button elevator-down-button) ..)
(step s4 (cogevent (summoned elevator)) (waitfor ?s3))
```

The <*event*> parameter of a `cogevent` step can be any parenthesized expression not containing variables.

4.3.5 `reprioritize`

Type: primitive

Syntax: *(reprioritize [<task>])*

`reprioritize` steps are used to specify conditions in which task priorities might have changed; justifying reevaluation of resource allocation decisions. A reprioritize action causes the architecture to recompute the specified task's priority, then initiate a general competition for the resource(s) needed by the task. If the task is enabled but has not been allocated resources, this may result in an immediate interruption of the task currently using those resources. If the task is currently ongoing, reprioritization may cause it to be interrupted.

4.3.6 `hold-resource`

Type: primitive

Syntax: *(hold-resource <resource-name> [:ancestor <integer>])*

`hold-resource` adds a resource to the list of resources a task needs in order to execute and then causes the task to compete for the resource with other contenders. Whereas the `profile` clause (4.2.8) establishes resource requirements as the task is enabled and its procedure selected, `hold-resource` adds requirements while the task is already ongoing. If the task competes successfully, there is no immediate effect. If some other task requiring the specified resource has higher priority, the task is interrupted.

The optional `ancestor` parameter specifies the target task. By default, the new requirement is added to the parent of the `hold-resource` task – i.e. the task whose associated procedure contains the `hold-resource` step. This corresponds to an `ancestor` value of 1 (1 level up the task hierarchy). Higher values target tasks higher in the hierarchy.

4.3.7 `release-resource`

Type: primitive

Syntax: *(release-resource <resource-name> [:ancestor <integer>])*

`release-resource` removes a resource from the list of resources a specified task requires in order to execute, and then causes the task to compete for its needed resources. It is typically invoked while the task is ongoing, freeing up the resource for use by some other task. The optional `ancestor` parameter specifies the target task. By default, the resource requirement is subtracted from the parent of the `release-resource` task – i.e. the task whose associated procedure contains the `release-resource` step. This corresponds to an `ancestor` value of 1 (1 level up the task hierarchy). Higher values target tasks higher in the hierarchy.

4.4 PDL Variables

An understanding of how variable binding occurs and where the information comes from is crucial for specifying behavior in PDL. Variables in PDL become bound (and rebound) to values in several different circumstances as summarized below:

- Variables in an `index` clause are bound after the procedure selection.
- Variables in a `profile` clause are bound after the procedure selection.
- `step` tags are turned into variables and bound to tasks during task refinement.
- Variables in `waitfor` clauses are bound when matching cogevents occur.
- Variables in a `selection` clause are bound prior to the procedure selection.
- Variables in a return value form (following a `=>`) are bound at task termination.
- The map variable in a `forall` clause is bound during task refinement.
- Global variables are initially bound as the agent is initialized.

Apex maintains two kinds of variables: local and global. Local variables are defined with respect to a set of sibling tasks – i.e. immediate subtasks of a common parent. For example, the task *{task-25* `(get milk from refrigerator fridge-1 with right-hand)`*}* might become enabled and the following procedure selected to carry it out:

```
(procedure
   (index (get ?item-type from refrigerator
      ?refrigerator with ?hand))
   (profile ?hand)
   (step s1 (open door of ?refrigerator with ?speed)
      (select ?speed (if (> (hunger ?agent) 5)
         'quickly 'slowly)))
   (step s2 (find ?item-type in ?refrigerator => ?location)
      (waitfor ?s1))
   (step s3 (grasp object at ?location with ?hand)
      (waitfor ?s2))
   (step s4 (remove hand ?hand from ?refrigerator)
      (waitfor (grasped ?item)))
   (step s5 (close door of ?refrigerator) (waitfor ?s4))
   (step s6 (terminate) (waitfor ?s5)))
```

In selecting the procedure, the variables `?item-type`, `?refrigerator` and `?hand` become bound to the values *milk*, *fridge-1* and *right-hand* respectively. Later, when *task-25* is allocated the *right-hand* resource, new tasks will be created including:

```
{task-28 (open door of refrigerator-1 with ?speed)}
{task-32 (close door of refrigerator-1)}
```

Together these local variable bindings generated by selecting a procedure for task-25 form the **local context** for these tasks. The local context is stored with the parent task. Note that the printed form of these tasks has some variables replaced by values and some as variables. The writing convention is that values are shown instead of variables if bindings have been established. Just after task refinement, the variable `?refrigerator` is bound but `?speed` is not. This convention does not imply that these values are fixed for the lifetime of the task. Generally, if a binding changes, the task description will change to reflect this. Tasks are stored with variables unbound; replacement occurs as needed based on the current local context.

When the task refinement process creates new subtasks, new bindings are added to the local context. First, a new variable is created for each step tag in the selected procedure. For instance, the variable `?s5` is created and bound to *task-32*; the binding is then added to the local context stored with *task-25*. Second, the variable `?self` is created and bound to *task-25*, enabling subtasks of *task-25* to refer to their parent.

Following task refinement, the state of each task is established. Tasks with `waitfor` preconditions are initialized in the *pending* state and must await enabling cogevents. Tasks such as *task-28* have no preconditions and thus start in an *enabled* state. Before the procedure-selection for this task is performed, its `select` clause is evaluated. In this case, the variable `?speed` will either be assigned the value *slowly* or *quickly* depending on the hunger value of the agent. This new variable binding will then be added to the local context, making it available to specify *task-28* and any of its siblings.

With the value of `?speed` defined, *task-28* becomes fully specified[11]. A procedure for it is selected, it is executed in the usual fashion and later terminates, enabling *task-29* for finding milk in *fridge-1*. The termination of *task-29* has an important side effect. It returns a value which is bound to the variable `?location`. The binding is added to the local context.

Bindings are frequently generated as cogevents match to preconditions expressed in a `waitfor` clause. In this case, it is assumed that a hand resource automatically generated an event of the form `(grasped` <*object*>`)` whenever it succeeds in a grasp action. In this case, the hand generates `(grasped milk-1)` which causes the variable `?item` to become bound to `milk-1`. This binding is then added to the local context.

Global variables are initially defined when the agent is initialized. The only global variable of general importance is `?agent` which is always bound to a representation of the intelligent agent as a whole (although see 4.2.11). To use this effectively requires knowledge of the kinds of state information stored in an agent structure and how to access them (6.5). The example used above in which the hunger value of the agent is accessed is fanciful, though it is possible to extend the general agent model to include any kind of state data.

Information that results in variable bindings comes from several places. First, it

[11] All the variables in a task description must be specified prior to selecting a procedure – i.e. all must be assigned values either before the task becomes enabled, during enablement as a consequence of cogevent matches, or during the procedure selection process by a select clause. If procedure selection is attempted for a task that has not been fully specified, this produces an error.

can come from processes internal to the Action Selection Architecture itself. For example, the architecture creates the tasks that get bound to step variables and generates cogevents signaling, e.g. task termination, task interruption and resource allocation (Appendix A). Second, resources generate cogevents describing the internal state of those resources and, in the case of perceptual resources, external events and states. Finally, the Action Selection Architecture can in principle retrieve information from a memory component. The Action Selection Architecture does not include a memory element, although see Freed (1998a) for an example.

4.5 Miscellaneous Features

4.5.1 Agent's Initial Task

An agent's initial task is specified with the `:initial-task` initarg, whose value is a procedure invocation and defaults to `'(do-domain)`. For example,

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

If `:initial-task` is not given when creating an agent instance, a PDL procedure whose index is `(do-domain)` must be defined.

4.5.2 PDL Partitions (Bundles)

A PDL procedure can be associated with some named category (called a *partition* or *bundle*) that can be referred to during agent creation. This allows different agents to have different skill sets. PDL procedures can be optionally assigned to a partition using the `use-bundles` form. Agents can be assigned a given bundle using the `:use-bundles` initarg. See the Roshambo simworld (*<apex>/examples/roshambo.lisp*) for examples.

5.0 Apex Programming Guide

This chapter of the manual, still under development, is meant to contain detailed descriptions of all functions that comprise the Apex Application Programming Interface (API), aside from PDL, which was documented in the previous chapter. The descriptions given here currently appear in alphabetical order.

5.1 `activity`

In Apex, the main role of simulation is to allow events to play out over time. An activity is a representation of such a time-structured event. In the simplest case, an activity is a single, discrete, delayed response to some occurrence such as a message to a simob. A "*message*" is simply the invocation of one of the simob's methods. But, an activity can also represent multiple and/or continuous responses. For example, the activity of falling (e.g. results from the removal of a supporting structure for a physical object) produces responses such as changing the position and motion-vector of an object, as well as an eventual collision.

`activity`, which is a kind of simob, is the superclass (direct or indirect) of all application-defined activity classes. For example,

```
(defclass reading (activity); Reading is a subclass of activity.
   ((rate                   ; Number denoting reading speed.
     :initarg :rate
     :reader rate)))
```

The predefined methods for `activity` are listed below. These methods are defined by Apex for the `activity` class and have some default behavior. Application-defined activity classes can override the default methods. Before listing the methods, two parameters that many of them take are explained.

Some of these methods take as argument a simob. This simob should be the value of the primary-object slot of the activity. There is some redundancy in this specification, but it is an artifact of an earlier design and may soon be obsolete.

Some of these methods take an optional keyword argument `:cause`, which

should be followed by an instance of the Event class. In practice one does not create this object explicitly, but rather just pass on the `:cause` argument in the calling function. This argument specifies the Event that "caused" the ensuing method call to occur and hence supports *causal tracing*, which is currently being designed and experimented with. Since causal tracing is not yet an advertised feature in Apex, this argument may be omitted or ignored.

Here are the `activity`-related methods:

`(start-activity simob activity-type &rest parameters)`
This function instantiates and starts an activity of a given type for a given simob. The activity type is the name of an activity class, and simob will become the *primary object* of the activity. What follows `activity-type` are parameters for the activity. They must be given as keyword argument/value pairs and must include at least one of either an `:update-interval` or a `:completion-time specification`.

The predefined parameters are:

`:update-interval`
Specifies the time interval (as an integer) in between activity updates, i.e. calls to update-activity.

`:completion-time`
Specifies the time (as an integer) when an activity should be completed.

`:duration`
Specifies a duration (as integral time) for an `activity`.

`:cause`
Specifies the cause of an `activity` start. See the above paragraph about this argument.

`(initialize-activity activity simob &key cause)`
This optional method is called when an `activity` starts, and provides a means to specify actions that should happen at this time.

`(update-activity activity simob &key cause)`
This method can be defined for `activity` and `simob` types to specify periodic updates to the simob resulting from the activity. For example, a rolling activity might update an object's state parameter such as location.

`(complete-activity activity simob &key cause)`
This method is called when an activity completes.

`(stop-activity activity &key cause)`
This method is used to terminate an activity without running its completion method, `complete-activity`.

`(schedule-completion activity time)`
This method is used to schedule completion of an activity in a given amount of time. Time may be given either as an integer or a time expression (see Appendix A for syntax).

5.2 Application Interface

The Lisp interface for manipulating applications is as follows:

`(initapp)` - initializes the application, which means executing the `:init` or `:init-sim` clause of its `defapplication` form.

`(startapp)` - starts the application from its initial state, or resumes execution from a paused state. This means executing the `:start` clause of its `defapplication` form.

`(stopapp)` - stops a running application (as specified in the `:stop` clause of `defapplication`).

`(stepapp)` - Advances the application by one "cycle" (as specified in the `:step` clause of `defapplication`)

`(restartapp)` - restarts the application from the beginning of its first run (applicable only to multi-run applications) (as specified in the `:restart` clause of `defapplication`)

`(reloadapp)` - reloads the current application if there is one.

These functions take no argument and return no value (they only perform side effects).

For simulations (native applications), all of these functions except `initapp` are predefined, and without `initapp` (specified as the `:init-sim` clause of `defapplication`) the simulation will have no behavior.

For non-native applications, none of these functions are predefined, and without `startapp` (specified as the `:start` clause of `defapplication`), the application will have no behavior.

Apex provides defaults for any of these functions that are not predefined or user-defined. The default behavior is to do nothing.

5.3 asamain

The function asamain invokes one cogevent-processing cycle of the ASA. It's argument is the agent whose ASA is being activated. asamain is called automatically in the following cases:

- when the keyword :trigger-asa is passed to the cogevent function
- in the assemble method of the human class
- at every cycle of the seeing activity of human.

For a non-native application, it may be necessary to call asamain explicitly, but it all depends on the nature of the application. For example, it may need to be invoked inside a control loop as done in the X-Plane® integration example. In general, it needs to be invoked whenever cogevents need to be processed, if :trigger-asa was not passed to the cogevent function.

5.4 defapplication

This form specifies the initial file set to load for the application, and code that defines the interface functions for the application (5.2). It has two formats. For native applications (simulations), it is:

```
(defapplication <name>
:libraries (<name1> .. <nameN>)
:files (<file1> .. <fileN>)
:init-sim <form>)
```

and for non-native applications:

```
(defapplication <name>
:libraries (<name1> .. <nameN>)
:files (<file1> .. <fileN>)
:init <form>
:reset <form>
:start <form>
:stop <form>
:step <form>
:restart <form>)
```

The first argument <name> is a string that names the application. The <name> arguments for the :libraries are strings naming libraries. These libraries will be loaded in the order specified. See the Libraries specifications for details about libraries. Note: this is not necessarily all of the libraries used by the application, both libraries and files can load other libraries.

The <file> arguments of the :files clause are strings representing either relative

or absolute pathnames. If the file's extension is omitted, a Lisp extension is assumed (.lisp, .lsp, or .cl). The files will be loaded in the order specified and constitute the "top level" files of the application. These files may load other files or libraries.

The `<form>` arguments of the other clauses are Lisp expressions that will be the bodies of the respective interface functions named in (5.2). When the function is called, the form is evaluated in the context of the current top-level Lisp environment (i.e. `de-fapplication` does not create a lexical scope).

All clauses are optional and can appear in any order.

5.5 Event Logging

In contrast to an activity, which represents an ongoing process, an *event* is an instantaneous occurrence in the simulation. Many events, such as activity starts, updates, completions, and in fact just about any kind of function call, are implicit in the execution of a simulation. What is useful to the modeler, however, is the tracking of *significant* events, and the current means for doing so are the following set of programming constructs.

```
(setx (slot-name simob) value :key cause agent)
```

The `setx` form changes the value of a slot and is a substitute for Lisp's `setf`. It works just like `setf` but also records the slot change as part of the simulation's history and provides a mean to specify a cause for the slot change. For example,

```
(defvar *my-book* (make-instance 'book :number-pages 430))
(setx (current-page *my-book*) 23)
```

```
(signal-event function-call &key agent)
```

The `signal-event` form is a wrapper around a function call and essentially makes an event of that function call. What is interesting is that, if the function call includes a `:cause` argument, the event created by `signal-event` is substituted for that cause as the new head of the causal chain[12]. For example, to make the turning of a page of a book a significant event, replace

```
(turn-page *my-book*)
```

with

```
(signal-event (turn-page *my-book*))
```

```
(log-event proposition &key cause agent)
```

The form `log-event` is used to record an event in a particular state, expressed as a proposition, which is represented with an arbitrary list. For example,

```
(log-event (finished-reading *my-book*))
```

In the above forms, the keyword parameter `:cause` has type *event* and is used for specifying a causal factor[12]. `The :agent` keyword parameter has type *agent* and specifies the agent that is responsible for the state change.

5.6 Pausing Simulations

Scheduled pause: A pause may be scheduled for a specified simulation clock time. The time may be specified before the simulation is run, or during a pause, if the (new) scheduled pause time is greater than the current time. Pauses are scheduled by typing the following in the Listener:

```
(set-pause-time N)
```

where `N` is an integer specification of the time in simulation time units (milliseconds by default).

Cyclic pause: A simulation pause can be scheduled to occur once every `N` simulation events (events in the simulation engine's internal activity queue). This is useful for coping with infinite loop bugs that can occur within a given simulation "moment," making it unhelpful to pause at a scheduled time that will never be reached. Such pauses are scheduled by typing the following into the Listener:

```
(set-pause-cycle N)
```

where `N` is an integer specifying the number of events.

After initialization*:* The simulation may be paused immediately after a simulation trial has been initialized using the form:

```
(pause-after-init <flag>)
```

where `<flag>` is either *T* (true) or *nil* (false). This is useful for determining whether a bug occurs before or after initialization is complete.

After each trial: The Apex simulation engine supports multi-trial simulation runs. The simulation may be paused at the completion of each trial using the form:

---

[12] A rudimentary causal tracing system has been added to Apex but is not yet documented or fully usable. Therefore, there is not much value in using the `:cause` parameter.

```
(pause-after-trial <flag>)
```

where `<flag>` is either *T* (true) or *nil* (false).

Pauses may be specified non-interactively (i.e. in code) by inserting the forms given above forms in your Lisp simworld code.

5.7 `simob`

Simob, or simulation object, is the superclass of all entities in an Apex simworld. Any classes defined by the user must inherit, directly or indirectly, from the `simob` class. For example,

```
(defclass book (simob)     ; Class BOOK is a subclass of SIMOB

  ((number-pages           ; slot for number of pages in the book
    :initarg :number-pages; name of keyword specifier for slot
    :reader number-pages) ; name of (read-only) slot accessor
   (current-page           ; slot for page being read (bookmark)
    :initform 1            ; initial value of slot
    :accessor current-page))); name of (read/write) slot accessor
```

In practice, user-defined objects will not inherit from `simob` directly, but from one of the subclasses of `simob` described in the section *Physical Environment Modeling*.

5.8 Specifying New Agent Resources

In an Apex human model, the general Action Selection Architecture does not interact with the world model directly. Instead, perceptual, cognitive, and motor resources comprising a resource architecture mediate interactions with the world and also constrain the agent to perform with human limits and other characteristics. Resources are implemented as software modules and may be replaced or modified with moderate effort[13]. This section describes how to create a new resource, e.g. a prehensile tail. Users interested in creating or modifying resources should look at examples in *apex/app/building-blocks/human*.

[13] It not always necessary to define new resources to get some of the functionality one might want. If the only need for a resource is to affect the agent's resource allocation, it is enough to simply name the resource in a profile clause (making it a virtual resource). The action selection architecture will do resource conflict detection and resolution without regard for whether that resource is associated with a class definition.

Step 1: Define the new resource type.

Every resource is implemented as Lisp class with slots representing resource state attributes. The following defines a class of resources called `tail` with a single state attribute called `grasp`. The value of this slot is a representation of an object that the tail is currently grasping – or *nil* if no such object exists.

```
(defclass tail (human-resource physob)
   ((grasp :accessor grasp :initarg :grasp :initform nil)))
```

`Tail` inherits from the classes `human-resource` and `physob` (which itself inherits from `visob` – see Appendix B), which carry along a number of state attributes. Users are encouraged to study the definitions of these objects. New resource classes associated with a particular model can be stored in a simworld definitions file.

Step 2. Redefine the class `standard-human` to include the new resource.

Human models in Apex are instances of the class `standard-human`. As there is currently no support for human models based on other classes, `standard-human` and associated functions must be modified to make use of new resource types. This class is defined in *<apex>/apexlib/human/human.lisp*. The first required modification is to the class definition itself, adding a slot named for the new resource.

```
(tail :accessor tail :initarg :tail)
```

Next modify the *assemble* method defined in the same file to include a call to the function `add-apex-resource`.

```
(add-apex-resource (make-instance 'tail) human-1)
```

In some cases, it is useful to create active resources – i.e. resources that engage in some periodic behavior rather than passively accepting commands from the Action Selection Architecture. Such behaviors can be initialized in the assemble method. For example, the following line will initiate a wiggle action every 1000 simulation time units (1 unit = 1ms by default). This assumes that the method `wiggle` has been defined and, when called, produces an appropriate effect.

```
(start-activity human-1 'wiggle :resource (tail human-1)
   :update-interval 1000)
```

Step 3: If appropriate, define activities the new resource can be commanded to carry out

Resources representing motor faculties (e.g. hands, tails) can be commanded to action by the PDL primitive action type `start-activity`. Each new kind of action (activity) is represented by a class (used to represent the state of the action at a given moment) and one or more methods defining the effect(s) of the activity. There are two kinds of

methods: `complete-activity` and `update-activity`. The former is used to describe what happens when the activity comes to completion. The latter describes what happens at intervals prior to completion. The following definitions support the activity `tail-grasp`.[14]

```
(defclass tail-grasp (resource-activity)
   ((target :accessor target :initarg :target :initform nil)))

(defmethod
   complete-activity ((act tail-grasp) (tail-1 tail)&key cause)
   (signal-event
      (grasped (target act) (setting act) :cause cause))

(defmethod grasped ((obj physob) (tail-1 tail) &key cause)
   (setx (grasp tail-1) obj :cause cause))
```

A step from such as the following can be used to invoke this behavior from PDL:

```
(step s1 (start-activity tail tail-grasp :target banana
   :duration 2500))
```

Step 4: If appropriate, define event-generating processes invoked by the resource.

Some resources, particularly those modeling perception, generate input to the Action Selection Architecture called *cogevents*. This is accomplished using the function `cogevent`

```
(cogevent <eventform> <agent> [:trigger-asa <Boolean>])
```

where <eventform> is an arbitrary list representing what has occurred and <agent> is a pointer to the (human) agent that has detected the event. The optional trigger parameter determines whether the event should be processed by the Action Selection Architecture immediately or whether it should be stored in a buffer and processed the next time a processing cycle for the architecture occurs.[15]

[14] Signal-event and `setx` are special forms used to track causal dependencies in a simulation. The former should be wrapped around a function or method call implementing a change of state to one or more simulated objects. `setx` should be used to effect the state change as it were `setf`.

[15] There is no automatic architecture cycle; it must be triggered by a resource. In the default model, the vision resource triggers processing periodically.

References

Freed, M. (2000) Reactive Prioritization. In *Proceedings 2nd NASA International Workshop on Planning and Scheduling for Space*. San Francisco, CA.

Freed, M. (1998a) Simulating Human Behavior in Complex, Dynamic Environments. Doctoral Dissertation. Department of Computer Science, Northwestern University.

Freed, M. (1998b) Managing multiple tasks in complex, dynamic environments. In *Proceedings of the 1998 National Conference on Artificial Intelligence*. Madison, Wisconsin.

Freed, M. and Remington, R. (2000a) GOMS, GOMS+ and PDL. In *Working Notes of the AAAI Fall Symposium on Simulating Human Agents*. Falmouth, Massachusetts.

Freed, M. and Remington, R. (2000b) Making Human-Machine System Simulation a Practical Engineering Tool: An APEX Overview. In *Proceedings of the 2000 International Conference on Cognitive Modeling*. Groningen, Holland.

Freed, M. and Remington, R. (1998) A conceptual framework for predicting errors in complex human-machine environments. In *Proceedings of the 1998 Meeting of The Cognitive Science Society.* Madison, Wisconsin.

Freed, M. and Remington, R. (1997*)* Managing decision resources in plan execution. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*. Nagoya, Japan.

Freed, M. and Shafto, M. (1997) Human System Modeling: Some Principles and a Pragmatic Approach. In *Proceedings of the 4th International Workshop on the Design, Specification, and Verification of Interactive Systems*. Granada, Spain.

Freed, M., Shafto, M., and Remington, R. (1998) Using simulation to evaluate designs: The APEX approach. In Chatty, S. and Dewan, P., editors, *Engineering for Human-Computer Interaction*, chapter 12. Kluwer Academic.

John, B. E., Vera, A. H., Matessa, M., Freed, M., and Remington, R. (2002) Automating CPM-GOMS. In *Proceedings of CHI'02: Conference on Human Factors in Computing Systems*. ACM, New York, pp. 147-154.

Glossary

ASA - Action Selection Architecture (ASA) is the algorithm 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. The architecture outputs commands to resources.

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

Emacs - Emacs is a text editor and software development environment with support for Lisp programming.

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

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

PDL - Procedure Description Language (PDL) is a formal language used to specify the behavior of Apex agents.

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

RAP - RAP is a plan and task representation based on program-like reactive action packages (RAP).

Appendix A: Event Traces

A.1 Predefined Show-Levels

all	: all events
none	: no events
default	: only task-started events
actions	: resource related events
asa-low	: Action Selection Architecture event, low detail
asa-medium	: Action Selection Architecture event, medium detail
asa-high	: Action Selection Architecture event, high detail
cogevents	: cognitive events
simulation	: activity related events

A.2 Lisp Commands for Controlling Trace Output

(show)	: query the current TraceConstraint (syntax on next page)
(show :runtime)	: see event trace as simulation runs (useful for debugging)
(show :hms)	: see time displayed in hours/mins/secs
(show :level *level*)	: affects the given ShowLevel (see levels list)
(show *EventType*)	: adds event type to trace (see event types list)
(show Constraint)	: adds events matching given TraceConstraint to trace
(unshow)	: turns off event trace
(unshow :runtime)	: suppress runtime display of event trace
(unshow :hms)	: see time displayed as an integer
(unshow *EventType*)	: removes event type from trace (see event types list)
(unshow Constraint)	: removes events matching given TraceConstraint from trace
(generate-trace)	: generate and print the trace
(trace-size)	: query number of events in latest trace

(define-show-level *name TraceConstraint*) : defines show level (*name* is symbol)

A.3 Trace Constraint Syntax

TraceConstraint:

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

TraceParameter :

(event-type <symbol>)	{ matches events of given type }
| (object-id <symbol>)	{ matches events containing given object }
| (time-range (<low> <high>))	{ matches events occurring in given time range }

TimeExpression : (TimePair+) { one or more int/unit pairs }

TimePair : (<integer> TimeUnit)

TimeUnit : ms | msec | msecs
| s | sec | secs | second | seconds
| m | min | mins | minute | minutes
| d | day | days

A.4 Event Types

Each event type is explicitly logged and can be filtered in/out for trace view. Verbose event descriptions name event parameters not including timestamp.

Causal event 0 is the *initialize* event. Terminology changes: *enabled* refers to satisfaction of non-resource preconditions – any resource preconditions not yet satisfied; *executed* tasks must take 0 time – i.e. primitive and special (Lisp callout) tasks; *started* is used for non-primitives. Resource deallocation events occur when a task is terminated or interrupted.

Need to review this list for meaningful and consistent naming, completeness/usefulness of causal information.

Table A.4.1 Action Selection Architecture Events

	Event type	Description (not including time)	Causal events
1	task-created	<task>	0,17
2	monitor-created	<monitor> <task>	0,17
3	monitor-satisfied	<monitor> <cogevent>	2+any
4	{monitor-tentatively-satisfied}	<monitor> <cogevent>	2+any
5	{monitor-expired}	<monitor>	2+time
6	{monitor-desatisfied}	<monitor> <cogevent>	2+any
7	enablement-testing-started	<task>	3
8	enabled*	<task>	1+7+3*
9	refused-enablement*	<task>	1+7
10	procedure-selected	<task> => <procedure>	8+10
11	conflict-detected	<task> <task> <resource>	10,12
12	conflict-resolved*	: winner <task> :loser <task>	11,13/13
13	priority-computed	for <task> = <priority>	11,15
14	resource-allocated*	<task> <resources>	11+12,16
15	interrupted*	<task> <task>	8+12
16	resource-deallocated*	<resource> :from <task>	15,20
17	task-started*	<task>	8+14
18	executed*	<task>	8+14
19	resumed*	<task>	15+14
20	terminated*	<task>	18,20
21	reset*	<task>	18
22	reinstantiated*	<task>	8,17,20
23	assumption-violated	<varname> <agent>	3

{} - ASA actions that are not yet supported
* - an associated cogevent is generated

Table A.4.2 Resource Architecture Events

	Event type	Description (not including time)	Causal events
Control			
1	started*	<activity> <parameters>*	
2	completed*	<activity>	
3	stopped*	<activity>	
4	clobbered	<activity> :by <activity>	
Vision			
1	nothing-new*	vision	
2	pos*	<visobfile> <coordinates>	
3	Color*	<visobfile> <colorname>	
4	orientation*	<visobfile> <degrees>	
5	Shape*	<visobfile> <shapelist|shape>	
6	contrast*	<visobfile> <value>	
7	Blink*	<visobfile> <rate>	
8	elements*	<visobfile> <list>	
9	contains*	<visobfile> <vof-list>	
10	contained-by*	<visobfile> <visobfile>	
Gaze			
1	fixated*	<visobfile>	
2	winnowed*	<visobfile> <feature>	
3	held-gaze*	<locus> <time>	
Memory			
1	encoded*	<proposition>	
2	retrieved*	<proposition>	
3	new*	<proposition>	
4	revalued*	<proposition>	
5	refreshed*	<proposition>	
6	refined*	<proposition>	
Hands			
1	grasped*	<hand> <object>	
2	released*	<hand> <object>	
3	moved*	<hand> <object>	
4	turned-dial*	<hand> <dial> <position>	
5	Typed*	<hand> <keyboard> <msg>	

* - an associated cogevent is generated

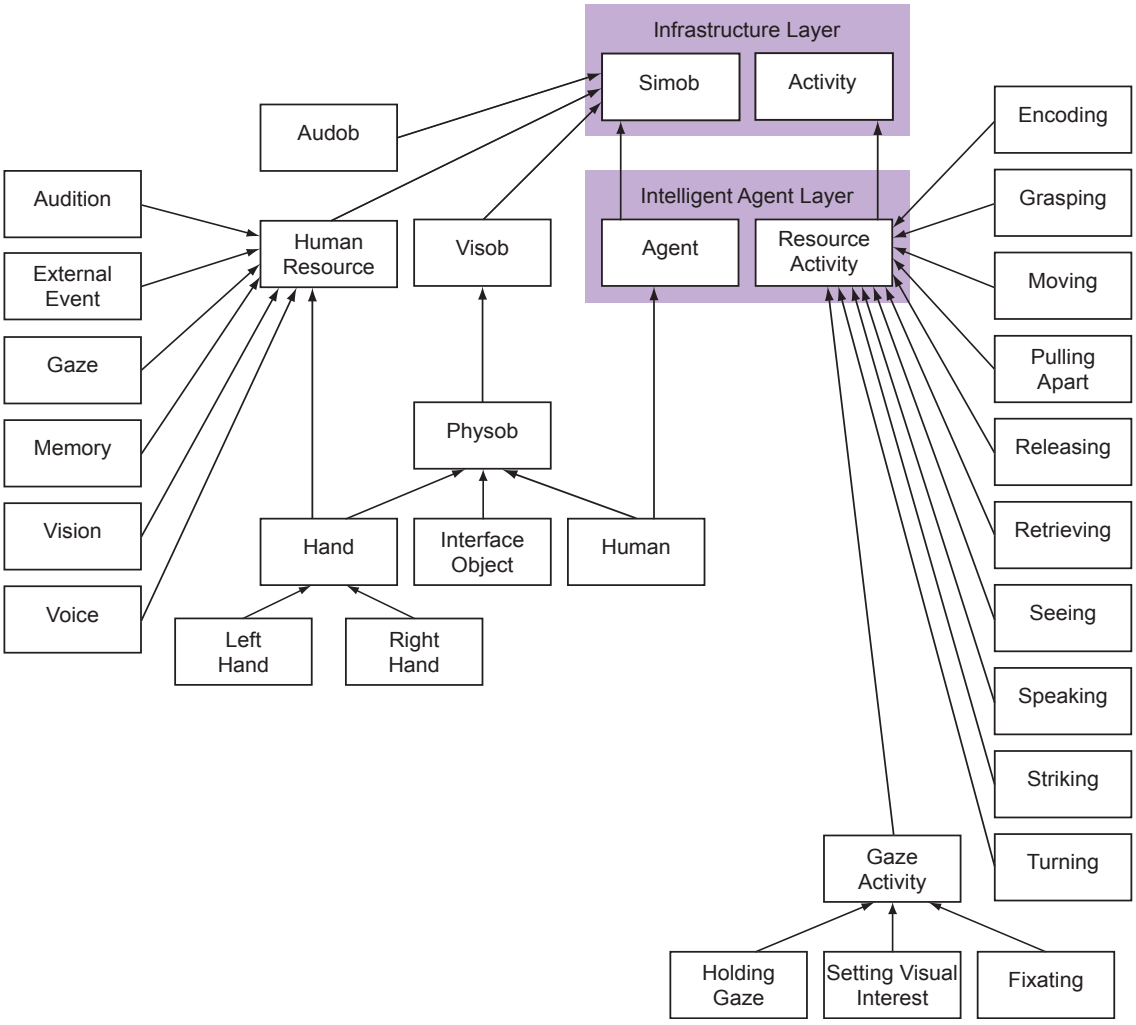
Table A.4.3 General Simulation Events

	Event type	Description (not including time)	Causal events
1	started-activity	<activity> <primary-simob>	
2	initialized-activity	<activity> <primary-simob>	
3	updated-activity	<activity> <primary-simob>	
4	stopped-activity	<activity> <primary-simob>	
5	completed-activity	<activity> <primary-simob>	

Appendix B: Apex Library

6/6/2003 apex - library.vthought

Classes comprising the provided library (found in apex/apexlib), are shown here, along with their super-classes in the Apex system.



Appendix C: Troubleshooting

C.1 Common Problems

This section contains possible solutions to some of the problems users have reported.

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

Explanations/Solutions:

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

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

Explanations/solutions:

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

C.2 Known Bugs

Note: bugs associated with specific Apex processes or PDL constructs are listed in the appropriate section.

Apex can crash if an agent acts in reference to a world object at time 0. The reason is that the behavior might be initiated before the world object is specified and incorporated into the physical environment model. Avoid this problem by insuring that the assemble method is called on all physical environment objects before any agent objects are initialized.

The read macro `#L` that forces a Lisp evaluation at create time does not work in primitive (directly executable) procedure steps.

Activities can be started with negative duration values. This should produce an error.

Appendix D: Pattern Matching

Pattern matching is used in a variety of PDL constructs including `index`, `waitfor`, and `step`. These examples illustrate the behavior and capabilities of the pattern-matching algorithm. Source: *Paradigms of AI Programming* by Peter Norvig (1991).

```
(pat-match '(x = (?is ?n numberp)) '(x = 34))
;;;; -> ((?n . 34))

(pat-match '(x = (?is ?n numberp)) '(x = x))
;;;; -> NIL

(pat-match '(?x (?or < = >) ?y) '(3 < 4))
;;;; -> ((?Y . 4) (?X . 3))

(pat-match '(x = (?and (?is ?n numberp) (?is ?n oddp))) '(x = 3))
;;;; -> ((?N . 3))

(pat-match '(?x /= (?not ?x)) '(3 /= 4))
;;;; -> ((?X . 3))

(pat-match '(?x > ?y (?if (> ?x ?y))) '(4 > 3))
;;;; -> ((?Y . 3) (?X . 4))

(pat-match '(a (?* ?x) d) '(a b c d))
;;;; -> ((?X B C))

(pat-match '(a (?* ?x) (?* ?y) d) '(a b c d))
;;;; -> ((?Y B C) (?X))

(pat-match '(a (?* ?x) (?* ?y) ?x ?y) '(a b c d (b c) (d)))
;;;; -> ((?Y D) (?X B C))

(pat-match '(?x ?op ?y is ?z (?if (eql (?op ?x ?y) ?z))) '(3 + 4 is 7))
;;;; -> ((?Z . 7) (?Y . 4) (?OP . +) (?X . 3))

(pat-match '(?x ?op ?y (?if (?op ?x ?y))) '(3 > 4))
;;;; -> NIL

(pat-match-abbrev '?x* '(?* ?x))
;;;; -> (?* ?X)

(pat-match-abbrev '?y* '(?* ?y))
;;;; -> (?* ?Y)

(setf axyd (expand-pat-match-abbrev '(a ?x* ?y* d)))
;;;; -> (A (?* ?X) (?* ?Y) D)

(pat-match axyd '(a b c d))
;;;; -> ((?Y B C) (?X))

(pat-match '(((?* ?x) (?* ?y)) ?x ?y) '((a b c d) (a b) (c d)))
;;;; -> NIL
```

Appendix E: Application Definition File Example

The following is an example of a well-formed Application Definition File. It is included with the distribution of Apex along with others in the examples directory.

```
;;; Hello World
;;;
;;; This is a trivial simulation designed to exemplify some basicfeatures of Apex and
;;; PDL programming. It consists of a telephone, which is initially silent, but rings
;;; after some time. Upon detecting the phone ring, the simulated humananswers it.
;;;
;;; ---- Application header

;;; Every Apex application file (including libraries) must contain the header. The
;;; value of :version is the global variable *apex-version*.

(apex-info :version "2.4")

;;; ---- Application definition

;;; The top level file of an application (e.g. this file) is called the Application
;;; Definition File and must containa defapplication form. It may be placed
;;; anywhere in the file.

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

;;; ---- Libraries

;;; One can specify needed libraries in defapplication (using the:libraries
;;; clause), but since we are including application code in the same file, we must
;;; explicitly load any librariesneeded by the code...

(require-apex-library "human")


;;; ---- Objects

;;; This scenario has just one object, a highly simplified telephone. A telephone is
;;; a kind of physicalobject (physob).

(defclass telephone (physob)
  ((state
    :type symbol   ; possible values: silent, ringing, engaged
    :initform 'silent
    :accessor state)
   (state-start-time :type number :initform 0 :accessor state-start-time)))


;;; ---- Activities

;;; 1. "Being" a Phone

;;; We are modeling the telephone as a passive object that simply"waits" until
;;; something happens to it, in our case being it starts ringing (without any
;;; modeled cause). The activity of "being" models this passive state.
```

```
(defclass being (activity) ())

;;; The (simulation) activity of "being a telephone" never ends.It is updated
;;; at regular intervals, at which time if the phone is silent it may randomly
;;; start ringing. Once the phone starts ringing, it does so forever in
;;; this simplified model.
;;;
;;; NOTE the call to cogevent. It implies that the telephone istelepathically
;;; informing the (sole) agentthat it is ringing!This is an unfortunate
;;; ramification of the lack of a communications framework for Apex
;;; agents. The Apex team is working on an elegant solution.

(defmethod update-activity ((act being) (tel telephone))
  (when (and (eq 'silent (state tel)) (fifty-fifty-chance))
    (setx (state tel) 'ringing)
    (cogevent `(ringing ,tel) *agent* :trigger-asa t)))

;;; Physical objects (physobs) require "assembly". The assemble method is a
;;; convenient means for combining many objects into one (not applicable in
;;; this case) and starting initial activities.

(defmethod assemble ((tel telephone) &key component-of)
  (start-activity tel 'being :update-interval 10))

(defun fifty-fifty-chance () ; Support function for update-activity.
  (= 0 (random 2)))

;;; 2. Answering the phone

;;; For simplicity we'll model picking up the phone as a directactivity. It is
;;; a kind of resource activity because it requires a resource (hand) of a
;;; simulated human.

(defclass picking-up-phone (resource-activity)
  ((phone :initarg :phone :reader phone)))

;;; The activity of picking up the phone has no interesting behavior other
;;; than to complete (after the duration specified in its start-activity). At
;;; this time, the phone is "placed" into the hand, it's state becomes
;;; engaged, and an appropriate events (setx) are generated.

(defmethod complete-activity ((act picking-up-phone)(hand hand))
  (let ((phone (phone act)))
    (setx (grasped-object hand) phone)
    (setx (state phone) 'engaged)))

;;; 3. Saying hello

;;; For this we'll used the SPEAKING activity defined in the Human library.
;;; We'll just specialize its completion method to create a an appropriate
;;; event.

(defmethod complete-activity :after
                              ((act speaking) (v voice))
   (log-event `(said ,(utterance act)) :agent *agent*))


;;; Procedures
```

```
;;; The top level goal for this scenario is simply to answer the phone when it
;;; starts ringing. Note the WAITFOR: it is what binds the variable ?phone to
;;; phone object when its given proposition (a cogevent) is detected.

(procedure
 (index (handle-phone))
 (step s1 (answer-phone ?phone) (waitfor (ringing ?phone)))
 (step s2 (end-trial) (waitfor ?s1)))

;;; To answer the phone, you pick it up and say hello...

(procedure
   (index (answer-phone ?phone))
   (step pickup (pickup-phone ?phone))
   (step talk (say-hello) (waitfor ?pickup))
   (step stop (terminate) (waitfor ?talk)))

;;; The phone is answered by starting the picking-up-phone activity (the
;;; chosen duration is arbitrary). This procedure completes when that activity
;;; completes.

(procedure
   (index (pickup-phone ?phone))
   (profile right-hand)
   (step pickup (start-activity right-hand picking-up-phone :phone ?phone
                                :duration (1 sec) => ?act))
   (step terminate (terminate) (waitfor (completed ?act))))

;;; Speech is uttered by starting the speaking activity (the chosen duration
;;; is arbitrary). This procedure completes when that activity completes.

(procedure
   (index (say-hello))
   (profile voice)
   (step talk (start-activity voice speaking :utterance "Hello?"
                              :duration (500 ms) => ?act))
   (step terminate (terminate) (waitfor (completed ?act))))


;;; ---- Initialization

;;; The essential elements of simulation initialization are creating objects
;;; and starting at least one activity. In this case the assemble methods do
;;; the latter (note that these initial "time 0"activities are started before
;;; the simulation starts, technically speaking). Finally, a few useful event
;;;  types are enabled with SHOW.

(defun hello-world ()
  (let* ((room (make-instance 'locale :name "Living Room"))
         (phone (make-instance 'telephone :name "Jill's Phone" :locale room))
         (jill (make-instance 'human :name "Jill" :locale room
                         :initial-task '(handle-phone))))
    (assemble phone)
    (assemble jill)
    (show state)
    (show said)
    (show completed)))

;;; End of file
```

Appendix F: Starting Apex within Allegro Common Lisp

If you have the Franz Allegro Common Lisp (ACL) development environment and wish to use it to run Apex instead of running Apex from its distribution, then please follow these steps.

1. **Start Emacs** (if you use Emacs).

2. **Start ACL**:
   Within Emacs (assuming you have the ACL Emacs interface),
   enter "`M-x fi:common-lisp`" to start ACL.

   Without Emacs, start the 'alisp' executable of ACL. There are several others.

3. **Load Apex**: Enter `load "... apex/load.lisp`", where "..." is your path.

4. **Start the Sherpa server**: Enter `sherpa.` Sherpa is started as a separate process, so that the Apex listener can still be used.

5. **Launch Sherpa**:
   On a Windows or Macintosh system, double click on the Sherpa icon.
   On Linux or Solaris, type "`java -jar sherpa.jar`" in the directory that contains sherpa.jar (or whatever the Sherpa file is called).

Index

U

V

W

X