Working Document:
Revisions as of September 25, 1997

The Act-Editor User's Guide

A Manual for Version 2.2

Karen L. Myers David E. Wilkins

Artificial Intelligence Center
SRI International
333 Ravenswood Ave.
Menlo Park, California 94025

The writing of this guide was supported by SRI International and by DARPA/Rome Laboratory Contracts F30602-90-C-0086 and F30602-95-C-0235.

Preface

The Act representation language provides a medium in which to express knowledge about actions for both the SIPE–2 plan generation system and the PRS-CL plan execution system. This document describes the Act-Editor, which provides a graphical user interface for creating and manipulating Acts. The document is designed for individuals who are already familiar with the Act formalism [10]. The Act-Editor runs on Sun workstations with either Allegro Common Lisp 4.2/4.3 with CLIM 2.0/2.1, or Lucid Lisp 4.1 with CLIM 1.1, as well as Symbolics Lisp Machines.

Contents

List of Figures

Chapter 1

Introduction

Many domains in which AI planning techniques can be profitably employed are dynamic in nature. For example, military operations planning and controlling a mobile robot both exhibit this characteristic: during either plan execution or plan generation, the state of the world can change dramatically as troops are dispatched to an area or a robot navigates through a hallway. For such domains, it is necessary that plan generation systems be sensitive to run-time concerns and that plan execution systems be capable of invoking the plan generator to address unexpected events at run-time.

Traditionally, plan generation and reactive execution have been considered as separate activities, with few attempts to integrate them within a single system. This dichotomy has led to the development of planning systems that ignore execution issues and reactive plan execution systems that cannot synthesize new plans at run-time. Not surprisingly, different representations have developed for defining operators for synthesizing and executing plans, making it difficult to support tightly integrated plan generation and plan execution systems. Such integration is difficult, given that generation and execution involve different kinds of knowledge and reasoning capabilities.

SRI International has developed the Act formalism [10] for representing the knowledge required to support both the generation of complex plans and reactive execution of those plans in dynamic environments. Act is intended to serve as a general-purpose representation language that could be used to share knowledge between many different execution and planning systems. A design goal of Act was its adequacy for practical applications and domain independence. The basic unit of representation is an *Act*, which can be used to encode plans, plan fragments, standard operating procedures (SOPs), and task networks for HTN (hierarchical task network) planning systems. An Act describes a set of actions that can be taken to fulfill some designated purpose under certain conditions.

The Act-Editor's home page, including the specification of the Act formalism, can be found at
`http://www.ai.sri.com/~act`

1.1 Implementations Using Act

An important design goal of Act was to develop a system that will be useful in practical applications – including those requiring complex plans with parallel actions that are interrelated. To achieve this goal, the development of Act was driven by the need for an interlingua that links a previously implemented planner (SIPE–2 [8, 9]) with a previously implemented executor (PRS-CL [2, 3]). Both of these systems have numerous practical applications, including robot control and military operations, thus attesting to Act's expressive and computational adequacy.

The Act formalism was used as the common representation in the Cypress system [11]. Cypress is based on versions of SIPE–2 and PRS-CL extended to support Act.[1] Cypress provides a framework in which to create taskable, reactive agents and supports the generation and execution of complex plans with parallel actions, the integration of goal-driven and event-driven activities during execution, and the use of replanning to handle run-time execution problems. In Cypress, plans generated by the planner and represented as Acts are sent to the executor for execution. Both the planner and executor use knowledge represented as Acts to perform, respectively, their planning and control roles.

Because of the Cypress implementation, several links exist between the Act-Editor and other systems. Files of Acts can be loaded into a PRS-CL agent, using the *Load* or *Append* commands in the Knowledge menu of PRS-CL. Similarly, Acts and files of Acts can be translated into SIPE–2 input files (by using the -> *SIPE Act* and -> *SIPE Graph* commands in the Act-Editor). These commands invoke translators that automatically map Acts onto SIPE–2 operators and PRS-CL *Knowledge Areas* (or, *KAs*).

There is also a translator to produce Acts from existing SIPE–2 operators and plans, although the Act-Editor is now the preferred tool for writing SIPE–2 operators. In SIPE–2, the Domain menu has commands for translating operators: the -> *ACT* command translates selected operators, while the -> *ACT all* command converts all operators and allows several options, including automatic creation of a dictionary for the operators (see Chapter 4). The Plan menu has commands for translating plans that have been generated: the -> *ACT* command translates the current plan into an Act, and the *ACT Options* command allows translation of several plans with several options. Finally, SIPE–2 has a -> *PRS* command that translates the current plan to an Act and loads it into a selected PRS-CL agent.[2]

As described in Chapter 4, SIPE–2 can also copy its existing sort hierarchy into an Act-Editor dictionary, and another SRI system, the GKB-Editor can be used to interactively edit and create a SIPE–2 sort-hierarchy.

[1] In particular, Cypress = SIPE + PRS.

[2] This command is active only when SRI's Cypress system is loaded.

1.2 The Act-Editor

The Act-Editor is a graphical user interface for creating, displaying, and manipulating Acts. This document describes the system and how it can be used to create Acts for either PRS-CL or SIPE–2 applications, or both. The document presumes some familiarity with the Act formalism, described in [10]. The Act-Editor provides a *Verifier* that can be used to verify that Acts have correct syntax, and that they use predicates and functions as specified by the dictionaries in use (see Chapter 4).

The Act-Editor system runs on Sun[3] workstations as well as Symbolics Lisp Machines. This manual focuses on the use of the Act-Editor on Sun workstations. The software runs on Sun workstations with either Allegro CL 4.3 and CLIM 2.0 or later, or Lucid Lisp 4.1 and CLIM 1.1.[4]

Someone attempting to model a new domain in Act will be referred to as a *programmer*. Programmers should be familiar with the representational ideas underlying Act [10]. Certain sections of this document labeled as *advanced concepts* can be skipped by a user; they are intended for the programmer and assume in-depth knowledge.

1.3 Overview

This chapter concludes with a short introduction to the basics of Grasper-CL. Chapter 2 then presents a brief overview of Act and a discussion of how Acts are represented in Grasper-CL. Details of how to use the Act-Editor to generate Acts are presented in Chapter 3, including directions for loading and initializing the system, an overview of the graphical user interface, and a sample editing session. The next five chapters describe the commands, customizations, and dictionaries. The final chapter offers guidance on trouble-shooting problems that might arise during use of the Act-Editor. Finally, an appendix explains how to create Acts programmatically. The Act formalism, including a grammar, is described in another document [6].

Acts can be stored either as graphs in Grasper-CL, or as text files using the ASCII representation [6]. In the Act-Editor, a *graph* is nothing more than a file that is comprised on any number of Acts, where the Acts are represented in Grasper-CL's language, which includes information about how to draw them on the screen. The Act-Editor manipulates graphs, and builds in some of the syntactic restrictions. For this reason, it is the preferred mechanism for constructing Acts. Commands are provided for writing graphs as ASCII files and for reading ASCII files into graphs. ASCII files do not include information about how to draw the Acts on the screen, and are useful for sending Acts to other systems that do not include Grasper-CL.

[3] All product and company names mentioned in this document are the trademarks of their respective holders.

[4] It would not be difficult to port the system to environments supporting Common Lisp and CLIM.

1.4 Basic Concepts of Grasper-CL in the Act-Editor

The Act-Editor is layered on top of a graphical editing tool called Grasper-CL.[5] Grasper-CL [4, 5] is a COMMON LISP system developed at SRI for displaying and manipulating graphs. Grasper-CL supports interactive graph editing and provides a foundation for constructing graphical user interfaces (GUIs) for application programs whose inputs or results can be presented as a graph. The Act-Editor constitutes one such application, while SIPE–2 and PRS-CL are others.

Grasper-CL consists of several different components: a core function library for manipulating graph data structures, a graph-display module for producing drawings of graphs, an interactive graph editor, and a suite of automatic graph-layout algorithms. Displays are created using the COMMON LISP Interface Manager (CLIM) — a high-level, platform-independent graphics system that runs on Sun workstations, Macintoshes, IBM PCs, and Symbolics workstations.

The Act-Editor operates on Acts represented in Grasper-CL (as opposed to the ASCII representation) because this representation can support a sophisticated GUI. (The Act-Editor can convert ASCII Acts to the Grasper representation.) To understand the workings of the Act-Editor, a certain basic knowledge of Grasper-CL is required. The basic concepts that underlie Grasper-CL apply also to the Act-Editor, as described in detail in Section 2.2. One important consequence of this architecture is that a user can employ Grasper-CL as well as the Act-Editor to manipulate and define Acts.

The primitive building blocks in Grasper-CL are *nodes* and *edges*. In the Act-Editor, the various components of an individual Act are represented as nodes and edges. A node is displayed as a combination of an *icon* and a *label*. The label is simply a string to be printed, and for most Act nodes is composed of the Act formulae for the node. The icon is a shape used in displaying the node, such as a rectangle or an ellipse. Icons can be invisible; in such a case, only the label is displayed. Edges connect nodes, and are also characterized by icons and labels.

Nodes and edges can be grouped into a data structure called a *space*. In the Act-Editor, each Act is represented in its own space — the name of the Act is the name of the space. Acts correspond to a restricted subclass of Grasper-CL spaces that are constrained to have certain characteristics. The Act-Editor also supports dictionaries, which are grasper graphs that contain a specified set of non-Act spaces for representing dictionary data, as described in Chapter 4.

In turn, spaces can be grouped into a structure called a *graph*. A graph is primarily a file that is associated with the set of Acts in the graph (graph-wide properties can also be stored with a graph). The filename is displayed in the status line of the GUI. Graphs can be saved to disk using the Graph menu of either Grasper-CL or the Act-Editor, and various other commands can be used to perform operations (e.g., replacing or finding a symbol) on all the spaces in one graph. Graphs should be used

[5] The name Grasper-CL will be used throughout this manual to refer to the most recent Grasper system available. Currently, this is Grasper-CL on Sun workstations and Grasper II on Symbolics workstations.

to group appropriate sets of Acts together. In particular, all Acts in a single graph are loaded into a given planning or execution system. For example, if some Acts are to be used in some applications but not others, they should be in their own graph allowing that graph to be loaded only when it is appropriate. Generally, a particular application of a reasoning system will load several graphs.

At any given time, the display module designates a single graph as the *current graph*. Similarly, a single space within the current graph is marked as the *current space*. All interactions with Grasper-CL result in modifications to the current space and current graph. Similarly, most commands in the Act-Editorare invoked on the current space and/or graph. Grasper-CL always displays the current space, barring errors and anomalous situations.

Further documentation is available for users who are interested in learning more about Grasper-CL. The *Grasper-CL User's Guide* (in [4]) provides a thorough overview of the foundations and use of Grasper-CL. The *Grasper-CL Programmer's Manual* (also in [4]) provides descriptions of the COMMON LISP application programmer's interface.

Chapter 2

Acts and their Representation

The Act formalism is briefly summarized here. In addition, Act structures and their Grasper-CL representations are described.

2.1 Act Structures

The basic unit of representation is an *Act*, which describes a set of actions that can be taken to fulfill some designated purpose under certain conditions. The purpose could be either to satisfy a goal or to respond to some event in the world. The purpose and applicability criteria for an Act are formulated using a fixed set of *environment conditions*. Action specifications are called the *plot*, and consist of a partially ordered set of actions and subgoals.

The environment conditions and plots are specified using *goal expressions*, each of which consists of one of a predefined set of *metapredicates* applied to a logical formula. The metapredicates permit the specification of many different modes of activity, including goals of achievement, maintenance, and testing.

Figure 2.1 is an example Act taken from the screen of the Act-Editor. This act was originally written as a SIPE–2 operator in the military domain and was translated by SRI's SIPE-to-ACT translator. The environment conditions are displayed on the left side of the screen and the plot nodes on the right side. This Act describes an operator for deploying an air force to a particular location. The Cue is used to invoke the Act when the system has the goal of achieving such a deployment. The Precondition enforces various constraints on the intermediate locations to be used in the deployment. The Setting essentially looks up the cargo that must go by air and sea for this deployment. The plot is described in Section 2.1.4.

It is important to remember that the term *graph* is never used to refer to a plot, such as the one drawn in Figure 2.1. *Graph* always refers to a group of Acts (or dictionary spaces) associated with a

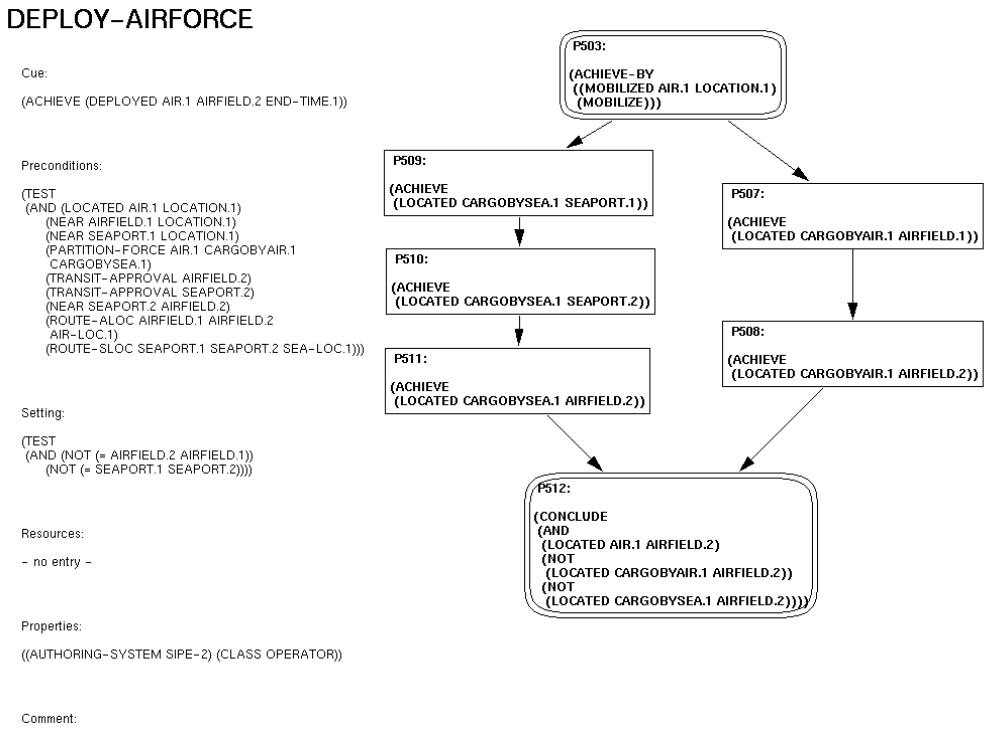

Figure 2.1: Deploy Airforce Act

file.

2.1.1 Goal Expressions

Goal expressions describe requirements on the planning/execution process and desired states to be reached. They consist of an Act metapredicate applied to a logical formula built from predicates specified in first-order logic, connectives, and the names of Acts. The predicates describe possible goals and beliefs of the system. Goal expressions are used to specify both applicability conditions for environment conditions and subgoals for plot nodes.

2.1.2 Metapredicates

Act metapredicates describe conditions on current goals and facts, and are used to specify applicability conditions in environment conditions and actions in plot nodes. Table 2.1 lists the metapredicates and hints at the syntax of each, with *wff* standing for a formula composed of first-order predicates and connectives. The syntax is described fully elsewhere [6]. If the structure-based editor is used (see Section 9.1), the user does not need to know the detailed syntax since the Act-Editor prompts for lower-level formulas or names and combines them into the correct syntax.

Metapredicate	Arguments
Test	(wff)
Use-Resource	(object ...)
Achieve	(wff)
Achieve-by	(wff (act_1 ... act_k))
Achieve-all	(wff_1 wff_2)
Wait-Until	(wff)
Require-Until	(wff_1 wff_2)
Conclude	(wff)

Table 2.1: Act Metapredicates

Environment Slot	Role
Name	identifier
Cue	used to efficiently retrieve this Act
Precondition	gating conditions on applicability of this Act
Setting	queries world to bind local variables
Resources	resource constraints
Properties	user-defined attributes, temporal constraints
Comment	documentation

Table 2.2: Environment Conditions and Their Roles

The *Test* metapredicate specifies a formula whose truth value must be ascertained. The *Use-Resource* metapredicate makes a declaration of resources that will be used by the Act, and hence that must be available for an Act to be applied. The *Require-Until* metapredicate designates conditions that must be maintained over a specified interval. The *Conclude* metapredicate designates information about changes in the world caused by an action.

The term *action metapredicates* refers to the metapredicates *Achieve*, *Achieve-by*, *Achieve-all*, and *Wait-Until*. Achieve directs the system to accomplish a goal by any means possible; Achieve-By is similar but specifies a restricted set of Acts that can be used to accomplish the task. Achieve-All directs the system to accomplish a set of Achieve metapredicates in parallel for all objects that satisfy a given pattern predicate. Wait-Until directs the system to wait until some specified condition holds.

2.1.3 Act Environment Conditions

The Act environment conditions are defined as a series of fixed *slots*, shown in Table 2.2. Name and Comment are straightforward; the former is a unique identifier for the Act, and the latter is a string that provides documentation. The slots Cue, Precondition, Setting, and Resources are referred to as the *gating slots* for an Act because they specify conditions that must be satisfied for the Act to be applicable in a given situation. The gating slots are filled with one or more goal expressions [10].

Gating Slot	Metapredicates
Cue	Achieve, Test, Conclude
Precondition	Achieve, Test
Setting	Test
Resources	Use-Resource

Table 2.3: Metapredicates Allowed in Gating Slots

The Properties slot is a list of property/value pairs for the Act. Properties are used for several purposes: to provide documentation, to represent information specific to a particular application or planning/execution system, and to represent knowledge that is not directly supported in Act, generally because it is needed by either the planner or the executor, but not by both. For example, SIPE–2 recognizes the property `Variables` for quantifying variables as existential or universal, and PRS-CL uses the property `Decision-Procedure` to designate an Act that is used for metalevel reasoning. In addition, certain properties are used to designate information to the translators; for example, the Act-to-SIPE translator looks for information about operator *types* to determine how the Act should be translated. The user is free to supply additional properties, as desired.

The most interesting property from a representational standpoint is `Time-Constraints`, which allows specification of any of the 13 Allen relations [1]. This property is used to specify time constraints between plot nodes that cannot be represented by ordering arcs, e.g., two actions must end at the same time. The Act-Editor can also represent these constraints as labeled arcs of different color between plot nodes. There are no required properties, although some are recommended for documentation purposes (such as `Author`).

Table 2.3 displays the metapredicates allowed in each of the gating slots. The Act-Editor enforces these restrictions during editing.

2.1.4 Plots

The plot specifies the activities for accomplishing the purpose of an Act. The plot consists of a directed graph, whose nodes represent actions and whose arcs impose a partial order for execution. (Any temporal relationship between two nodes can be represented using the Time-Constraint property.) Associated with each plot node is a list of goal expressions for the node.

Metapredicates in Plot Nodes

All Act metapredicates are allowed on plot nodes. However, at most one goal expression on each node can be built using the same metapredicate. Furthermore, the action metapredicates (Achieve, Achieve-by, Achieve-all, and Wait-Until) are mutually exclusive: if one is used on a node, the others are prohibited. Empty plot nodes are allowed.

Metapredicate group	Metapredicates
Context	Test, Use-Resource
Action	Achieve, Achieve-by, Achieve-all, Wait-Until
Effects	Require-Until, Conclude

Table 2.4: Metapredicate Grouping for Execution in Plot

For purposes of ordering their execution, the metapredicates are partitioned into three groups, as shown in Table 2.4. The Context metapredicates, Test and Use-Resource, are executed first by the execution system. During planning, Use-Resource makes a declaration that will be used by the planner. The Action metapredicate for the node, if there is one, is executed next. The Effects metapredicates, Require-Until and Conclude, are executed last. Require-Until sets up a protection interval that must be maintained, and Conclude specifies any effects to be added to the system database.

Plot Topologies

A plot has a single *start node* (a node with no incoming arcs) but may have multiple *terminal nodes* (a node with no outgoing arcs). Loops can be specified by connecting the outgoing arc of one node to an ancestor node in the graph, as in the `Iterative Factorial` Act in Figure 2.2. While most planning systems are not able to use plots with cycles, such plots are necessary for executors. Execution of a plot requires successful execution of all nodes along some path from the start node to one or more terminal nodes (depending on whether the terminal nodes are disjunctive or conjunctive). Successful execution of a node requires satisfaction of all the node's goal expressions.

Plot nodes come in two types, *conditional* and *parallel*. Conditional nodes are drawn as single-border rectangles, and parallel nodes are drawn as double-border ovals. In Figure 2.1, nodes `P503` and `P512` are parallel, while all other nodes are conditional. Arcs coming into and going out of a parallel node are *conjunctive*, meaning that all the arcs need to be executed. For the plot in Figure 2.1, `P503` specifies that the air force is to be mobilized. Since it is a parallel node, its successors can be invoked in either order or at the same time. `P512` joins the parallel actions and cannot begin execution until both its incoming branches have completed. During planning, both branches are inserted into the plan as unordered subplans.

Arcs coming into and going out of a conditional node are interpreted as *disjunctive*, meaning that only one of the arcs need be executed. Consider first a disjunctive node with multiple successor nodes. A planner produces a conditional plan following this node. An executor executes the successor nodes until one is found whose goals are satisfied. At that point, execution 'commits' to the branch headed by that successor node and ignores all other branches. A disjunctive node with multiple incoming arcs can be executed as soon as one of its ancestor nodes has been successfully executed. As an example,

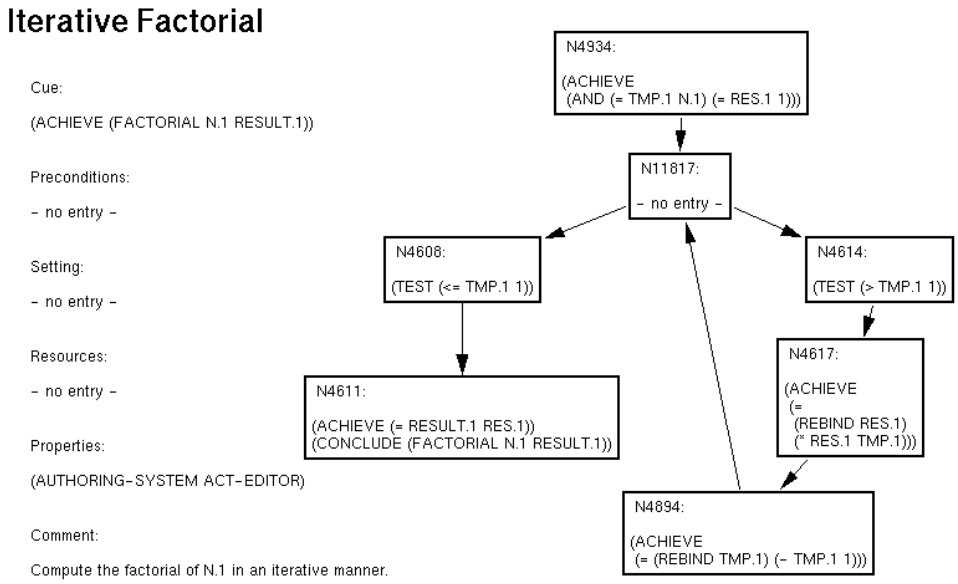

Figure 2.2: Iterative Factorial Act

consider the Act in Figure 2.2 for computing the factorial of a number in an execution system (this Act is not intended for use by a planner). After execution of `N11817`, the executor nondeterministically chooses one of its successor nodes for execution. If the goal expression on this choice is satisfiable, then the executor continues executing that branch. If not, it tries to satisfy the other successor. In this Act, one of the two successors always succeeds. In general, they might both fail and then `N11817` is said to fail.

Two consequences of the typing conventions should be noted: (1) if a node has zero or one incoming edge and zero or one outgoing edge, it is irrelevant whether it is a conditional or a parallel node, and (2) if one action is to be activated by only one of its incoming edges and must activate all of its outgoing edges, then it must be represented by a conditional node that collects the incoming edges followed by a parallel node that collects the outgoing edges. The metapredicates may appear on either one of the nodes, while the other node would be empty. We considered alternative representations for plots that combine disjunctive incoming edges and conjunctive outgoing edges, but the complexity of such representations makes them hard to understand.

2.1.5 Variables

Act uses *typed variables*. In particular, the name of the variable indicates a class to which any instantiation of the variable must belong. e.g., `airplane.1` is a variable for objects in the class `airplane`.

The hierarchy of classes is defined in the dictionary. To fully support the capabilities of the Act-Editor, class names used in variables should be present in the dictionary. When SIPE–2 is loaded, the *Init Objects* command can be used to write the SIPE–2 sort hierarchy into the current dictionary as the hierarchy of classes.

Non-typed variable names can be entered. This happens when Acts are written without a dictionary, when variable names are used whose classes have not been defined in the dictionary, or when other variable naming schemes are used. In such cases, the full power of the Act-Editor will not be available. In particular, the Verifier, the *Edit-Structure Node* command, completion, and the building of dictionaries will not work. While reasoning systems that use Acts (e.g., SIPE–2 and PRS-CL) support non-typed variable names, their use is not recommended.

By default, variables are treated as *logical*, meaning that they denote a single, fixed object. As such, they can be bound to at most one value during execution of an individual Act. (Each application of an Act is associated with its own local variables, so the variables in different applications of the same Act are distinct.) Logical variables can be contrasted with the *dynamic* variables employed in standard programming languages; a dynamic variable is rebound to different values throughout its lifetime.

It is sometimes necessary to use dynamic variables within Act plots. One such situation arises during the writing of an Act that must execute a loop a certain number of times. Another situation arises when an Act is to be used to monitor the value of some changeable feature of the environment (such as the pressure measured by a gauge) and to take certain steps when the value crosses some threshold. Such Acts are difficult to write using only logical variables.

For this reason, the Act language supports two methods for rebinding variables during execution of an Act. The first is through the use of the Achieve-all metapredicate, and the second requires the user to explicitly specify where the rebindings are allowed. These specifications are made by applying the function `REBIND` to variables for which rebinding is to be allowed. For example, the expression

```
(ACHIEVE (= (REBIND X.1) (+ 1 X.1)))
```

expresses the goal of rebinding the variable `X.1` to the value that is 1 greater than the current value of `X.1`. This expression is different from

```
(ACHIEVE (= X.1 (+ 1 X.1)))
```

which seeks to equate a value with a value that is 1 larger, and hence always fails. The Act `Iterative Factorial` in Figure 2.2 illustrates how the `REBIND` function can be used to specify plots with loops.

2.2 Grasper-CL Representations for Acts

Acts are represented as Grasper-CL spaces and can be displayed graphically. Figure 2.1 is taken from the screen of the Act-Editor.

Act environment slots are mapped onto an array of individual Grasper-CL nodes along the left-border of the space. These nodes have invisible icons, meaning that only their labels are displayed. The labels consist of the slot name along with the slot values. The node containing the Act name appears in the upper-left corner. These nodes have fixed locations (that is, they are not movable using Act-Editor commands). However, there is an option to depict the slots as "buttons" without displaying their contents. In this case, the buttons do have icons, and clicking on the button displays the slot as described above.

Act plot nodes are individual Grasper-CL nodes, while plot arcs are Grasper-CL edges. Conditional plot nodes are drawn with a single-border rectangle and parallel nodes are drawn with double-border oval. Each plot node is labeled with a system-generated identification number and the associated metapredicates for the node.

Chapter 3

Using the Act-Editor

The procedures for loading and using the Act-Editor are described here. This chapter begins with a description of how to load the Act-Editor system, followed by an overview of the Graphical User Interface (GUI). The chapter also contains a summary of the command menus, a description of the birds-eye view window, and an example session that illustrates the process of creating an Act.

`Act` is the default package for Acts. This is a data package for all the formulas in an Act. The default package for the Act-Editor code is `act-lisp`, which also has the nickname `clg` (for historical reasons). All functions and variables described in this manual are in the `act-lisp` package unless otherwise noted.

3.1 Loading and Initializing

The Act-Editor may be loaded indirectly as part of some larger system (e.g., Cypress, SIPE–2, and PRS-CL all load it). Otherwise, the Act-Editor must be explicitly loaded. Loading requires entering the appropriate window system, running a COMMON LISP that includes CLOS and CLIM, and evaluating `(sri:run-system :act-editor)`. Run-system loads the system if the system is not already loaded. To load the Act-Editor without running it, evaluate `(sri:load-system :act-editor)`. After the system is loaded, the GUI can be created by evaluating `(run-act-editor)`. Figure 3.1 illustrates the initial GUI window for the Act-Editor.

Grasper-CL can support more than one application a time. Clicking on the *Application* command on the top left portion of the screen displays a menu of the Grasper-CL-based application systems that are currently loaded. Selecting any of these menu items transfers control of the window to the chosen system. Note that evaluating `(sri:exe-history)` prints version information about all loaded systems.

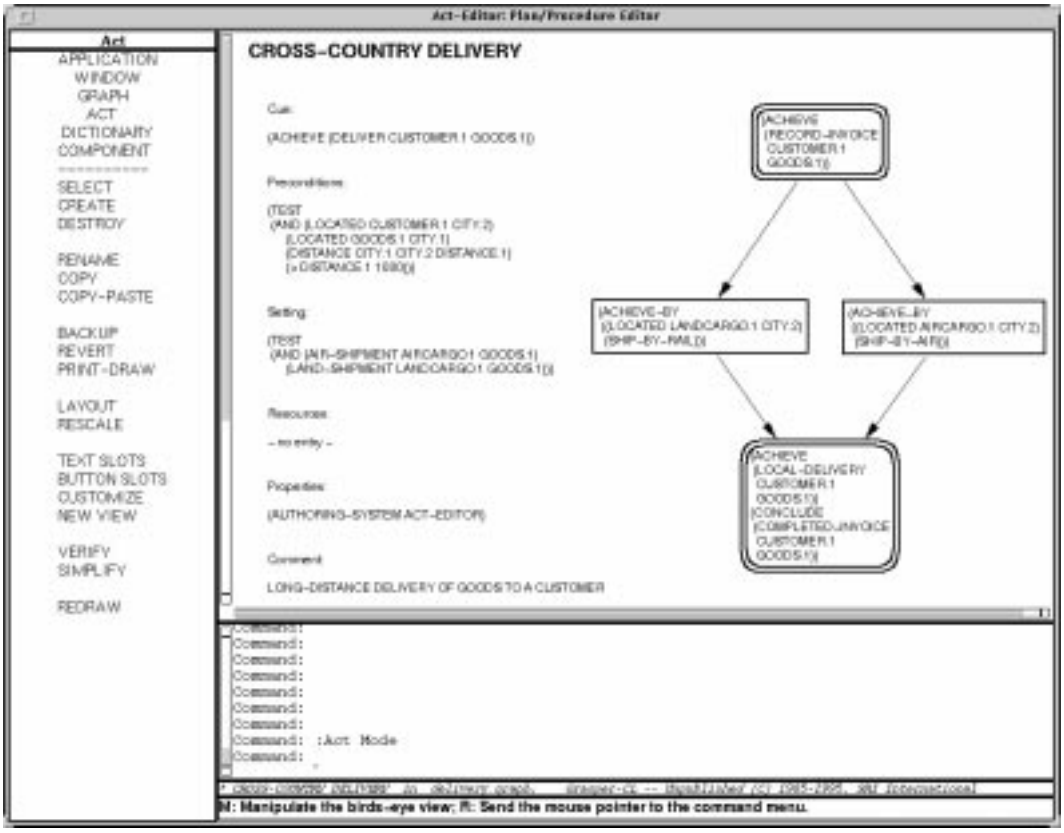

Figure 3.1: The Act-Editor's Display Window

3.2 Organization of the Act-Editor GUI

The GUI consists of two windows: the *display* window (shown in Figure 3.1) and a (usually) smaller *birds-eye* window (see Figure 3.3). The display window constitutes the principal part of the GUI. The birds-eye window is used to assist in viewing large Acts, and is discussed further in Section 3.4.

The display window is divided into several regions called *panes*. Each pane presents different kinds of information during the editing process. The Act-Editor supports several pane layouts, as described in Section 6.1. Here, we describe the default layout.

The largest area in the GUI is the *graph pane*, which always displays the current Act of the current graph (barring errors and anomalies). In Figure 3.1, the graph pane is displaying the Cross Country Delivery Act. This pane is scrollable in case the current Act is too large to be displayed in its entirety. In the upper-left corner is the *mode pane*, which displays the name of the current command mode (when not in any command mode, it displays `Act Editor`). The Act-Editor has a separate mode of operation (and hence a separate collection of commands) for different types of structures. There are five modes: Window, Graph, Act, Dictionary, and Component. Beneath the mode pane is the *command*

pane, which displays a menu of the modes, plus all the commands within the currently selected mode. In Figure 3.1, the system is in Act command mode. Clicking left on a command in the command pane invokes it, while clicking right displays its documentation (in CLIM 2).

Directly below the graph pane is the *interactor pane*. When the Act-Editor is waiting for the user to invoke a command it is said to be at the *top level*, and it displays the prompt `Command:` in the interactor pane. At this point, the user can either click on a command name in the command pane, or enter any Lisp form. That form is evaluated in the context of the currently selected space and the currently selected graph, and its results printed in the interactor pane. Chapter 10 describes what to do if the system appears to be at the top level, but does not respond.

When the user clicks on a command in the command pane, the command name is printed in the interactor pane. The interactor pane is also where various Act-Editor commands print prompts for the user, and where input typed by the user in response to those prompts appears. This pane is scrollable.

Below the interactor pane is the *status pane*, which displays the name of the currently selected graph and space. The asterisk at the left of the status pane indicates the Act has been modified since it was read from disk. Below the status pane is the *documentation pane*, whose contents changes as the mouse pointer passes over different objects on the screen. For example, when the pointer passes over a command name in the command pane, documentation about that command appears in the documentation pane.

Alternative pane layouts supported by the Act-Editor allow the user to modify the size and locations of the various panes (see Section 6.1). For instance, the interactor pane can be increased or decreased in size. In addition, the command pane can be relocated to the top of the window, with the various modes implemented as pull-down menus.

3.3 Command Modes

The Act-Editor has five command modes: Window, Graph, Act, Dictionary, and Component. The names of these modes appear at the top of the command pane, beneath the *Application* command. The command modes determine the type of object on which the Act-Editor is currently operating. Window mode operates on the GUI and the system as a whole, Graph mode operates on Grasper-CL graphs (i.e., files of Acts), Act mode operates on individual Acts, Dictionary mode operates on dictionaries, and Component mode operates on the plot nodes and environment slots within an Act.

Left-clicking on a mode command such as *Component* enters that command mode, and displays a command menu specific to the currently selected mode. Figure 3.2 contains the command menus associated with each of the modes. These commands are described in more detail in the following chapters. To identify commands unambiguously in this document, we generally write the name of a command followed by the name of the current command menu, such as *Create Component*. However,

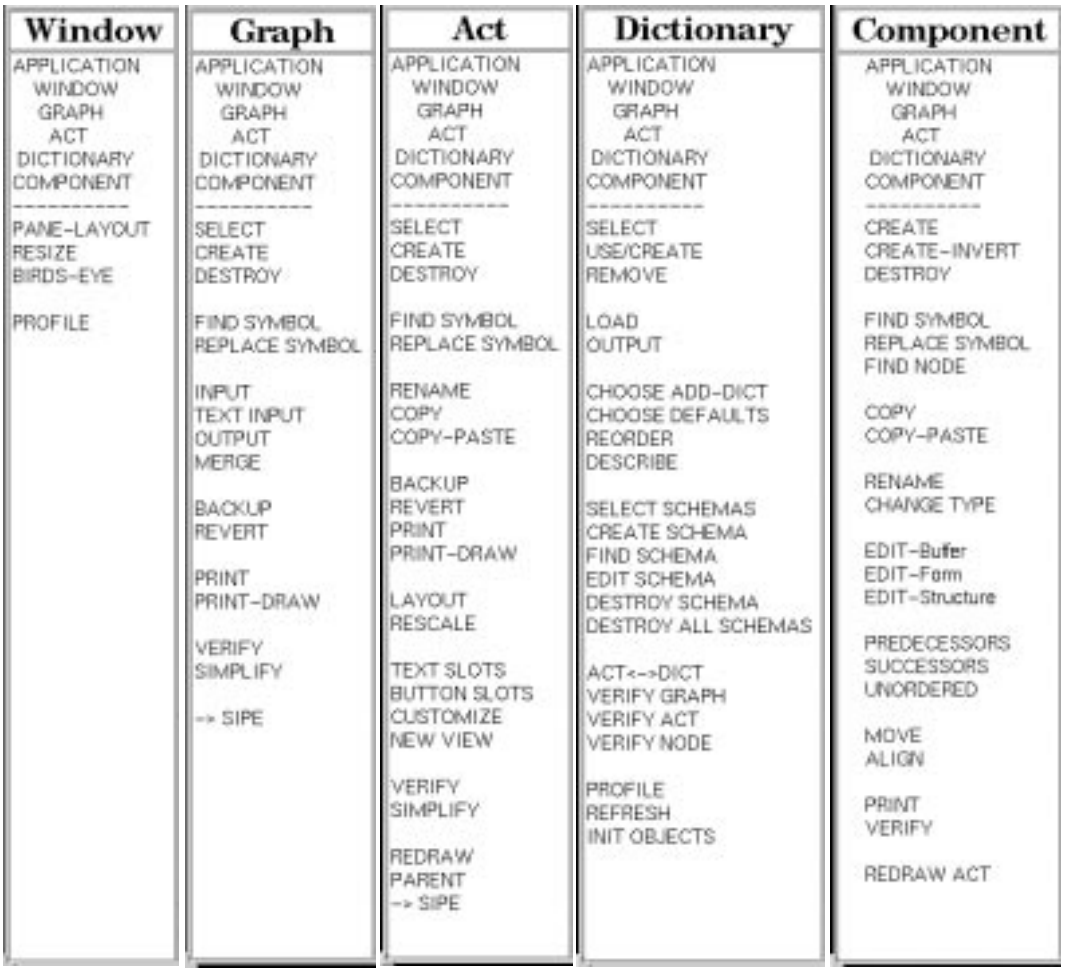

Figure 3.2: Act-Editor Command Menus in Full-Menu Mode

certain Component commands are followed by *Node* rather than *Component* to indicate that they apply only to nodes, and not to all components.

Commands often prompt the user for information, which the user can supply by (1) typing to the interactor pane, (2) clicking on objects in the graph pane, or (3) clicking and typing to pop-up menus that the commands create. For example, the *Rename Node* command asks the user to click on a plot node to be renamed, and then prompts the user for a new name in a pop-up window.

Additional commands can be activated directly by mouse clicks from the Act-Editor top level, as shown in Table 3.1. For example, left-clicking on a plot node allows the user to drag the plot node to a new position.[1] These commands are accessible only from the Act-Editor top level: while a particular command is being executed, the meaning of clicks within the graph pane is determined by that command. The mouse-activated commands are Grasper-CL commands rather than Act-Editor

[1]See Known Bug #1, Section 10.3, for a discussion of problems with node-dragging.

Mouse button	Object clicked	Action
left	plot node	Move plot node
middle	node	Print node name
middle	*background*	Warp to birds-eye window
right	*background*	Warp to command menu

Table 3.1: Mouse-activated Commands at the Act-Editor Top Level

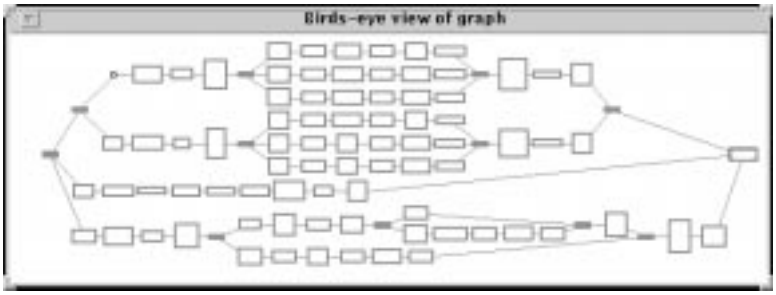

Figure 3.3: The Birds-Eye Window

commands; a full overview of the mouse-activated capabilities can be found in Table 4.1 of the *Grasper-CL Documentation Set* [5]. Clicking on the scroll bars will scroll the graph pane in various standard ways. Additional Grasper-CL commands can be used by selecting Grasper-CL after clicking on the *Application* command, provided one understands how they differ from their Act-Editor counterparts.

3.4 The Birds-Eye View Window

Large graphs do not fit in the graph pane in their entirety; therefore, the graph pane has been constructed as a scrollable viewport onto the entire graph drawing. The birds-eye view window (or simply, the birds eye) provides a low-resolution view of the entire graph that is scaled to fit completely in one window. Figure 3.3 presents a sample birds-eye view window. The birds eye is useful for visualizing and navigating through a large, complex graph. The *Birds Eye Window* command controls what is displayed in the birds-eye window.

The user can manipulate the birds-eye window when in *birds-eye mode*, which is entered by middle-clicking on the background of the graph pane. In birds-eye mode, the user can select commands from a menu of the following items:

- **Scroll Graph** — A rectangle within the birds eye shows the position of the graph pane viewport within the entire graph. The user can reposition this rectangle to a new position to scroll the graph pane. After selecting this menu item, left click in the background of the birds-eye window and drag the rectangle to a new position.

- **Return To Graph** — This command exits birds-eye mode and returns to the top level of the Act-Editor.

The remaining commands in the birds-eye mode menu are still under development and are not yet implemented. They should not be used.

3.5 Display of Environment Slots

The standard display of environment slots is as an array of Grasper-CL nodes along the left-border of the Act. These nodes have invisible icons, meaning that only their labels are displayed. These nodes have fixed locations: they are not movable using Act-Editor commands, and commands that redraw the space often realign the environment slots to their standard location.

However, there is an option to depict the slots as "buttons" without displaying their contents.[2] In this case, the buttons do have icons denoting whether the slot is empty or has a value. Clicking on the button displays the slot in its standard textual representation. Clicking on the text collapses the slot to a button again.

Using buttons permits more of the screen to be used to display plot nodes, and also makes all slots easily accessible in a small portion of the screen. Acts written for planners often have large preconditions, and the Precondition slot then fills up most of the screen. Without buttons, the user must repeatedly scroll the graph pane to search for the other slots.

There is an option to have the buttons appear across the top of the screen when all slots are buttons. Top buttons can be used with horizontal plots to allow more space on the screen for plot nodes. This option is not the default and must be specifically selected (see Chapter 5).

3.6 Inputs

This sections provides hints for using the CLIM interface. Most of the mouse-based selection operations are activated by left-clicking on some appropriate item. For this reason, the phrase 'click on an item' is used throughout this guide to mean 'click using the left mouse button'. When either middle- or right-clicks are required, they will be mentioned explicitly.

Some commands immediately put up a menu of choices or a window asking for confirmation. For confirmation windows, you must hold the mouse button down after clicking on the command and drag over to the desired choice before releasing the button. A quick click will result in no action being taken — the confirmation window may breifly flash on the screen, but it will have been exited without

[2] This option is disabled by PRS-CL as described in Section 5.4.

confirmation for the action. For commands that put up a menu of choices, there are two options. Again, you can hold the mouse button down after clicking on the command and drag over the list of choices that appear. Alternatively, a very quick, sharp click will cause the menu to appear and remain for later selection. A more lingering click with have no effect — the menu will flash on the screen but be exited without a selected choice.

When inputing to CLIM, Emacs-like control characters often provide editing capabilities during type-in (including control-Y to yank text). (Assuming you have employed SRI's `clim.Xdefaults` file as specified in the installation instructions.) For example, such editing capabilities are available while typing to the interactor pane and while typing expressions in menus. In the interactor pane, meta-control-Y will recall the last input, and meta-Y can then be used repeatedly to yank previous inputs. On Sun workstations, the `Del` key and not the `Backspace` key must be used for deletions.

Certain commands employ a type of interactive menu called a Parameter Choice (PC) Menu. This menu displays the current settings of a number of program parameters, which the user can then alter. Here we describe how these menus handle input in CLIM 2.[3] Some parameters have a small fixed number of possible values; these can be selected from a menu with the current value displayed next to a "radio box" indicating there is a menu of choices.

Other parameters can take on a value that the user provides; for example, this situation arises with the command *Edit-Form Node*. There are two different input forms, one with a CLIM box around the entry, and one without. They differ significantly in how input is terminated.

Without a CLIM box, left-click on the current value to delete it and enter an alternative, and middle-click on the value and a cursor will appear and Emacs-style editing commands can be used to modify the entry. Input *must* be terminated by the Return key; hitting Return immediately will simply reinsert the original entry (even if it has disappeared after a left click). When all parameters have the desired values, the user should click on `Save Changes` (or equivalent) at the bottom of the menu. Clicking on `Abort` restores the original values.

With a CLIM box, either a left-click or a middle-click will activate the box for input and allow Emacs-style modification of the existing entry. In this box, control-K does not kill the rest of the line, but meta-control-DEL will. If other edits are to be made, input *must not* be terminated by the Return key; hitting Return causes the menu to exit as if `Save Changes` had been clicked.

A second type of pop-up window is used for entering a single input (as is the case when a name is required for a new Act). These windows act like PC menu inputs without a CLIM box. That is, left-click replaces, middle-click modifies, and Return exits.

Two conventions are observed throughout our CLIM interfaces. First, whenever a pop-up menu is present on the screen, pressing control-Z with the mouse pointer inside the menu aborts the execution

[3] In CLIM 1, things are different, but fairly obvious. For example, "radio-box" choice are all displayed and can be clicked to select.

of the current command and returns to the top level, or, for some commands in the Component menu that loop selecting components, to the selection loop. [4] Control-Z also returns to the top level if pressed when keyboard input is expected.

Second, at times when the user is prompted for keyboard input, a default response is printed between square brackets or quotation marks. The user can select the default by pressing either Return or control-meta-Y followed by Return. Control-meta-Y can also be used to yank defaults in pop-up menus. Defaults often appear in pop-up menus, but when the user clicks on a default, it disappears. Typing control-meta-Y yanks the default back into the buffer.

3.7 Example Act Editing Session

This section describes how a user can interact with the Act-Editor to create a graph containing an Act similar to that pictured in Figure 3.4. It is assumed that the Act-Editor is already loaded. Commands mentioned in this section will be described in the following chapters.

We begin by creating a new graph and an Act within that graph.

- Click on the *Graph* command in the upper part of the command menu to enter Graph mode; the graph commands appear in the command pane.
- Click on the *Create Graph* command to create a new graph, and enter the name of the file in which the graph should later be saved (such as `"/homedir/examples.graph"`) in response to the prompt.
- The next prompt asks for the name of a new Act within that graph — type `Cross-country Delivery`.

The Act-Editor sets the current Act to be this newly created Act and displays it in the graph pane. At this point, the Act consists only of nodes for the environment slots. The name of the Act is displayed in the upper-left corner. All other environment slots are displayed as icon-less nodes labeled by the slot name and the current entry. Initially, all entries are empty except for the Property slot, which lists the Act-Editor as the authoring system for this Act. To specify the Act further, it is necessary to first select the `Component` command to enter component mode.

Let's start by filling the environment slots. This can be done with any of the *Edit-Structure Node*, *Edit-Buffer Node*, or *Edit-Form Node* commands. The first prompts with a series of self-explanatory menus and constructs a formula in the correct syntax. The second permits editing of one large expression with Emacs commands, but requires a complete knowledge of the syntax. Here the use of the *Edit-Form Node* command is described.

[4] In CLIM 1, because of a bug in CLIM, the user sometimes must next press some key (e.g., Rubout or any letter) with the mouse inside the GUI, or commands in the command pane will not be mouse sensitive.

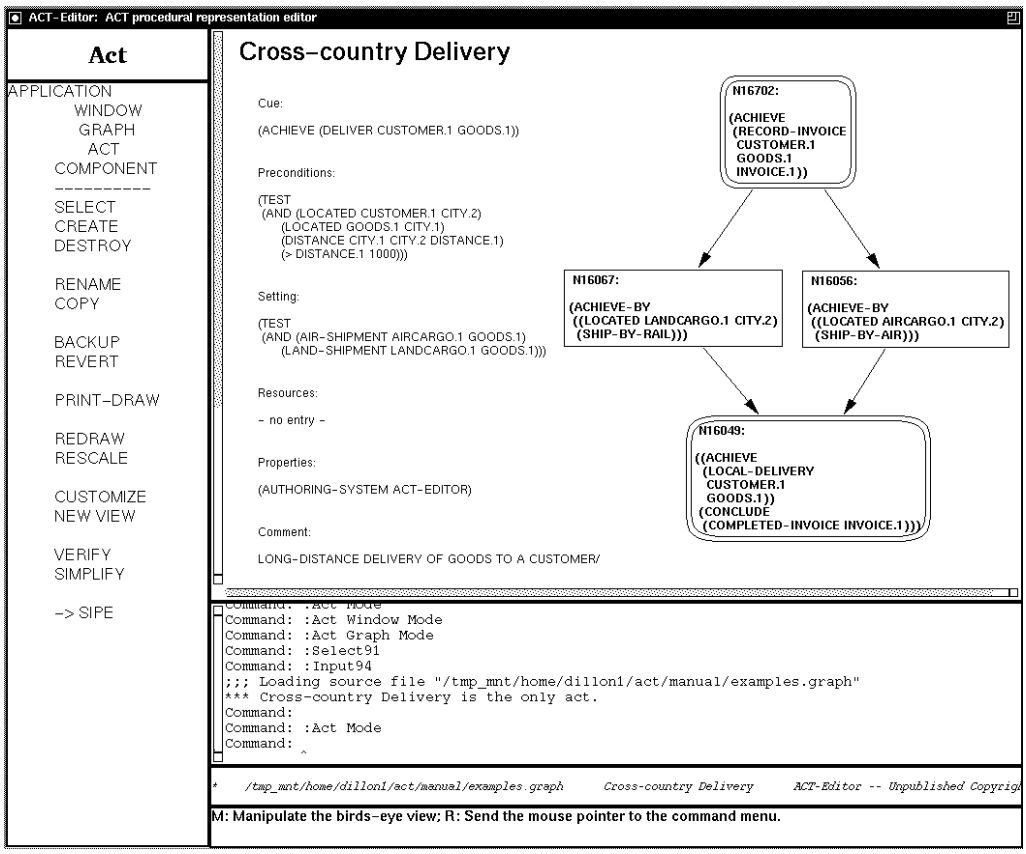

Figure 3.4: Sample Act

- Click on the *Component* command in the upper part of the command menu to enter Component mode; the component commands appear in the command pane.
- Click on the *Edit-Form Node* command, which enters a loop for editing and/or examining nodes.
- Click on the Cue slot. A pop-up menu containing the allowed metapredicates for an Act Cue appears.
- Click on the entry ?? for the metapredicate Achieve. The ?? disappears, being replaced by a blank line headed by a small cursor.
- Type in the appropriate formula for the Test metapredicate using Emacs commands, followed by Return.
- Exit the pop-up window by clicking on the Save Changes button at the bottom.
- To change other environment slots, point at them with the mouse and repeat the process. When all the slots have been filled, right-click on the background to exit.

Now suppose there is a typing error in one of the entries.

- Click on the *Edit-Structure Node* command, which enters a loop for editing nodes.
- Click on the slot with the error. Continue holding the mouse button down until the cursor is over the desired metapredicate in the pop-up menu that appears. After the selection, a series of menus, customized for the syntax of the selected metapredicate, allows editing of that metapredicate.
- NOTE: Remember to terminate non-boxed entries in a PC menu by pressing Return, but do not press Return in boxed entires (except to exit the menu).
- When finished with the modifications, click right on the background to exit the editing mode.

3.7.1 Creating and Editing an Example Plot

Now we move on to the plot. We will create two parallel nodes, corresponding to nodes `N16702` and `N16049` in Figure 3.4, which we will hereafter refer to as the "split node" and "join node" respectively.

- Click on the *Create Component* command, which enters a loop for creating components (nodes and/or edges).
- Click at the approximate positions for the nodes. Don't worry about the exact positions of the nodes; we will fix those later using the *Move Node* and *Align Component* commands.
- Click right on the background to exit the creation loop.

Note that the nodes have different system-generated names and are conditional nodes. Since we desired parallel nodes:

- Click on the *Change Type Node* command, which again enters a node selection loop.
- Click on the new nodes to change them to parallel nodes.
- Click right on the background to exit the change-type loop.

To create the remaining nodes, we'll use a different capability of the *Create Component* command.

- Again, select the *Create Component* command, entering the component-creation loop.
- Click on the split node.

We are now in a nested loop for creating edges or node/edge combinations. If a node is now clicked, an edge will be created from the split node to it. If the background is clicked, a new node will be created together with an edge from the split node to the new node. This behavior continues until a right-click exits the nested loop.

- Successively click left on the background at the approximate positions of where the nodes `N16057` and `N16056` should be situated. The Act-Editor creates both the nodes and edges to them from the split node.
- Click right on the background to terminate the nested node/edge creation loop, returning to the top-level component-creation loop.

At this point, a second right click would return us to the top level of the Act-Editor. Instead, for each of the two nodes just created, we will enter the nested edge creation loop to link it to the join node.

- Click on the new node, then on the join node to create the edge.
- Right-click on the background to exit the nested loop.
- Repeat the above two steps for the second node.

Alternatively, this last sequence of creating predecessor links for the join node can be accomplished by using the *Create-Invert Component* command. With a larger number of nodes preceeding the join, this alternative becomes distinctly more efficient.

- Right-click twice on the background to return to the top level.
- Select the *Create-Invert Component* command, entering the component-creation loop.
- Click on the join node, entering the nested edge-creation loop.
- Repeatedly click on each predecessor of the join node.
- Right-click twice on the background to exit both loops and return to the top level.

At this point, the Act should resemble the Act in Figure 3.4. Now use the *Edit-Structure Node*, *Edit-Buffer Node* or *Edit-Form Node* command to fill in the metapredicates for the individual plot nodes.

3.7.2 Layout of an Example Plot

The last step in creating the Act is to reposition the nodes attractively. This can be done automatically using the *Layout Act* command described in Chapter 7. Here, we describe a custom layout done by the user.

First, we move the split node to its final position.

- Click on the *Move Node* command, entering a node-selection loop.
- Click on the split node.

- Click on the desired position of the split node; the node jumps to this new position.
- Right-click to exit the mode.

The same result could have been obtained by dragging the node with the mouse (as described in Section 3.3), provided the desired destination is already on the screen with the node. Because of CLIM problems, node movement occasionally leaves duplicates of the moved node on the pane (see Known Bug #1, Section 10.3). Should that happen, click on the *Redraw Act* command to refresh the display.

Now we align the join node with the split node.

- Click on the *Align Component* command. You can hold the mouse button down and drag over the list of choices that appear. Alternatively, a very quick, sharp click will cause the menu to appear and remain.
- Choose the *align vertically* option from the first pop-up menu that appears (by rolling off the right edge), and the *center* from the second menu.
- Click on the split node. A vertical line passing through the center of this node appears.
- Click on the join node to bring it into alignment with the split node.
- Right-click on the background to halt alignments with the split node. At this point, the system is still in alignment mode.
- Repeat the same process using the *align horizontally* option to align the other two nodes.
- Click right on the background to exit alignment mode.

3.7.3 Saving and Printing the Act

The Act is now finished. It is necessary to save the graph (remember, a graph is a collection of Acts, not the plot you just created) to disk, because all our changes have been to the internal, working representation of the Acts.

- Click on the *Graph* command in the upper part of the command menu to enter Graph mode.
- Click on the *Output Graph* command.
- You will be prompted for a file name, with the graph's file as the default. To get the default, press Return. If you have reconsidered your choice of file name, you can modify the file name with Emacs-commands before the file is written.

The *Output Graph* command saves all the spaces (Acts) in the current graph, including the one we just created, to a disk file. To read the graph from disk at a later time, use the *Input Graph* command.

You can print all the Acts in a graph with the *Print-Draw Graph* command. To print the Act we just created, go the following:

- Click on the *Act* command in the upper part of the command menu to enter Act mode.
- Click on the *Print-Draw Act* command and select “Draw Picture To Printer” from the pop-up menu. You may want to first select the printer with the “Print-Draw Setup” option that appears on the menu.

Chapter 4

Dictionaries

Advanced concepts: This section is primarily of interest to programmers and can be skipped by users. This chapter describes the concept of dictionaries, their relationship to verification, their creation, and their use for completion. Finally, the commands in the Dictionary menu are described. The Act-Editor can be used without using dictionaries; however, the full power of the Act-Editor will not be available. In particular, the Verifier, the *Edit-Structure Node* command, and completion will not work.

The Act-Editor supports the creation and use of *dictionaries*, which represent knowledge about the terms used in the representation of a domain (the domain ontology). A dictionary is a graph that contains at least three spaces:

- An Object-Types space containing an ISA hierarchy of object types (classes).
- A Predicates space for predicate schemas that represent the number and type or arguments for each predicate.
- A Functions space for function schemas that represent the number and type or arguments for each function.

While the knowledge in dictionaries is not necessary for using an Act, it is of great utility in knowledge acquisition, user interaction, and verification. These concerns become more important as domains become larger and more realistic. For example, in recent SIPE–2 domains there are hundreds to thousands of objects, each with descriptive names of a few dozen characters (e.g., cargo-pump2-richmond-480 and sf-bay-wildlife-refuge). There is a great potential for misspelling and typing cramps, which dictionaries can ameliorate.

The Act-Editor (and SIPE–2) use dictionaries to provide completion of names being input to the system, e.g., an input might complete to all names of a certain type. Typing the Space Bar completes

the name if it is uniquely specified, or brings up a menu of all possible completions of the input so far. This is enormously useful when acquiring new Acts and during interaction with the system.[1]

The Act-Editor provides a *Verifier* that can be used to verify that Acts use predicates and functions as specified by the dictionaries in use. The Verifier can automatically create a new dictionary for an existing set of Acts (for use when it is known that the Acts are correct). Alternatively, the Verifier can be used to build up a dictionary interactively, where the user can take appropriate action for unknown symbols.[2]

Multiple dictionaries are permitted so, for example, there can be one dictionary (shared by several applications) and another for the domain-specific aspects of the problem. With multiple dictionaries, there is the concept of the *add-dictionary*, which is the dictionary selected for the addition of new schemas. Multiple dictionaries are ordered, and the Verifier searches them in the given order.

In this chapter, predicate schemas will be described, but they are identical to function schemas, so all statements apply equally to function schemas. Schemas allow the same predicate name to be used with different numbers of arguments (each different number of arguments essentially defines a different predicate). Also, a schema specification for the arguments of a predicate can be disjunctive. Each schema is represented by a Grasper-CL node.

4.1 Dictionary and Verification Profile

To be general over many domains, there are several options for using and creating dictionary schemas. Schemas are typically built by invoking the Verifier and having it correct conflicts that are found between an Act and the existing dictionaries (see Section 4.3).

The *Profile Dictionary* command can be used to select the correct options for schema building. It brings up the menu in Figure 4.1. The first option allows selection of one of the following modes for running the Verifier:

- **Log-only.** Warning messages for all conflicts are printed to *terminal-io* (to allow editing in Emacs). Neither Acts nor dictionaries are modified. This supports making corrections in an editor, e.g., doing string replacements in Emacs to process repeated misspellings.
- **Auto-modify-dictionary.** The dictionary is built automatically, assuming all Acts are correct.[3]

[1] The full capability of completing all arguments of a predicate is available only when SIPE-2 is also loaded in the same image as the Act-Editor. The Act-Editor alone can complete only individual names, such as predicate names and variable names.

[2] The Verifier currently does not have a dictionary for the predicates and functions that are built into PRS-CL, so these constructs will be treated as unknown during verification. For this reason, `act-prs-mode` currently removes commands to invoke the Verifier and disables dictionaries.

[3] There is no mode for changing Acts automatically because the intended spelling cannot be determined.

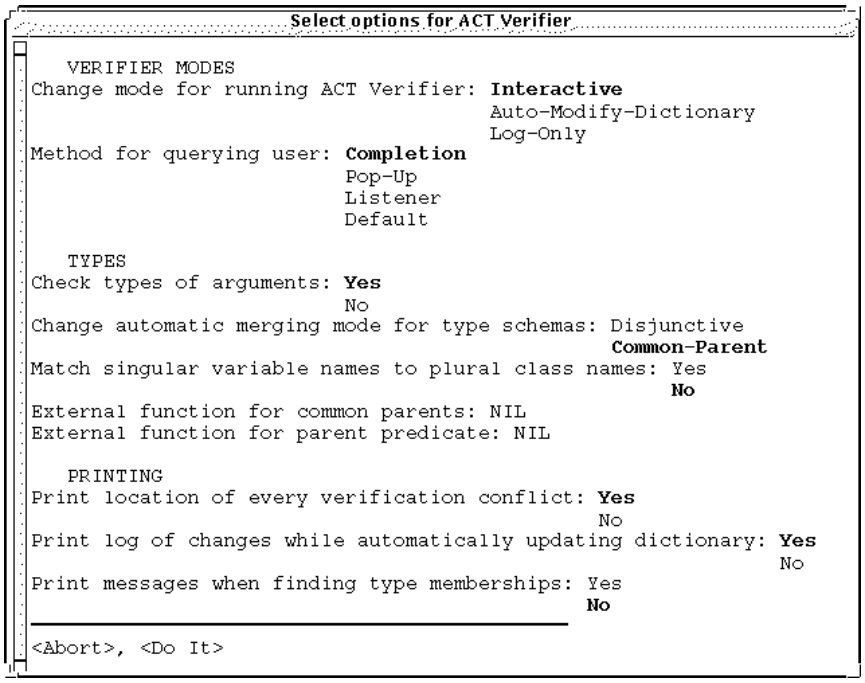

Figure 4.1: Profile Dictionary Menu

- **Interactive.** Through a series of menus, the user can change both Acts and/or dictionary schemas for every conflict. This is the default mode.

Most other options in Figure 4.1 are self-explanatory. The options regarding "external functions" apply to using external knowledge bases. The printing options at the bottom should probably be disabled for an automatic dictionary creation, since they can print a large volume of text.

The fourth option regarding "merging" needs more explanation. Building a predicate schema in the Verifier involves extending it to cover a predicate instance in an Act that currently violates the schema. Thus, the schema specifies one class, A, for some argument, while the predicate instance has an object of another class, B, for that argument. The Verifier must merge these two classes. It provides the choice of merging with a disjunctive schema (where the argument must be either A or B), or of keeping a single nondisjunctive schema by using the common parent of A and B in the schema. Common-parent schemas are the default and should be used when possible.[4] The fourth option allows choosing whether two legal classes for an argument should be merged by taking their common parent or creating a disjunction.

Some of these options should be chosen when the Verifier is being used for SIPE–2 Acts. In

[4]While dictionaries and the Verifier support disjunctive schemas, disjuncts are not currently used when completing the arguments of a predicate. For now, the system uses the first disjunctive schema during completion. Thus, if entering predicates with completion is desired, the default option of common-parent merging is best. Common-parent merging also makes dictionaries simpler and smaller.

particular, the options for checking types of arguments and matching singular variables to plural classes should be selected. The latter option permits classes in the Object-Types space to be named with either singular or plural names. For example, if the variable `location.1` is used in an Act, and the class `Location` is not in the dictionary, the system will then use the class `Locations` if it exists.

As described in Chapter 5, there is a standard profile to load for SIPE–2. Note that some or all dictionary commands are disabled in user mode and in the standard PRS-CL profile. Evaluating `(full-menu-mode)` activates all Act-Editor commands. The following variable can be used to disable dictionaries:

`*use-act-dictionaries*` [*Variable*]
A value of `nil`*, instructs the system to do the best it can without dictionaries. A value of* `t`*, the default, instructs the system to use the* SIPE–2 *sort-hierarchy for objects, and dictionaries for predicates and functions. This is the default because the Act-Editor does not currently support individual objects in the dictionaries (only classes). The* SIPE–2 *hierarchy allows completion on individual object names. A value of* `:always`*, instructs the system to use dictionaries even for objects.*

4.2 Creating a Dictionary

Several tools exist for creating dictionaries. We first describe the manual creation of a dictionary by the user. Section /refdictionary-tools describes how SIPE–2 and the GKB-Editor can be used to automate this process.

To create a dictionary for the current graph of Acts, enter the Dictionary menu and select the *Use/Create Dictionary* command and select the Create Graph option. A new dictionary graph is created with the necessary spaces, but it is not saved on disk. The created dictionary is declared as a dictionary in the current graph. If another dictionary is already in use for the current graph, you will be asked whether to order the new dictionary before or after it.

It is also possible to copy an existing dictionary and modify it. Input the dictionary with the *Input Graph* command, then use the *Output Graph* command to save it under a new file name. You can then click *Use/Create Dictionary* and use the new file name you gave to the copied graph.

To see the status of dictionaries associated with the current graph of Acts, select the *Describe Dictionary* command, which will print relevant information in the interactor pane. If dictionaries do not seem to be working in the Verifier or for completion, the *Describe Dictionary* command should be checked.

4.3 Editing and Extending a Dictionary

To add entries manually to the dictionary, first switch to the dictionary graph from the graph with Acts. This can be done by selecting either the *Select Dictionary* command or the *Act<->Dict Dictionary*

command which toggles the display between the Act graph and the add dictionary graph.

The schema editing commands work on the classes in the Object-Types space as well as the predicate and function schemas. The *Select Schemas Dictionary* command selects the space in which you want to view, create or edit schemas or classes. The *Create Schema Dictionary* command can be used to create schemas or classes. When creating classes in the Object-Types spaces, there should be and edge from a class to each of its subclasses. The *Edit Schema Dictionary* command allows modification of an existing schema, including renaming of classes. The *Destroy Schema Dictionary* command allows deletion of selected schemas and classes. Finally, the *Destroy All Schemas Dictionary* command will destroy all the schemas or classes in a selected space.

While the Object-Types space is created manually whenever a SIPE–2 sort hierarchy does not exist for the domain, the other schemas are rarely created manually. Once an Object-Types space exists in a dictionary, and the dictionary is being used by the current graph and is loaded, then the Verifier can be used to create the Predicate and Function spaces. Using the Verifier does not save the dictionary to disk; the user must do this with the *Output Dictionary* command. If the Verifier is invoked in Interactive mode, there will be a dialogue with the user who chooses, for each conflict, whether to modify the Act (e.g., by correcting a misspelling) or the dictionary (by extending a schema or creating a new schema).

If the Verifier is invoked in Auto-modify-dictionary mode, the Predicate and Function spaces are built automatically. This option for building schemas should be used when an existing set of Acts is known to be correct. This process creates schemas for all predicates and functions based on how they are used in the Acts. If there may be errors in the Acts (e.g., misspellings), then incorrect schemas and classes will be created. In such cases, the Verifier should be used interactively.

4.4 Automated Tools for Dictionary Creation

While the Act-Editor can automatically build the predicate and function schemas in the Verifier, other systems also provide tools for creating dictionaries, particularly for creating the hierarchy of object classes.

The *Init Objects Dictionary* command in the Act-Editor allows SIPE–2 to be used in conjunction with manual creation of a dictionary. When SIPE–2 is loaded, this command will write the SIPE–2 sort hierarchy into the current dictionary as the hierarchy of classes in the Object-Types space. Before invoking this command, the dictionary should already have been created and it should be the current graph.

Another SRI system, the GKB-Editor can be used to interactively edit and create a SIPE–2 sort-hierarchy that SIPE–2 can then translate into an Object-Types space in a dictionary. Because of the functionality of the GKB-Editor, this is the preferred method for creating an Object-Types space.

SIPE–2 has completely automated the process of creating a dictionary for a domain it has loaded (that is assumed correct). This is invoked with SIPE–2's -> *ACT all* command [9], which does the following:

- Create a graph with the Act translations of all operators in it
- Create another graph and link it as the dictionary of the first graph
- Copy the sort hierarchy into the dictionary graph
- Run the Verifier on all Acts to automatically build up the dictionary.

4.5 Using an Existing Dictionary

To be used, dictionaries must be declared in the current graph and must be loaded. Declaration allows different graphs to automatically load their own dictionaries. Declaration is done interactively with the *Use/Create Dictionary* command. Unfortunately, Grasper-CL does not support naming graphs with logical pathnames. Therefore dictionaries are specified with local pathnames and need to be changed when used at another site. The Act-Editor facilitates this by prompting for another pathname to insert as the declared dictionary when the specified dictionary path is not found. The Act graph declaring the dictionary must then be saved to disk after the correct path is given.

The following variable allows declaration of a global dictionary for graphs with no declared dictionary. Its use is not recommended since it must be reset whenever the domain is changed:

`*global-dictionaries*` [*Variable*]
A value of `nil` *is the default, which implies that graphs specify their dictionaries. This variable can be set to a list of graph names. The Act-Editor uses these dictionaries whenever it cannot find another. The user (or domain files) is responsible for resetting this global variable if Acts for another domain are loaded.*

If dictionaries do not seem to be working in the Verifier or for completion, the user should determine whether a dictionary is declared for the current graph (using the *Describe Dictionary* command), and that it is loaded.

4.6 Completion

In the *Edit-Structure Node* command and when querying the user for input about the domain, the Act-Editor uses PC menus that allow completion of names being input. Pressing the Space Bar completes the name if it is uniquely specified, or brings up a menu of all possible completions of the input so far.

Completion has become enormously useful as domains have grown in size. Names for Acts, predicates, objects, classes, and variables can be completed.

For example, when the Act-Editor asks the user to choose an Act, it uses a menu for a small number of choices. When the number of choices is larger than a given threshold, it uses typing with completion since CLIM has problems with menus taller than the screen.

The full capability of completing all arguments of a predicate or function formula (as opposed to individual names) is available only when SIPE–2 is loaded in the same image as the Act-Editor. SIPE–2 generates a PC menu for a predicate that has one entry for each argument, where each entry completes to its specified type. The Act-Editor automatically uses the SIPE–2 functionality whenever SIPE–2 is loaded (unless disabled by the profile). The Act-Editor alone can only complete individual names, such as predicate names and variable names.

If completion does not work, the user should determine whether a dictionary is declared for the current graph (using the *Describe Dictionary* command), that it is loaded, and that the profile has not disabled dictionaries (see Section 4.1).

Some known CLIM bugs affect completion. They are not difficult to work around, as long as the user is aware of them:

1. After completing to a legal name, then typing to extend it to a new name, CLIM reverts back to the completed name. To extend the name, simply backspace over its last character, type it again, and then continue typing.
2. After selecting a completion from a menu, it is sometimes necessary to press the return key to see it.
3. When no completions are found, CLIM erases the input so far. It is necessary to reenter any substring that had been typed.

Chapter 5

Customization

Advanced concepts: This chapter is primarily of interest to programmers and can be skipped by users.

Different behaviors of the Act-Editor are preferred in different application domains, or by different users. Therefore, the Act-Editor provides several options for customizing its behavior. For example, selecting an Act from a menu of all Act names is desirable when there are only a few Acts, but other selection methods may be preferred when there are 500 Acts.[1]

The *Profile Window* command is used to customize the behavior of the Act-Editor.[2] However, this command is not present in the standard user mode (see below). Evaluating `(full-menu-mode)` activates all Act-Editor commands.

Several customizations are described here. Customizations for the creation and use of dictionaries are described in Chapter 4.

5.1 Commands

While programmers may wish to have all Act-Editor commands available, too many commands can be confusing to users. Therefore, the Act-Editor provides three command-menu modes:

- :hacker – This mode, also known as full-menu mode, has all commands and is preferred for programmers developing domains with dictionaries, and anyone wishing to use the commands relating to using buttons for environment slots.
- :user – This mode removes several commands from the Dictionary menu, and all *Simplify* commands. The commands for button environment slots are removed, as is the *Profile Window* command.

[1] Particularly since CLIM has problems with menus taller than the screen.

[2] Loading PRS-CL changes the profile, so any invocation of the *Profile Window* command must be redone after loading PRS-CL.

- :no-verify — This mode removes the same commands as the user mode, and in addition removes the entire Dictionary menu, and all *Verify* commands form the Graph, Act, and Component menus.

The *Profile Window* command can be used to change modes while in full-menu mode, but the other modes remove the *Profile Window* command. The following function can be called to switch command-menu modes:

`change-menu-mode` [*Function*]
This function takes an argument that is one of `(:hacker :user :no-verify)`, and resets the command menus to the given mode. `(full-menu-mode)` is equivalent to `(change-menu-mode :hacker)`.

The following sections describe options available through the *Profile Window* command, which prompts for each option with an explanatory description. The variable names provided in the following sections can be used in initialization files of particular applications.

5.2 Selecting Acts

Several customization options relate to the selection of Acts by the Act-Editor.

`*select-with-completion*` [*Variable*]
The default value, 30, causes the system to use with typing with completion to select an Act whenever there are more than 30 Acts. With fewer Acts, selection is done by a menu of choices. The default number can be changed, a value of 0 disables the menu option, and a negative value disables the typing-with-completion option. When this value is too large for your screen or negative, the menu may extend beyond the boundary of the screen and attempts to resize or move it will result in the menu disappearing. This variable is also used by other selection commands, such as selecting a node name to find.

`*never-select-non-act*` [*Variable*]
The default value is `nil`*. A value of* `t` *causes the Act-Editor to never select a Grasper-CL space that is not an Act. This was the default behavior before version 1.06. However, only selecting Acts has two disadvantages. First, the user cannot view other spaces without first switching to Grasper-CL. In particular, the dictionary spaces cannot be drawn. Second, if the user selects a graph with no Acts (perhaps unintentionally), the system leaves the GUI in an inconsistent state where the space on the screen does not correspond to the current space. Most commands will then result in error messages saying that no Act is selected when there is a valid act drawn on the screen.*

`*allow-non-acts*` [*Variable*]
If `*never-select-non-act*` *is* `t`*, this variable is ignored. Otherwise, if this variable is* `nil`*, the default, Acts are selected whenever possible, but a a non-Act is selected whenever there is no Act, and no action would leave the GUI inconsistent with the system. If this variable is* `t`*, the user can select spaces that are not Acts when using the Select Act command. Whenever a non-Act is selected, the Component menu cannot be entered, and only a small number of commands work in the Act menu.*

5.3 Display of Environment Slots

Environment slots can optionally be displayed as "buttons" so that more of the screen is used to display plot nodes. Buttons make all slots easily accessible in a small portion of the screen, avoiding the need to repeatedly scroll the graph pane to search for environment slots. When buttons are enabled, clicking on an environment slot toggles it between a button and its standard textual representation. There is an option to have the buttons appear across the top of the screen when all slots are buttons. Top buttons are useful with horizontal plots.

Note that the following variables are set by the standard profiles described in Section 5.4.

`*buttons-enabled*` [*Variable*]
The default value is `t`*, which allows environment slots to become buttons when clicked. Clicking on the buttons restores the environment slots. Buttons permit more of the screen to display the plot. A value of* `nil` *disables buttons.*

`*top-buttons*` [*Variable*]
The default value is `nil`*, which means buttons appear on the left, as the environment slots did. A value of* `t` *places the buttons across the top of the screen when all slots are buttons, allowing horizontal plots more space on the screen.*

5.3.1 Editing and Verifying Nodes

Several options exist for interacting with menus while editing and verifying nodes. These variables apply to the Verifier and the *Edit-Structure Node* command. The second variable below specifies how the user prefers to be queried for information during node editing. The system has a default for each type of user query.

`*looping-edit-menus*` [*Variable*]
While editing a node, there is a choice of whether to continually display the editing choices until exit is clicked, or to perform one edit and return. The default value of `nil` *selects the latter option, which allows the user to see the newly edited entry after each edit. A value of* `t` *continually displays the edit menu until "exit" is clicked. This saves an extra click for each subsequent edit, but requires an extra click to exit. In addition, CLIM will not display the newly edited form between edits, although clicking "print" in the edit menu prints the current form. Clearly, the default is best if there is only one edit per invocation of the Edit-Structure Node command. Looping is better when several edits are done on one long formula.*

`*user-input-mode*` [*Variable*]
With the default value of `:completion`*, the system uses typing with completion whenever it is available, otherwise the default option is used. With a value of* `:default`*, the system always uses a default option specified by the function requesting input (which may be different for different inputs). Two other values are permissible:* `:pop-up` *and* `:listener`*, which prompt in pop-up windows and the Lisp Listener pane, respectively.*

5.4 Standard Profiles

The Act-Editor provides two functions, `act-sipe-mode` and `act-prs-mode`, for selecting a whole set of the above customizations. These are the customizations generally used by users primarily developing Acts for a planner, and users primarily developing Acts for execution systems, respectively. For example, SIPE–2 domains generally have many objects with long names, so typing names with completion is preferred. Similarly, Acts for planning often have long preconditions. This means the Precondition slot fills up most of the screen, and the user must repeatedly scroll to find the other slots. Thus, act-sipe-mode selects the option of displaying slots as buttons that are expanded when they are clicked. Buttons make all slots available in a small portion of the screen.

The Verifier currently does not have a dictionary of the predicates and functions that are built into PRS-CL, so these constructs will be treated as unknown during verification. For this reason, `act-prs-mode` currently removes commands to invoke the Verifier and disables dictionaries.

`act-sipe-mode` [*Function*]

This function sets several variables to customize the Act-Editor for SIPE–2. Environment slots are displayed as mousable buttons across the top of the graph pane. Typing with completion is used rather than menus when the list of choices is longer than 30. The Verifier is set to use common-parent merging, check argument types, and allow plural class names.

`act-prs-mode` [*Function*]

This function sets several variables to customize the Act-Editor for PRS-CL. `:No-verify` mode is used for command menus, which removes the Dictionary menu and commands to invoke the Verifier. Buttons are disabled for environment slots, and the commands to invoke them are removed. The *Copy-Paste Act* and *Profile Window* commands are removed. Menus are always used to select from a set of choices, no matter how large the set is. The Verifier is set to not look for plural class names, and to not check the types of arguments. Temporal ordering edges other than the default ordering edge are disabled. Another key, *Retract*, is added to editing menus. The Act syntax uses the Conclude metapredicate for doing retractions, but Retract is needed to support existing Acts.

Calls to these functions can be placed in initialization files. For example, users who prefer act-sipe-mode can insert a call to `(act-sipe-mode)` in user initialization files. A programmer might insert such a call in a application domain file. However, loading PRS-CL automatically calls `act-prs-mode`. Therefore, `act-sipe-mode` must be called after PRS-CL is loaded. Likewise, any customizations done by the user (perhaps using the *Profile Window* command) must be done (or redone) after PRS-CL is loaded.

Chapter 6

Window Mode and Graph Mode

Detailed descriptions of the commands in the Window and Graph menus are given here.

6.1 Window Mode

The Window Mode commands can be used to change system-wide defaults. In particular, the appearance of both the graph pane and the birds-eye window can be changed. The current space and graph remain unaltered.

Pane-Layout Window

This command alters the arrangement of the panes within the Grasper-CL window. The user can choose one of a fixed set of prespecified layouts from a pop-up menu.

Resize Window

By default, the CLIM graphics system does not allow scrolling outside the boundaries of an existing drawing. This restriction is problematic when the user wishes to create new nodes or edges that (for example) lie to the right of the rightmost area of the graph pane that can be exposed by scrolling. *Resize Window* allows the user to expand the size of the graph pane drawing plane in the positive direction along both coordinate axes, that is, rightward and downward. This command pops up a PC menu that displays the current extreme X and Y coordinates. These extrema are set originally to the largest X and Y coordinates present in all nodes in the current Act, which also represent a lower bound on the values that the user can assign. By increasing the X extremum, the user creates a new drawing Act on the right edge of the graph pane where new nodes and edges can be drawn. The drawing process can be less efficient with a large virtual window.

Birds-Eye Window

This command is used to disable or enable the birds-eye window and also controls what is displayed in the birds-eye window.

Profile Window

This command is used to customize the behavior of the Act-Editor, as described in Chapter 5.[1] However, this command is not present in the standard user mode. Evaluating `(full-menu-mode)` activates all Act-Editor commands.

6.2 Graph Mode

Graph Mode provides commands that operate on Grasper-CL graphs. A graph is primarily a file that is associated with the set of Acts in the graph (graph-wide properties can also be stored with a graph). Because graphs are the unit of transfer between memory and disk, the Act-Editor name for a graph is simply the name of the file in which the graph is (or will be) stored. Upon starting the Act-Editor, the first action a user should perform is to either input a graph from disk, or to create a new graph (and a new Act within that graph).

Select Graph

An arbitrary number of different graphs can be loaded within the Act-Editor at one time. At any instant, the graph displays only the *current Act* from the *current graph.* The *Select Graph* command allows the user to determine which of the loaded graphs should be the current graph. If this graph contains more than one Act, the user is prompted to select one as the current Act.

Create Graph

This command creates a new graph within the Act-Editor, and makes it the current graph. The user is prompted for the name of the file in which the graph will be stored. The file name should end in ".graph". The user is also prompted for the name of a new Act to be created in the graph.

[1] Loading PRS-CL changes the profile, so any invocation of the *Profile Window* command must be redone after loading PRS-CL.

Destroy Graph

The *Destroy Graph* command removes the current graph from Act-Editor memory. All unsaved changes to the graph are lost. The disk file in which the graph was last saved (if any) is unaffected by this operation.

Find Symbol Graph

This command prompts for a symbol and then prints the names of all the Acts in the current graph that contain this symbol. The type of symbol is first requested in a menu so that completion can be used to find the valid entries.

Replace Symbol Graph

This command replaces one symbol with another in all spaces in the current graph. This affects properties of nodes in the spaces, not their names, nor any properties of graphs or spaces as a whole. This command first asks for which type of symbol you wish to replace (so it can use completion) and then asks for two symbols, and replaces all occurrences of the first with the second. This command does not work in dictionary graphs.

Input Graph

This command reads a Grasper-CL graph from a disk file and makes that graph the current graph. The command pops up a menu that prompts the user for the name of the directory from which the graph should be read; the user can either select a directory already present in the menu, or choose the option of inputting a new directory name. The directory name can be either a physical pathname for the local machine (e.g., a UNIX pathname), or a logical pathname that has been defined in COMMON LISP using the function `lp:logical-pathname-translations` (see [7, p632] and the supplied source file `logical-pathnames.lisp`).

After selection of the directory, the system presents a menu of all files in the specified directory that end in ".graph". The user can click on the graph to be input.

Text Input Graph

This command reads a disk file containing the ASCII representation [6] of a sequence of Acts. Each Act in the file is created as an Act in the current graph. The command uses the same series of menus as the *Input Graph* command for specifying the file name.

Output Graph

The *Output Graph* command saves the current graph to a disk file; the user is prompted to type in the name of the desired file. This file name can be either a physical pathname (such as a UNIX pathname) or a logical pathname (see [7, p628]). The saved graph can later be read into the Act-Editor using the *Input Graph* command.

Merge Graph

This command merges the current graph with another loaded graph that the user selects from a pop-up menu. All Acts in the selected graph are copied into the current graph. If both graphs contain Acts with the same name, the Act in the current graph is discarded and replaced by the Act from the selected graph.

Backup Graph

The *Backup Graph* command creates a memory-resident backup copy of the current graph. This command enables the current state of the graph to be recovered at a later time if that state is altered in undesirable ways by subsequent graph editing. This command does not save the graph on secondary storage, and therefore does not protect the graph from any event that would corrupt the current Lisp process. The *Revert Graph* command is used to recover the backup created by *Backup Graph*.

Revert Graph

Revert Graph replaces the current graph with the backup copy of the current graph as created by the *Backup Graph* command. All changes in the memory-resident version of the current graph are lost.

Print Graph

The *Print Graph* command writes a file containing the ASCII representation [6] for each Act in the current graph. The new file has the same file name as the current graph except the type field of the filename is changed to `.text` (normally a graph file has a type of `.graph`). This representation is readable by other programs that support Act. The Acts are written with the task, plan, and action networks syntax used by Act to describe plans at multiple levels of abstraction [6]. If no *plan* and *task* properities are given in the properties of the Acts, then the tokens *unknonw-plan* and *unknown-task* will be used for grouping.

To get multiple action networks grouped into a single plan expression, they must all be in the same graph, unless you set the following variable:

`*htn-graphs*` [*Variable*]
The default value is `nil`*. The value can be a list of file names of Grasper graphs that together compose the current task and/or plan.*

Print-Draw Graph

Two forms of hardcopy graphs are supported, and each form can be either sent to a printer or saved in a file. *Print text* is an ASCII dump of the internal Grasper-CL data structure. *Draw picture* is a PostScript rendition of the graphical form of a graph that the Act-Editor displays.

The *Print-Draw Graph* command pops up a menu asking which of these two forms of output is desired, and whether the output should be sent to a printer or saved in a file. It then prints or draws all Acts within the current graph to the specified destination.[2] Drawing the graph to a file is a special case. Every Act in the current graph will be drawn to a file called *Act-name*`.ps`, in a directory whose name is supplied by the user. On UNIX systems, directory names must end with the "/" character.

Verify Graph

This command invokes the Verifier on all the Acts in the current graph. The Verifier can automatically or interactively verify Acts and build a dictionary (see Chapter 4).

Simplify Graph

The simplifier is used to streamline the logical structure of an Act. The simplifier eliminates both unnecessary plot nodes and redundant ordering links. These simplifications produce Acts that are semantically equivalent but often more compact. The compactness is of benefit primarily for viewing purposes.

-> SIPE Graph

This command translates the set of Acts in the current graph into SIPE–2 operators. The results are stored in a text file, whose name is supplied by the user in response to a prompt in the interactor pane. This command is present only when SIPE–2 is loaded.

[2]Previous problems printing acts have mostly been solved. Acts can be scaled to fit one page, printed in landscape or portrait format and in black/white or color. However, default printing of an act tends to be rather large and the "fit to one page" option is still limited.

Chapter 7

Act Mode

Commands in Act mode operate on individual Acts within the current graph. An individual Act is referenced by its name. By default, the name of the current Act is displayed in the upper-left corner of the graph pane and in the status pane. Some of the most basic of these commands (e.g., Select) also work on Grasper-CL spaces that are not Acts (e.g., dictionary spaces).

Select Act

This command allows the user to set the current Act to be any of the Acts in the current graph. The chosen Act is displayed automatically in the graph pane.

Create Act

This command creates a new act within the current graph, and makes that Act the current Act.

Destroy Act

This command deletes the current Act from the current graph.

Find Symbol Act

This command prompts for a symbol and both highlights all the nodes in the current Act that contain this symbol and prints their names in the interactor pane. The type of symbol is first requested in a menu so that completion can be used to find the valid entries. This command does not work in dictionary graphs.

Replace Symbol Act

This command replaces one symbol with another in the current Act. This affects properties of nodes in the spaces, not their names, nor any properties of graphs or spaces as a whole. This command first asks for which type of symbol you wish to replace (so it can use completion) and then asks for two symbols, and replaces all occurrences of the first with the second. This command does not work in dictionary graphs.

Rename Act

This command supports the renaming of the current Act. The user is prompted for a new name.

Copy Act

This command creates a copy of the current Act within the current graph. The user is prompted for a new name. The new name should not be the name of an existing space in the graph.

Copy-Paste Act

This command supports copying a set of nodes and slots from one Act and pasting them into a different Act. Clicking on this command pops up a menu of commands to support copying to the edit buffer, pasting from the edit buffer, clearing the edit buffer, and viewing the contents of the edit buffer.

A typical use of this command begins by clicking *Copy-Paste Act* which brings up the copy-paste menu. If the edit buffer is not empty, the *Clear Buffer* command should first be selected. Various copy-paste commands can then be used to copy individual nodes, the whole plot, and/or the environment slots from the current Act into the edit buffer.

Once the edit buffer has the desired contents (the *Describe Buffer* command prints the contents), the user then navigates to the desired graph and Act for pasting. This can be done by selecting or creating both graphs and Acts. Once in the desired Act, the user again clicks the *Copy-Paste Act* command but this time selects the *Paste buffer into Act* command from the copy-paste menu.

Backup Act

Much like the *Backup Graph* command, *Backup Act* creates a memory-resident backup copy of the current Act. The backup copy can be restored at a future time using the *Revert Act* command.

Revert Act

Revert Act replaces the current Act within the current graph with the most recent backed-up copy of the Act that was created using the *Backup Act* or the *Backup Graph* commands. The current version of the Act is destroyed.

Print Act

The *Print Act* command writes a file containing the ASCII representation [6] of the current Act. The name of the file is *Act-name*`.text`, and it is in the same directory as the current graph. The Act is written without any surrounding task or plan syntax that describe plans at multiple levels of abstraction [6]; however any *plan* and *task* properities will be written as part of the properites slot.

Print-Draw Act

This command provides similarly functionality to *Print-Draw Graph*, but prints or draws only the current Act.

Layout Act

This command relocates all plot nodes using the default layout algorithm of the Act-Editor, and then redraws the Act. The algorithm seems to be a fairly general layout for plots, and attempts to be compact, without overlapping nodes (but the problem is very difficult) . It works for both horizontal and vertical plots. The algorithm tries to retain the same layout direction that already exists. It determines this by finding the first node with a singleton outgoing edge, and determines whether its successor is displaced vertically or horizontally. If this algorithm does not find a direction (e.g., when the nodes have no locations or equal locations), the default layout direction is used:

`*default-layout-direction*` [*Variable*]
The default value is `:left-to-right`. *A value of* `:top-to-bottom` *is also permitted.*

By setting the following variable, a tree layout algorithm can be used instead:

`*layout-plot-as-tree*` [*Variable*]
The default value is `nil`. *A value of* `t` *uses the Grasper-CL tree layout algorithm, which does a reasonable layout only if the plot is a true tree, although even then the layout is very wide so not many nodes get on the screen. For non-trees, the layout is ugly and can cause errors.*

Rescale Act

This command scales the positions of the nodes and edges in the current Act. A pop-up PC menu asks the user to specify scaling and translation factors for the X and Y dimensions that are applied to every plot node and edge in the current Act. These scaling factors affect the positions of nodes and edges only — the sizes of labels and icons are not changed. If, for example, an X scale factor of 2 were given, then the X coordinate of every node and edge would be multiplied by 2. The origin for the graph pane is located in the upper left corner. Thus, using scale factors for X and Y that are greater than 1 cause the Act plot to expand downward and to the right while scale factors less than 1 shrink the Act upward and to the left.

Resize Window can be used in conjunction with *Rescale Act* to open up new empty space for adding nodes and edges to the graph pane. For example, we could increase the Y extremum using *Resize Window* to add empty space at the bottom of the graph pane, and then translate the entire Act downward to create empty space at the top of the pane.

Text Slots Act

This command converts all environment slots to their textual representation and redraws the Act.

Button Slots Act

This command converts all environment slots to their button representations and redraws the Act.

Customize Act

This command can be used to change display parameters for plot nodes in the current Act. When this command is selected, a pop-up menu appears with the parameters and their current settings. The current parameters that can be changed are the *pretty-print width* for the Act and the *node drawing style*. The former specifies the maximum line length for the display of plot node labels.

Several node drawing styles are available for selection. The *default* style displays all metapredicates for the node along with the node identifier. The *no-label* option prints all metapredicates but not the node identifier; the *no-comments* style prints everything but the Comment metapredicate. These two styles are useful for reducing the size of plot nodes. The *only-comments* style displays only the Comment metapredicate (if there is one), and is useful for summarizing the documentation of the plot.

The *Customize Act* command changes the Act itself; to revert to the original drawing parameters, it is necessary to explicitly undo any customizations. In contrast, The *New View Act* command presents alternative displays of an Act without changing the Act itself.

New View Act

This command can be used to draw alternative displays of Acts, while leaving the underlying Act structure unchanged. The user can vary the amount of detail to be presented on each plot node, with the layout of the Act automatically changed to adapt to the addition or removal of displayed information. Alternative views are useful when examining large Acts, since their size can render them unwieldy. For example, viewing an Act with only the metapredicate names displayed at each plot node will make the plot much smaller and enable many more nodes to fit on the screen.

Clicking *New View Act* presents a menu that allows the user to specify which metapredicates, labels, and arguments to base-level predicates should be suppressed in the new view. For a given Act-name, a new Act called 'Act-name (VIEW)' is created using the requested changes in the displayed information. This secondary *view Act* is necessary to prevent corruption of the original. *New View Act* executes the default layout algorithm on the view plot. Only one view for a given Act is possible at any point in time. Additional requests for views will overwrite previous views of the same Act.

Verify Act

Verify Act performs the verification process described for the command *Verify Graph* but only for the current Act.

Simplify Act

Simplify Act performs the simplification process described for the command *Simplify Graph* but only for the current Act.

Redraw Act

The Act-Editor always attempts to automatically update the graph pane so that the display accurately reflects the nodes and edges present in the current Act. There may be times when the display in the graph pane diverges from the contents of the current Act, or when the contents of the graph pane become corrupted. Executing *Redraw Act* redraws the current Act in the graph pane. This command does not relocate nodes — if nodes overlap, the *Layout Act* command must be used to relocate nodes.

Parent Act

This command displays the parent Act of the current Act (as specified in the `:parent` property), possibly switching to a different graph if the parent Act is not in the current graph and the `:parent-graph` property is specified.

-> SIPE Act

This command translates the current Act into a SIPE–2 operator. The user is prompted for a file name in which to store the translated operator. This command is present only when SIPE–2 is loaded.

Chapter 8

Dictionary Mode

The commands in Dictionary mode primarily operate on dictionaries and their spaces and schemas. The "current dictionaries" are the dictionaries associated with the current Act graph. Some of these commands assume the current graph is an Act graph, and operate on the current dictionaries of that graph. Other commands, e.g., the schema editing commands, assume the current graph is a dictionary. Some commands, e.g., output, operate on both types of graphs. The schema editing commands work on the classes in the Object-Types space as well as the predicate and function schemas.

The commands that prompt for a node/schema follow the same selection convention as the commands in Component mode. Namely, the command repeatedly prompts the user to left-click on a node on which to operate. The user can exit this loop and return to the Act-Editor top level by right-clicking on the background of the graph pane.

Note that some or all dictionary commands are disabled in user mode and in the standard PRS-CL profile. Evaluating `(full-menu-mode)` activates all dictionary commands.

8.1 Commands Operating on Act Graphs

Most of the Dictionary commands in this section assume that the current graph is an Act graph. The others operate on both Act graphs and dictionary graphs. The following commands manipulate the declaration of dictionaries in the current graph.

Select Dictionary

This command is like *Select Graph* but selects one of the current dictionaries. If there are no current dictionaries, it prints a warning and allows selection of any graph.

Use/Create Dictionary

This command prompts for a dictionary name and associates it with the current Act graph. If a new file is created, then it is made into a dictionary by creating the necessary spaces, but it is not saved on disk. The created dictionary is associated with the current graph.

Remove Dictionary

This command prompts for the name of a current dictionary and removes it from the current Act graph.

Load Dictionary

This command is like *Input Graph* but loads one of the current dictionaries as selected by the user.

Output Dictionary

This command is the same as *Output Graph*. It writes any graph to disk but is available only in full-menu mode; in other modes it is necessary to enter Graph mode to write files.

Choose Add-Dictionary

This command prompts for the name of a current dictionary and specifies it as the add-dictionary in the current Act graph.

Choose Defaults Dictionary

This command allows specifying whether the dictionaries should load automatically whenever the current Act graph is loaded, and whether existing dictionaries should be overwritten.

Reorder Dictionary

This command allows reordering of the current dictionaries.

Describe Dictionary

This command prints the current dictionaries and the current add-dictionary.

Verify Graph
Verify Act
Verify Node

The three *Verify* commands from the corresponding command modes are repeated in the Dictionary menu, but only in full-menu mode. This enables the programmer to quickly build a dictionary without having to constantly change command modes. If these commands are invoked while in a dictionary graph, the system first redisplays the last displayed Act and then executes the command.

8.2 Commands Operating on Dictionary Graphs

Most of the Dictionary commands in this section assume that the current graph is a dictionary. The others operate on all graphs. These commands manipulate the spaces and schemas in the current graph.

Select Schemas Dictionary

This command is used to select one of the spaces in the current dictionary graph.

Create Schema Dictionary

This command creates a new schema or class. The current space must be either the Predicates, Functions, or Object-Types space of a dictionary graph.

Find Schema Dictionary

This command is the same as *Find Node* and finds a schema or class.

Edit Schema Dictionary

This command prompts for a schema to edit. The current space must be a space in a dictionary graph. For classes, the class can be renamed; for other schemas, a series of menus allows the user to edit the selected schema.

Destroy Schema Dictionary

This command prompts for a schema to destroy. The current space must be a space in a dictionary graph.

Destroy All Schemas Dictionary

This command will destroy all the schemas or classes in selected dictionary spaces.

Act<->Dict Dictionary

This command toggles the display in the graph pane between the last Act that was displayed and the last dictionary space that was displayed, provided the dictionary is a dictionary of the current Act graph. If an Act is displayed and none of its dictionaries was the last dictionary displayed, the Predicates space in the current add-dictionary will be displayed. This command is available only in full-menu mode and can be useful during dictionary development. The behavior of this command could also be accomplished by entering Graph mode, clicking on *Select Graph*, clicking on the desired graph, clicking on the desired space, and returning to Dictionary mode. That process of five mouse clicks and scanning three choice menus is replaced by a single mouse click in this command.

Profile Dictionary

This command selects options for schema building within the Verifier. The options are described in Section 4.1, and shown in Figure 4.1. This command can be invoked in any graph.

Refresh Dictionary

When the current space is a Functions or Predicates space in a dictionary graph, this command applies the Grasper-CL array layout algorithm to the current space. In all cases, the space is redrawn.

Init Objects Dictionary

This command is only present when SIPE–2 is loaded. This command will write the currently loaded SIPE–2 sort hierarchy into the current dictionary as the hierarchy of classes in the Object-Types space.

Chapter 9

Component Mode

The commands in Component mode operate on nodes or edges within the current Act. Certain of the component-level commands apply to both nodes and edges, while others operate on nodes exclusively. For example, the *Destroy Component* command applies to all components, while the *Change Type Node* command applies only to nodes. We refer to commands using the name of the command followed by either *Component* or *Node*, depending on the class of objects to which the command can be applied.

Most of the commands in Component mode follow these conventions:

- Once the user clicks on one of these commands, the command repeatedly prompts the user to left-click on a node or edge on which to operate. The user can exit this loop and return to the Act-Editor top level by right-clicking on the background of the graph pane.
- Most of the Component mode commands operate on both nodes and edges. The most common way of selecting the object on which to operate is by left-clicking on that object, although some commands, such as *Create Component*, accept other inputs.

Create Component

The use of this command to create a plot was described in Section 3.7.1.

This command repeatedly prompts the user to create a node, edge, or node/edge combination. Clicking on the background creates a node at that position, while clicking on a node is described below. The nodes have different system-generated names and are conditional nodes, but can be changed to parallel nodes with the *Change Type Node* command. When you are finished creating components, click right on the background to exit the creation loop.

Clicking on an existing node, which we will call the “split” node, enters a nested loop for creating edges or node/edge combinations. If a node is now clicked, an edge will be created from the split node

to it. If the background is clicked, a new node will be created together with an edge from the split node to the new node. When finished creating edges or combinations, two right clicks are needed to exit both loops and return to the Act-Editor top level. Alternatively, on right click will terminate the nested node/edge creation loop and return to the top-level component-creation loop, allowing the creation of additional components.

There is currently no command to create node/edge combinations with edges in the opposite direction. Such a command would be easy to implement, but would add complexity to the interface.

Create-Invert Component

This command is the same as the *Create Component* command, but the edges are created in the opposite direction, i.e., an edge is created from the second node selected to the first node selected. This command is useful for creating a "join" node that has several predecessors.

Destroy Component

This command allows the user to remove nodes or edges from the current Act.

Find Symbol Node

This command is the same as *Find Symbol Act*, and prompts for a symbol and then highlights all the nodes in the current Act that contain this symbol.

Find Node

This command can be used to bring into view nodes that are scrolled outside the visible region of the graph pane. The user selects a desired node from a pop-up menu of all nodes in the current Act, after which the Act-Editor scrolls the graph pane to make the specified node visible.

Copy Node

Copy Node copies a selected node and positions the new copy at a location selected by the user.

Copy-Paste Node

This command is the same as *Copy-Paste Act* and is described in detail in Chapter 7.

Rename Node

Nodes created interactively are assigned system-generated identifiers as names. This command can be used to change node names as desired by the user.

Change Type Node

Nodes created interactively are conditional nodes. This command changes the types of selected plot nodes (e.g., from conditional to parallel or vice versa).

9.1 Edit Node Commands

There are three commands for editing nodes; they allow editing in a buffer, in a form, or in a structure-based editor. They differ along two main dimensions: complexity of the menus used, and knowledge of the syntax required of the user. A user who is completely familiar with the syntax may wish to use the *Edit-Buffer Node* command which is simply an editor buffer with Emacs-like commands, but the user must know the exact syntax. For a user unfamiliar with the syntax, the *Edit-Structure Node* command uses a structure-based editor that obtains specific names from the user through a series of menus. The system then combines the names using the correct syntax. In addition, the *Edit-Structure Node* command allows automatic completion of names. The *Edit-Form Node* is in between the other two commands, allows editing in a form, and is well suited for examining a node to see its contents.

These commands allow the user to enter or modify plot node and environment slot values, including the values for all *keys*. The most common keys are metapredicates, although certain nodes support additional keys (for example, the Properties node allows arbitrary user-defined keys).

Edit-Buffer Node

This command permits editing of one large expression for all the keys on the node. The editing is done in an editor buffer with Emacs-like commands. The user must ensure that the returned expression is in the correct syntax. To get Emacs-like commands in CLIM, the `clim.Xdefaults` file must be loaded as specified in the installation instructions.

Edit-Form Node

Key values for the node are displayed in a form, which is a CLIM window with one entry for each key. This form is well suited for examined a node to see its contents. Editing is possible for values that are not too large for the window. Even large values can be edited, but the CLIM display gets confused.

To enter a user-supplied value, left-click on the current value to delete it and enter an alternative. To modify the given value, middle-click on the value and a cursor will appear and Emacs-style editing commands can be used. Input in both cases *must* be terminated by the Return key; hitting Return immediately will simply reinsert the original entry. When all parameters have the desired values, the user should click on `Save Changes` at the bottom of the menu. Clicking on `Abort` restores the original values.

Edit-Structure Node

This command uses a structure-based editor to obtain names from the user through a series of menus. The system combines these inputs into the correct syntax, thus relieving the user of the responsibility of getting the parentheses right. Initially, a menu prompts the user for the key to be changed. After a key has been selected, the system brings up a series of menus that present editing commands and prompt for lower-level formulas or names. The *Edit-Structure Node* command also completes names, so is often preferred for domains with long names (especially when the nesting level of formulas is shallow).

During the editing process, several features assist the user:

- Emacs-style commands can be used in any entry being edited.
- When dictionaries are being used, most symbols can be completed by hitting the space bar, which completes a unique substring or brings up a menu of legal completions (see Chapter 4). Symbols that can be completed include predicate names, function names, and variables that are the arguments to predicates and functions.[1]
- When entering a string, the string quotes do not need to be typed, as CLIM will add them automatically when needed. However, CLIM never displays the quotes in its menu entries. Thus, argument lists containing strings may appear incorrect, but the internal structure is in fact correct.

Predecessors Node

This command prompts the user for a plot node. All predecessors of the selected node are then highlighted.

Successors Node

This command prompts the user for a plot node. All successors of the selected node are then highlighted.

[1] If completion does not work, a dictionary is probably not declared for the current graph or the declared dictionary is not loaded.

Unordered Node

This command prompts the user for a plot node. All plot nodes that are unordered with respect to the selected node are then highlighted.

Move Node

Move Node supports movement of plot nodes and their associated edges. The command allows the user to select one or more plot nodes by clicking on them, and then clicking on a background location. The selected group of nodes is moved so that the last node selected is centered on the new location. Attempts to move an environment slot will result in no actions being taken. By middle-clicking rather than left-clicking on a node after selecting the *Move Node* command, the user can choose nodes from a menu rather than clicking on them.

Align Component

This command enables the user to reposition nodes relative to each other. When selected, the user is prompted for the type of alignment. It is possible to align nodes horizontally (along their tops, bottoms, or centers), to align nodes vertically (along their left sides, right sides, or centers), to snap nodes to a grid, or to have the nodes spread automatically along a horizontal or vertical axis. It is also possible to center nodes between two designated reference nodes.

Environment slots can be used as reference objects for alignment (that is, objects can be aligned *to* environment slots) but attempts to change the position of an environment slot using the alignment command are ignored.

Print Node

This command prompts for a node and then prints the ASCII representation of this node to the interactor pane.

Verify Node

Verify Node prompts for a node and then invokes the Verifier on that node.

Redraw Act

This command invokes the *Redraw Act* command from the Act menu. Although logically an Act-level command, one frequently wants to redraw an Act when doing a series of component commands, particularly given the CLIM bug that leaves “ghosts” on the screen (described in Section 10.3).

Chapter 10

Troubleshooting the Act-Editor

Certain difficulties may arise during the use of the CLIM version of the Act-Editor. This chapter presents suggestions for recovering when the system is not responding, and describes some known bugs.

One problem that is not a bug occurs when the user wishes to manipulate an Act in a region outside the virtual window viewed in the graph pane. The virtual window can be enlarged by using the *Resize Window* command.

10.1 Recovering from Stuck States

The term *stuck state* refers to the situation when the GUI apparently does not respond to input.[1] Stuck states can arise for different reasons because of known CLIM bugs. Several remedies are given below.

CLIM does not update the GUI window until the Act-Editor returns to the CLIM command loop. Thus the GUI will be unresponsive while the Act-Editor is computing. For example, the *Copy Act* command can take several minutes to finish when applied to a large Act. Suppose the user either closes the GUI window or exposes another window on top of it during this computation. Upon reopening, the GUI window will be completely blank with all displayed information gone and no indication that the GUI is active. Simply wait, and when the Act-Editor returns to the CLIM command loop the screen will be refreshed and displayed information will reappear.

If the Act-Editor is taking longer than expected, then the remedies below should be tried. If none of them work, Unix commands can be used from another window to determine whether the Act-Editor Lisp process is still running. The remedies are these:

[1]This section applies to only the CLIM implementation. Generally, the only stuck state on the Symbolics is an error, which generally appears in the GUI Lisp listener with several reasonable continuation options to select among.

1. The first step to take is to click right on the background. Certain modes of the Act-Editor cause the GUI to enter a loop reading mouse clicks; for example, many of the commands in the Component menu enter such a loop. Clicking right on the background exits such loops, and also recovers from a stuck state described in 2, below.

2. Sometimes when the GUI should be in the command loop (e.g., after returning to the command level from an error), it does not respond to mouse clicks. Let's call this Stuck-state-1. Typing any key (e.g., any letter, Return or Rubout) restores CLIM. Unfortunately, if CLIM is not in Stuck-state-1, typing an extraneous Return puts CLIM in Stuck-state-2.

3. CLIM enters Stuck-state-2 when an extra Return has been typed. In this state, the GUI Lisp listener generally displays the message:
   `The input ‘’’’ is not a complete Lisp Expression. Please edit your input.`
   Doing a right mouse-click anywhere in the GUI restores CLIM from Stuck-state-2. Alternatively, selecting any mouse-activated command in the command pane simultaneously initiates the command and exits Stuck-state-2. A right mouse-click never gets CLIM stuck, so it is always a good first attempt at recovery.

4. Another cause of stuck states is errors, which can be hard to recognize in CLIM. If an error occurs, nothing will appear in the GUI. Any error message will be in the window from which the Lisp process was first run. So if the GUI seems to have stopped, check the Lisp window for an error. Errors generally include a continuation option to return to the Act-Editor command level, and this should generally be selected.[2]

5. If the GUI becomes hopelessly wedged, it is best to kill it and create a new one. Killing is done by interrupting the Lisp process (type controls-C twice to the Lisp window), aborting to the top level, and calling the function `(run-act-editor)` again. If even that fails, try meta-x panic.

10.2 Known Bugs Affecting Completion

The following known CLIM bugs affect completion:

1. After completing to a legal name, then typing to extend it to a new name, CLIM reverts back to the completed name. To extend the name, simply backspace over its last character, type it again, and then continue typing.

2. After selecting a completion from a menu, it is sometimes necessary to press the return key to see it.

3. When no completions are found, CLIM erases the input so far. It is necessary to reenter any substring that had been typed.

[2] If any Act-Editor functions are called from the break in the Lisp window, their output will often still go to the CLIM window and not to the Lisp window where the function is called. To get the output in the Lisp window, execute the LISP function call `(setq *standard-output* *terminal-io*)`.

If completion does not work, the user should determine whether a dictionary is declared for the current graph (using the *Describe Dictionary* command), that it is loaded, and that the profile has not disabled dictionaries (see Section 4.1).

10.3 Other Known Bugs

The user should note the following bugs, which may cause problems during the use of the Act-Editor.

1. There are several annoying but fairly inoccuous bugs with CLIM input windows, including inappropriate window sizing, poor input of lines of characters, and some editing anomalies including boxed characters.
2. Problems can arise with certain commands, e.g., the *Find Node* command, when invoked on Acts with many components. If the number of components is less than the number in the variable `*select-with-completion*`, certain commands issue a pop-up window containing the components in the current Act, from which the user can select one. When there are many choices, the pop-up window may extend beyond the boundary of the screen. Attempts to resize or move the window will result in the window disappearing. The solution is to set `*select-with-completion*` to a smaller number that is appropriate for your screen size.
3. The use of strings in CLIM menus (e.g., while using the *Edit-Structure Node* command) has anomalies. The string quotes do not need to be typed, as CLIM will add them automatically when needed. However, CLIM will never display the quotes in menu entries. Thus, argument lists containing strings may appear incorrect in the CP menu, but the internal structure is in fact correct.

10.4 Bugs Eliminated from Previous version

The user should note the following bugs, which did cause problems in previous versions of the Act-Editor have been fixed in this version.

- The node dragging problem has been eliminated. Nodes may now be dragged in the usual manner (by left-clicking on a node and, keeping the button depressed, moving to the destination before releasing the button). If, in the unlikely situation that, a "shadow" node does appear then they can be eliminated by redrawing the Act (use the *Redraw Act* command in the Act or Component menu).
- Right clicking on an item such as a command or an item in a menu will now bring up a menu of valid choices which may be selected without causing error.
- Problems printing acts have mostly been solved. Acts can be scaled to fit one page, printed in landscape or portrait format and in black/white or color. However, default printing of an act tends to be rather large and the "fit to one page" option is still limited.

Bibliography

[1] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the Association for Computing Machinery*, 26(11):832–843, 1983.

[2] M. P. Georgeff and F. F. Ingrand. Decision-making in an embedded reasoning system. In *Proceedings of the 1989 International Joint Conference on Artificial Intelligence*, American Association for Artificial Intelligence, Menlo Park, CA, 1989.

[3] M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning: an experiment with a mobile robot. In *Proceedings of the 1987 National Conference on Artificial Intelligence*, 1987.

[4] P. Karp, J. Lowrance, T. Strat, and D. Wilkins. The Grasper-CL graph management system. *LISP and Symbolic Computation*, 7:245–282, 1994.

[5] P. D. Karp, J. D. Lowrance, and T. M. Strat. *The Grasper-CL Documentation Set, Version 1.1*. SRI International Artificial Intelligence Center, Menlo Park, CA, 1992.

[6] K. L. Myers and D. E. Wilkins. *The Act Formalism*. Artificial Intelligence Center, SRI International, Menlo Park, CA, version 2.1 edition, May 1997.

[7] G. L. Steele. *Common LISP the Language*. Digital Press, second edition, 1990.

[8] D. E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm*. Morgan Kaufmann Publishers Inc., San Mateo, CA, 1988.

[9] D. E. Wilkins. *Using the SIPE-2 Planning System: A Manual for Version 4.14*. SRI International Artificial Intelligence Center, Menlo Park, CA, February 1997.

[10] D. E. Wilkins and K. L. Myers. A common knowledge representation for plan generation and reactive execution. *Journal of Logic and Computation*, 5(6):731–761, December 1995.

[11] D. E. Wilkins, K. L. Myers, J. D. Lowrance, and L. P. Wesley. Planning and reacting in uncertain and dynamic environments. *Journal of Experimental and Theoretical AI*, 7(1):197–227, 1995.

Appendix A

Application Programmer's Interface

The Application Programmer's Interface (API) for the Act-Editor is briefly described here. First, functions for manipulating the ASCII representations of Acts are described.

Normally, Acts are created and accessed interactively with the Act-Editor. Sometimes it is desirable to create Acts programmatically, e.g., when writing a translator from some language to Act, or from Act to some other language. This appendix next documents the functions of the API that can be used for this purpose. The SIPE–2 planning system includes a translator to and from Act that can be used as an example for other translators.

`Act` is the data package for all formulas in Acts. The default package for the Act-Editor code is `act-lisp`, which also has the nickname `clg` (for historical reasons). All functions and variables described in this manual are in the `act-lisp` package unless otherwise noted.

In this appendix, the name of a function and its arguments are often the only documentation provided. For example, in the arguments of a function, `act` refers to the name of an Act, `graph` refers to the filename of a Grasper-CL graph, `node` refers to either a plot node or an environment slot, and a metapredicate name refers to the expression that give the formula for that metapredicate. Function names ending in "?" are generally predicates that return non-nil if the predicate is satisfied. As a further example, the first functions below describe their arguments in more detail.

Several variables have already been described in the manual. These descriptions are not repeated in this appendix.

A.1 ASCII Act Support Functions

The following LISP functions can be used to manipulate ASCII versions of Acts.

`print-acts-to-file` *&optional (file *grasper-file*)* [*Function*]

This function takes a Grasper graph (file) and writes the ASCII version of all Acts in the graph to a file named *file*.text.

`print-act-to-file` *&optional (Act (space))* [*Function*]

This function prints an Act in the current graph in ASCII form to the file *actname*.text. The default is to print the Act currently drawn on the GUI.

`print-acts` *&optional (stream *standard-output*)* [*Function*]

This function writes the ASCII version of all Acts in the current Grasper graph to the given stream.

`print-act` *&optional (Act (space)) (stream *standard-output*)* [*Function*]

This function prints an Act in ASCII form to the given stream. The default is to print the Act currently drawn on the GUI.

`print-act-node` *node &optional plot? (Act (space)) (stream *standard-output*)* [*Function*]

This function prints a node in the given Act in ASCII form to the given stream. If *PLOT?* is non-nil, then the type of the node is printed.

`input-acts-from-file` *file* [*Function*]

This function reads the ASCII specifications of Acts from the given file into the current Grasper graph.

`input-act` *form* [*Function*]

This function creates an Act in the current graph from the ASCII specification *form*.

A.2 Creating Acts

The functions in this section are used to create Acts programmatically.

`create-act` *act-name &key cue-tuples preconditions-tuples setting-tuples resources-tuples properties-tuples comment-tuples append?* [*Function*]

This function creates an Act with the given name. The keywords allow creation of all the environment slots. However, this function is generally used without keywords and the slots are created with additional function calls. For each environment slot, there is a function named `create-act-<slot>` that creates the corresponding slot. The metapredicates are given with keyword arguments.[1]

`create-act-cue` *act &key achieve test comment tuples append?* [*Function*]

`create-act-precondition` *act &key achieve test comment tuples append?* [*Function*]

`create-act-preconditions` *act &key achieve test comment tuples append?* [*Function*]

`create-act-setting` *act &key test comment tuples append?* [*Function*]

`create-act-resources` *act &key use-resource comment tuples append?* [*Function*]

`create-act-property` *act &key comment tuples append?* [*Function*]

`create-act-properties` *act &key comment tuples append?* [*Function*]

`create-act-comment` *act &key comment tuples append?* [*Function*]

The plural names are synonyms for the singular names. The first argument of each function is the name of the Act, and the allowed metapredicates are given as keywords. In addition, the keywords `:tuples` and `:append?` are accepted. `:Tuples` accepts the syntax for arbitrary properties, and can be used as an alternative means to specify metapredicates by using the metapredicate name as the property name. `:Append?` is a Boolean value that when true specifies that values for a given property are to be appended to any values already present. The default is that the most recent property-value pair would override any existing value for that property. Below are some example calls to these functions.

```
(create-act-cue 'ACT1 :achieve '(P1 A1 B1))
(create-act-property 'ACT1
     :tuples '((act::authoring-system act::sipe-2)
               (act::class operator)))
(create-act-cue 'ACT1
  :tuples '((ACHIEVE (P1 A1)) (ACHIEVE-BY ((Pn An) (act-foo))))
  :append? t)
```

Plots are networks of plot nodes that are ordered by edges. Plot nodes are either parallel or conditional nodes. A parallel node cannot be activated unless all incoming edges are active, and it activates all outgoing edges. A conditional node may be activated when only one of its incoming edges is active, and it in turn can succeed if only one of its outgoing edges is successfully activated.

[1] Metapredicates are implemented as properties in the Grasper-CL representation.

`create-parallel-node` *name act &key location achieve achieve-by test conclude retract wait-until require-until use-resource comment tuples append?* [*Function*]

This creates a parallel node with the given name in the given act. Metapredicates are specified by keywords, with `:tuples` and `:append?` also accepted. The keyword `:location` is also accepted. Its value should be a list of numbers of length two that specify the X and Y coordinates of the node.

`create-parallel-relationship` *source-node &key name location* [*Function*]

Creates a parallel node with name, at location (x y), in Act's plot as a successor of source-node.

`create-conditional-node` *name act &key location achieve achieve-by test conclude retract wait-until require-until use-resource comment tuples append?* [*Function*]

This function creates a conditional node with the given name in the given act. The keywords are the same as for `create-parallel-node`.

`create-conditional-relationship` *source-node act &key name location* [*Function*]

Creates a conditional node, name, at location (x y), in Act's plot as a successor of source-node.

`create-ordering-edge` *source-node destination-node act &key location* [*Function*]

This function creates an ordering edge from source-node to destination-node in the given act. `:Location` is a list of (x y) locations that specify knots along the edge.

`destroy-ordering-edge` *source-node destination-node act* [*Function*]

Destroys any ordering edge from source-node to destination-node in Act's plot.

Below are some example calls to these functions.

```
(create-parallel-node 'foo 'act1
          :conclude (predlist->act (effects opr))
          :location (list start-x start-y))
(create-parallel-node 'baz 'act1
          :location (list (+ 200 start-x) start-y))
(create-ordering-edge 'foo 'baz 'act1)
```

A.3 Accessing Acts

The functions in this section allow access to the information encoded in an Act. Environment slots are referred to as `environment nodes` because they are represented as nodes in Grasper-CL. Plot nodes are also referred to as nodes.

`act?` *name* [*Function*]

This function returns non-nil if its argument is a Grasper space that represents an Act.

`set-of-acts` [*Function*]

This function returns the names of all the Acts in the currently selected graph.

`act-node?` *node act* [*Function*]

This function returns non-nil if Node is either an environment slot or a plot node in Act.

`set-act-node` *name tuple act &key append?* [*Function*]

This function binds Name (the name of a node, either environment or plot) in Act with tuple. If Append?, this function appends Tuple to the present value, otherwise Tuple replaces any existing value.

`get-act-node` *name key act* [*Function*]

This function returns the value of Key for Name in Act.

A.3.1 Accessing Environment Slots

`set-of-environment-nodes` *act* [*Function*]

Returns the set of environment nodes in Act.

`environment-node?` *node act* [*Function*]

Predicate for determining if a node in Act is an environment node.

`act-cue` *act &optional key* [*Function*]

Returns the clauses from Act's cue that match the given key or list of keys.

`set-act-cue` *act &key achieve test conclude comment tuples append?* [*Function*]

Replaces (or appends?) the clauses in Act's cue that correspond to the given keyword arguments.

`clear-act-cue` *act* [*Function*]

Clears all clauses from Act's cue.

`act-setting` *act &optional key* [*Function*]

Returns the clauses from Act's setting that match the given key or list of keys.

`set-act-setting` *act &key test comment tuples append?* [*Function*]

Replaces (or appends?) the clauses in Act's setting that correspond to the given keyword arguments.

`clear-act-setting` *act* [*Function*]

Clears all clauses from Act's setting.

`act-resources` *act &optional key* [*Function*]

Returns the clauses from Act's resources that match the given key or list of keys.

`set-act-resources` *act &key use-resource comment tuples append?* [*Function*]

Replaces (or appends?) the clauses in Act's resources that correspond to the given keyword arguments.

`clear-act-resources` *act* [*Function*]

Clears all clauses from Act's resources.

All functions ending in precondition and "property" are also defined with a plural ending of the same name.

`act-preconditions` *act &optional key* [*Function*]
`act-precondition` *act &optional key* [*Function*]

Returns the clauses from Act's preconditions that match the given key or list of keys.

`set-act-preconditions` *act &key achieve test comment tuples append?* [*Function*]
`set-act-precondition` *act &key achieve test comment tuples append?* [*Function*]

Replaces (or appends?) the clauses in Act's preconditions that correspond to the given keyword arguments.

`clear-act-preconditions` *act* [*Function*]
`clear-act-precondition` *act* [*Function*]

Clears all clauses from Act's preconditions.

`act-properties` *act &optional key* [*Function*]
`act-property` *act &optional key* [*Function*]

Returns the clauses from Act's properties that match the given key or list of keys.

`set-act-properties` *act &key comment tuples append?* [*Function*]
`set-act-property` *act &key comment tuples append?* [*Function*]

Replaces (or appends?) the clauses in Act's properties that correspond to the given keyword arguments.

`clear-act-properties` *act* [*Function*]
`clear-act-property` *act* [*Function*]

Clears all clauses from Act's properties.

`act-comment` *act &optional key* [*Function*]

Returns the clauses from Act's comment that match the given key or list of keys.

`set-act-comment` *act &key comment tuples append?* [*Function*]

Replaces (or appends?) the clauses in Act's comment that correspond to the given keyword arguments.

`clear-act-comment` *act* [*Function*]

Clears all clauses from Act's comment.

A.3.2 Accessing Plot Nodes

`set-of-plot-nodes` *act* [*Function*]

Returns the set of plot nodes in Act.

`plot-node?` *node &optional act* [*Function*]

Predicate to determine if node is a plot-node in Act (by default, the current Act).

`parallel-plot-node?` *node act* [*Function*]

Predicate to determine if node is a parallel plot node in Act.

`conditional-plot-node?` *node act* [*Function*]

Predicate to determine if node is a conditional plot node in Act.

`plot-node` *node act &optional key* [*Function*]

Returns the clauses from the given plot node in Act that match the given key or list of keys.

`set-plot-node` *node act &key achieve achieve-by test conclude retract wait-until require-until use-resource comment tuples append?* [*Function*]

Replaces (or appends?) the clauses in the given plot node in Act that correspond to the given keyword arguments.

`clear-plot-node` *node act* [*Function*]

Clears all clauses from the given plot node in Act.

`destroy-plot-node` *node act* [*Function*]

Destroys the given plot node in Act.

`predecessor-plot-nodes` *node act* [*Function*]

Returns the set of predecessor to node in the plot of Act.

`successor-plot-nodes` *node act* [*Function*]

Returns the set of successors to node in the plot of Act.

`root-plot-node?` *node act* [*Function*]

Predicate to determine if node is a root of the plot in Act.

`end-plot-node?` *node act* [*Function*]

Predicate to determine if node is an end of the plot in Act.

`root-plot-node` *act &optional nodes* [*Function*]

Returns the root node(s) from the plot of Act or from the given plot nodes.

`end-plot-node` *act &optional nodes* [*Function*]

Returns the end node(s) from the plot of Act or from the given plot nodes.

`layout-plot` *&optional acts* [*Function*]

Lays out plot nodes for a set of Acts. A function named Layout-plot-with-joins is the default and seems to be a fairly general layout for Act plots. It works for both horizontal and vertical plots (it uses heuristics to determine which direction) and is used by the system. The layout is compact, without overlapping nodes.

A.4 Dictionaries

`dictionaries` [*Function*]

Returns a list of the dictionaries of the current graph

`set-dictionaries` *dicts* [*Function*]

Sets dictionaries of the current graph to be Dicts.

`set-dictionary` [*Function*]

Denotes current graph to be a dictionary.

`act-dictionary?` [*Function*]

Predicate that determines whether current graph is a dictionary.

`add-dictionary` *&optional current* [*Function*]

Returns default add dictionary of the current graph, if Current return current graph if it is a dictionary.

`set-add-dictionary` *dict* [*Function*]

Sets the default add dictionary of the current graph to be Dict.

`reload-dictionary?` [*Function*]

Returns reload flag of the current graph.

`auto-load-dictionary?` [*Function*]

Returns auto-load flag of the current graph.

A.5 Miscellaneous Functions

`act-pkg` *x* [*Function*]

Takes an expression (symbol, number, list, or dotted pair) and converts it to ACT package.

`set-act-slot-shapes` *&optional mode* [*Function*]

Set shapes of all environment slots to given mode (:button, :top-button, or :default).

`break-string-for-act` *string &optional length* [*Function*]

Returns a list of strings that breaks String into strings of given length.

`verify-acts` *acts* [*Function*]

Invoke Verifier on a single Act or a list of Acts

`change-menu-mode` *mode* [*Function*]

Changes mode of command menu to one of :user, :no-verify, or :hacker

`full-menu-mode` *&optional leave-gui* [*Function*]

Enables all commands in all command menus.

`user-menu-mode` [*Function*]

Disables several commands to produce standard user mode.

`act-prs-mode` [*Function*]

Customize Act-Editor in mode specialized to typical PRS graphs.

`act-sipe-mode` [*Function*]

Customize Act-Editor in mode specialized to typical SIPE-2 graphs.

A.6 Variables

Several variables have already been described in the manual. Variables not described there are briefly mentioned here.

`*plural-classes-ok?` [*Variable*]
If T, check for both singular and plural class names in dictionaries.

`*check-arg-types*` [*Variable*]
If nil, do not check types of arguments in Verifier.

`*schema-type-mode*` [*Variable*]
Mode for creating dictionary schemas, one of :disjunctive or :common-parent.

`*allen-relations*` [*Variable*]
A list of all the legal symbolic Allen relations in Act package.

Index

C

D

E

S

T

U

V

W