Procedural Reasoning System
User's Guide

A Manual for Version 2.0

March 13, 2001

Artificial Intelligence Center
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025

Contents

List of Figures

Chapter 1

Introduction

SRI International's Procedural Reasoning System (PRS) is a framework for constructing real-time reasoning systems that can perform complex tasks in dynamic environments. PRS relies on procedural knowledge representation, which describes how to perform a set of actions in order to fulfill some purpose. For example, we might know that a sequence of actions "*put clothes in washer*", "*put soap in washer*", "*turn washer on*", "*wait 45 minutes*", will achieve the goal "*clean the clothes.*" Much of people's knowledge about how to achieve specific goals is procedural in nature. PRS provides an environment in which this kind of knowledge about actions and goals is readily expressed and executed.

PRS is designed to operate as an embedded execution system in environments that may be highly dynamic. PRS attempts to achieve any goals it might have in light of its current beliefs about the world, while simultaneously reacting to any new events that occur. As such, PRS provides a framework in which goal-directed and event-driven behaviors can be integrated smoothly.

PRS contains several capabilities that make it a powerful system for developing real-time applications. Multiple copies of PRS objects, each referred to as an *agent*, can be run in parallel. Agents operate asynchronously but can communicate through message-passing to solve problems in a distributed, cooperative manner. Another valuable feature of PRS is its support of metalevel reasoning and planning. The metalevel capabilities can be used to implement complex control and scheduling behaviors, as required for individual applications.

The version of PRS described in this document employs a representation of actions called Act, rather than the original PRS representation based on *Knowledge Areas* (also known as *KAs*). As might be expected, these two formalisms have much in common. The Act formalism has two major advantages over the KA formalism, however. One advantage is that the Act syntax is more intuitive and supports certain capabilities not possible with KAs. The second advantage is that knowledge expressed as Acts can be translated into SIPE–2 planning operators[20; 21], making it possible to generate plans automatically for responding to user-specified goals.

Users are encouraged to consult the documents [3; 4; 5; 7]) as well as the WWW site `http://www.ai.sri.com/~prs` for background information on procedural reasoning, PRS-CL, and applications.

1.1 Overview of PRS

The architecture of PRS is depicted in Figure 1.1. PRS consists of (1) a *Database* containing current *beliefs* or facts about the world; (2) a set of current *Goals* to be realized; (3) a set of plans, called *Acts*, describing how sequences of actions and tests may be performed to achieve certain goals or to react to particular situations; and (4) *Intentions* containing those plans that have been chosen for (eventual) execution. An

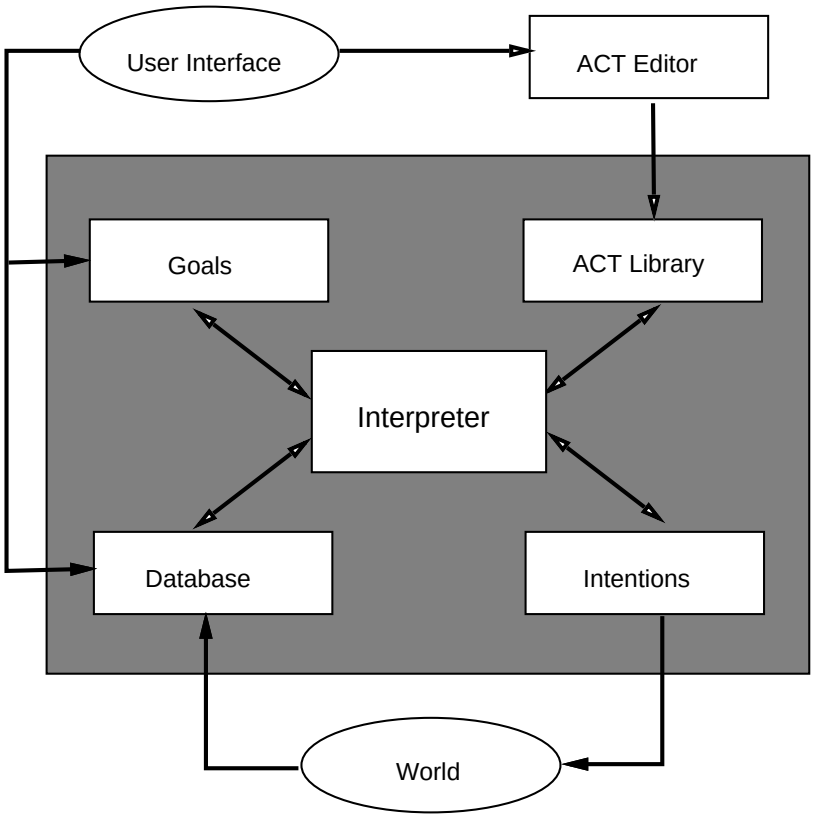

Figure 1.1: Architecture of a PRS Agent

Interpreter manipulates these components, selecting appropriate Acts for execution based on the system's beliefs and goals, creating the corresponding intentions, and then executing them.

The system interacts with its environment, including other systems, through its database (which acquires new beliefs in response to changes in the environment) and through the actions that it performs as it carries out its intentions.

1.1.1 The Database

The contents of the PRS database may be viewed as the system's current beliefs about the world. The knowledge contained in the database is represented in a restricted form of first-order predicate calculus.

The database generally contains both *static* and *dynamic* information about a domain. Static information describes fixed properties about the application domain, such as the structure of some subsystems or the physical laws that must be obeyed by certain mechanical components, and remains in the database throughout the lifetime of the system. Dynamic information changes over time, such as current observations about the world and conclusions derived by the system from these observations.

Database facts can also describe the internal state of PRS itself. Such facts are referred to as *metalevel facts* (or, *meta-facts*). Metalevel facts can be used to describe the current goals and intentions of the system, as well as the Acts being considered for execution. Metalevel facts play an important role in specifying alternative control strategies for PRS, as described in Chapter 9.

1.1.2 Goals

Goals are expressed as conditions over an interval of time (i.e., over a sequence of world states) and are specified as a combination of a *goal operator* applied to a *logical formula*. The accepted goal operators are:

(ACHIEVE C) achieve a certain condition

(ACHIEVE-BY C (A1 ... An)) achieve a certain condition using a restricted set of Acts

(TEST C) test for the condition

(USE-RESOURCE R) allocate a resource

(WAIT-UNTIL C) wait until the condition is true

(REQUIRE-UNTIL G C) check that goal G stays true until condition C is satisfied

(CONCLUDE P) add P to the database

(RETRACT P) retract P from the database

As an example, the goal to close valve `v1` could be represented as `(ACHIEVE (position v1 closed))`, and the goal to test for it being closed as `(TEST (position v1 closed))`. A given action (or sequence of actions) is said to *succeed* in satisfying a goal if its execution results in a state where the terms of the goal are met.

As with database facts, goal descriptions are not restricted to specifying desired behaviors of the external environment but can also characterize the internal behavior of the system. Such descriptions are called *metalevel goals*.

1.1.3 The Act Library

Knowledge about how to accomplish goals or how to react to certain situations is represented in PRS by declarative procedure specifications called *Acts*. Each Act consists of a *plot*, which describes the steps of the procedure, and an *environment*, which specifies the conditions for which the Act can be applied. Together, the environment and the plot of an Act specify declaratively how a set of actions can be used to respond to a goal or event in certain situations. Sample Acts are provided in Appendix A.3.3 (page 57).

The plot of an act can be viewed as a plan or plan schema. It is represented as a graph with one distinguished start node and possibly multiple end nodes. Nodes in the graph are labeled with subgoals to be achieved in carrying out the plan. Successful execution of an act consists of achieving each of the subgoals on the nodes of a path from the start node to an end node. This formalism allows richer control constructs (including conditional selection, iteration, and recursion) than most plan representations.

Each PRS application contains Acts from two sources: Acts supplied by the user, (these constitute the procedural knowledge for the domain), and a set of predefined *default Acts* that are built into the system itself. User-specified Acts may include both Acts that pertain to the application domain, and metalevel Acts that manipulate the beliefs, goals, and intentions of PRS. Metalevel Acts can be used to encode actions that influence the operation of the system itself, such as methods for choosing among multiple applicable Acts, modifying intentions, or computing the amount of reasoning that can be undertaken given the real-time constraints of the problem domain.

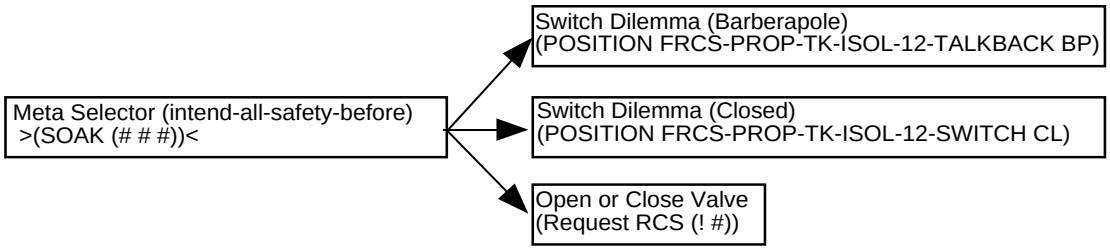

Figure 1.2: Example of an Intention Graph

1.1.4 Intentions

An intention corresponds to a task to be performed by the system in response to some posted goal or fact. A single intention consists of some initial Act together with all the 'sub-Acts' being applied to satisfy the subgoals of the original Act. Many intentions may be active simultaneously, some of which may be suspended or deferred, and some of which may be waiting for certain conditions to hold prior to activation. The set of active intentions are stored in a structure called the *intention graph*.

The intention graph defines a partial ordering of the intentions, with possibly multiple least elements (called *roots*). An intention earlier in the ordering must be either realized or dropped (and thus disappear from the intention graph) before intentions appearing later in the ordering can be executed. This precedence ordering enables the system to support prioritized execution of intentions.

Figure 1.2 shows a schematic of an intention graph from an application of PRS to perform fault diagnosis in the Reaction Control System of the NASA Space Shuttle. In handling a malfunction, the system might have, at some instant, four tasks (intentions) in the intention graph. Two intentions address problems with individual switches, while one responds to a request from a user to achieve the goal of closing a valve. In this case, the root intention corresponds to a meta-act being executed to decide which way to accomplish the valve-closing goal, and whether to schedule its execution ahead of the other intentions.

1.1.5 The Interpreter

The PRS interpreter runs the entire system. The interpreter repeatedly executes the set of activities depicted in Figure 1.3.

At any particular time, certain goals are established and certain events occur that alter the beliefs held in the system database (1). These changes in the system's goals and beliefs trigger (invoke) various Acts (2). One or more of these applicable Acts will then be chosen and placed on the intention graph (3). Finally, PRS selects a task (intention) from the root of the intention graph (4) and executes *one step* of that task (5). This will result either in the performance of a primitive action in the world (6), the establishment of a new subgoal or the conclusion of some new belief (7), or a modification to the intention graph itself (8).

At this point the interpreter cycle begins again: the newly established goals and beliefs (if any) trigger new Acts, one or more of these are selected and placed on the intention graph, and again an intention is selected from that structure and partially executed.

Execution of primitive actions can effect not only the external world but also the internal state of the system. For example, an action may operate directly on the beliefs, goals, and intentions of the system. Alternatively, the action may indirectly affect the system's state as a result of the knowledge gained by its interaction with the external world.

Unless some new belief or goal activates a new Act, PRS will try to fulfill any intentions it has previously decided upon. This results in focused, goal-directed reasoning in which Acts are expanded in a manner analogous to the execution of subroutines in procedural programming systems. If some important new

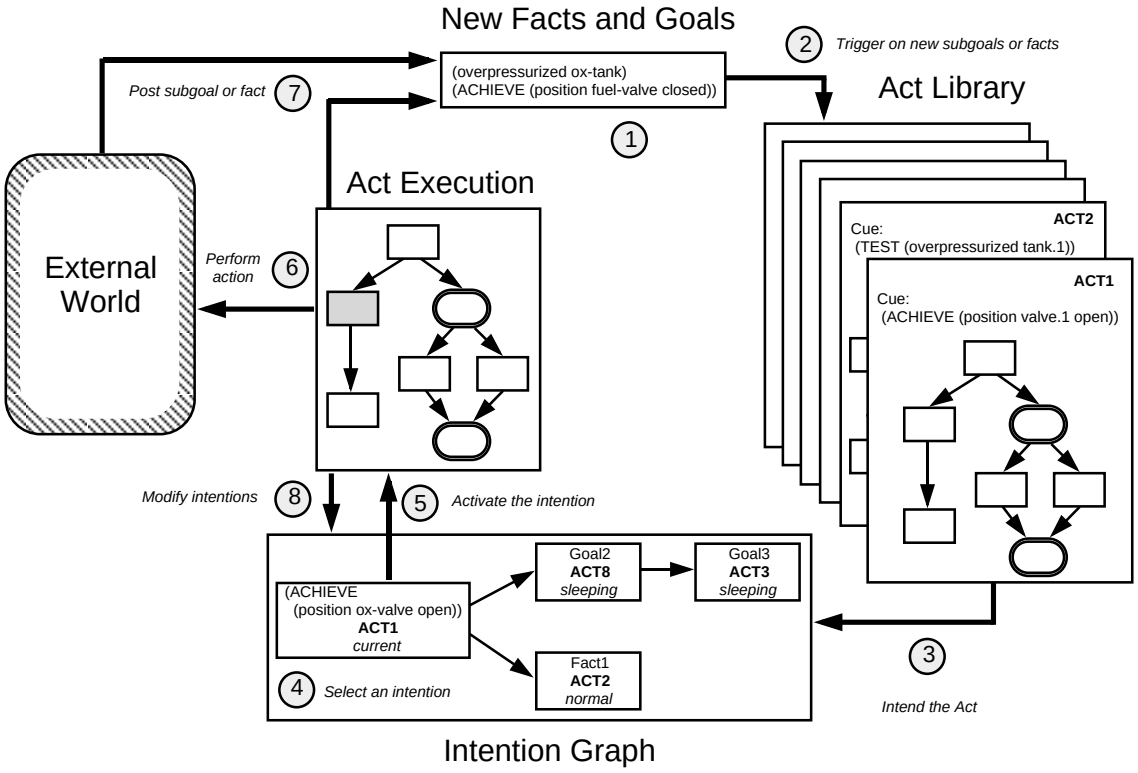

Figure 1.3: The Interpreter Loop

fact or goal does become known, PRS will reassess its current intentions, and perhaps choose to work on something else. Thus, not all options that are considered by PRS arise as a result of means-end reasoning. Changes in the environment may lead to changes in the system's goals or beliefs, which in turn may result in the consideration of new plans that are not means to any already intended end. PRS is therefore able to change its focus completely and pursue new goals when the situation warrants it. In many applications, such switches happen frequently as emergencies of various degrees arise in the process of handling less critical tasks.

1.2 Using PRS

There are two stages involved in applying PRS to a particular problem domain: *specifying* the domain knowledge and *running* PRS to perform the tasks specific to that application.

To build a PRS application, a user must supply three types of domain knowledge. First of all, a set of facts or beliefs describing the relevant properties of the application domain and the initial state of the world is required. This information will be loaded into the PRS database. Second, the user must supply a set of Acts that defines the procedural knowledge of the domain. Finally, the user must provide any domain-specific functions that may be required during the course of execution. PRS stores its domain knowledge in a set of application files, with different types of files for the three types of knowledge. Details on how to set up these files are provided in Chapter 5.

To begin operation of PRS, the user simply creates the desired PRS agents and loads the relevant application files. PRS then begins running, waiting for events (such as a change in the environment or the acquisition of some goal) to trigger the execution of some Acts.

PRS is designed to operate as an embedded reasoning system; that is, at run time, the system expects to be situated in a changing environment. This environment can best be thought of as a number of pro-

cesses with which PRS interacts by sending and receiving messages. One such "process" is the user. Other processes may include processors controlling sensors and effectors in the actual application domain, application simulators, other reasoning systems, or other computer systems. During operation, PRS updates its database of facts as it receives information (messages) from the processes comprising its environment.

The version of PRS described in this manual has a graphical user interface built on top of Grasper-CL[17; 16]. Grasper-CL is a programming-language extension to LISP that introduces graphs — arbitrarily connected networks — as a primitive data type. Grasper-CL supports both the graphical display of Acts and a menu-driven interface for interacting with PRS. Figure 6.1 (page 26) provides an illustration of the graphical interface for PRS.

1.3 Terminology: Act vs KA

As noted at the outset of this chapter, the version of PRS described in this document makes use of the Act language for representing procedural knowledge rather than the KA language. More accurately, Act serves as the input language for procedural knowledge in this version of the system but the KA representation is still used internally by PRS. The translation from Act to KA takes place when the user loads his or her Acts into the PRS system.

For the most part, the internal manipulation of KAs by PRS is invisible to the user. However, some PRS functions that are used in defining meta-acts still refer to KAs, since they must access the run-time structures that encode the procedural knowledge. References to these KA-based functions appear throughout this manual. As well, the user may encounter references to these functions while debugging PRS applications.

1.4 Getting Started

Several simple demonstration applications are available as an aid to help new users familiarize themselves with PRS concepts. These applications are described in Appendix A. It is highly recommended that users experiment with these demonstrations prior to undertaking the development of their own applications.

Chapter 2

Act Representation

Acts are used to encode the possible activities that the system can undertake. They are represented graphically, as illustrated in Figure 2.1. Acts can be created using an interactive graphical editing tool called the Act-Editor. For information on the use of the Act-Editor, consult *The Act Editor User's Guide* [19], available at the URL `http://www.ai.sri.com/~act/act-ed.ps`.

Note: The Act formalism is used as a common representation language for several different action-based problem-solving systems. To support capabilities required in these other systems, extensions are periodically made to Act. As a result, users may encounter Act-related documents that include constructs not described in this manual; these capabilities are not relevant to PRS-CL.

An Act 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*. The environment conditions appear along the left-hand border of the act (see Figure 2.1). Action specifications are called the *plot* of an Act, and consist of a partially ordered set of actions and subgoals. The plot is represented graphically as a directed network of nodes.

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

Individual PRS applications will require domain-specific Acts written by the users. In addition, PRS contains a set of default Acts which are loaded when the entire system is loaded. The default Acts are summarized in Appendix B.

This chapter begins with an overview of the goal expressions supported in the Act language. Environment conditions and plots are next described, followed by an explanation of variable usage in Acts.

2.1 Goal Expressions

Goal expressions describe desired states for PRS, expressed as conditions on the current goals and beliefs of the system. They consist of an Act goal operator applied to a logical formula. Goal expression are used both to specify applicability conditions for environment slots and subgoals for plot nodes.

The logical formulas allowed in goal expression are defined by the following BNF grammar. Here, the form on the left of the symbol `::=` can be replaced by the form on the right. Items in the typewriter font (e.g., `OR` and `AND`) represent actual primitives for the grammar while italicized text (e.g., *s-expression*) defines primitives descriptively. Square brackets [ ] are placed around optional objects. The symbol | represents "or", the symbol * represents any number of repetitions including zero, and the symbol +

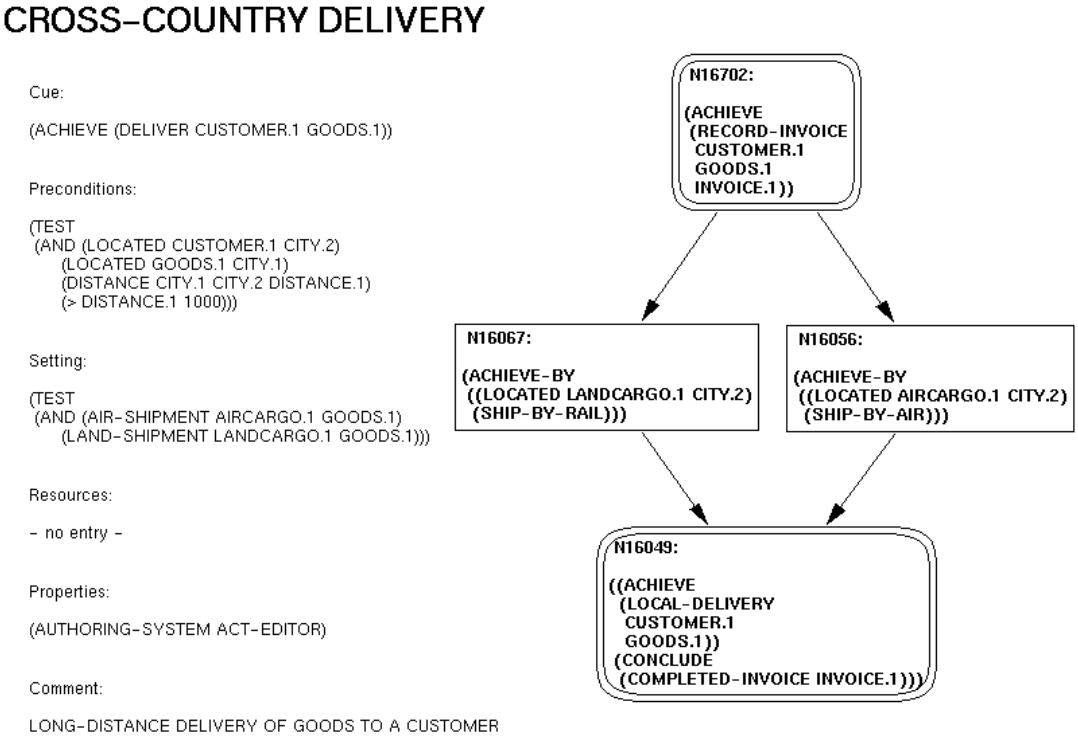

Figure 2.1: The Act `Cross-Country Delivery`

represents any number of repetitions greater than one. Braces {} without * or + appended simply indicate grouping.

Logical Formulas

wff	::=	(pred-name {term}*) | `(NOT` wff`)` | `(AND` {wff}+`)` | `(OR` {wff}+`)`
term	::=	simple-term | function | `(REBIND` variable `)`
simple-term	::=	individual | variable
variable	::=	{class} . {integer}
function	::=	(fn-name {term}*)
pred-name	::=	*the name of a predicate*
fn-name	::=	*the name of a function*
individual	::=	*a domain object*
class	::=	*the name of a class*
integer	::=	*a positive integer*

The above syntax for logical formulas is mostly standard, although the treatment of variables requires some further explanation. PRS supports the use of *typed variables*. In particular, the name of the variable indicates a class/type to which any instantiations of the variable must belong; for example, `airplane.1` is a variable for objects in the class *airplane*. Type information will be ignored in PRS unless an explicit type-checking function is supplied, as discussed elsewhere. The `REBIND` function, which appears in the specification of terms above, is somewhat nonstandard; it is discussed in Section 2.4.

Act goal expressions are built by applying a *goal operator* to a logical formula. There are seven goal

operators in the Act language: `TEST`, `ACHIEVE`, `ACHIEVE-BY`, `CONCLUDE`, `USE-RESOURCE`, `WAIT-UNTIL`, and `REQUIRE-UNTIL`. Goal expressions adhere to the following syntax.

Goal Expressions

goal-expn ::= test | conclude | achieve | achieve-by
| use-resource | wait-until | require-until

test ::= (TEST {wff | wff-list})
achieve ::= (ACHIEVE {wff | wff-list })
achieve-by ::= (ACHIEVE-BY {wff+acts | ({wff+acts}$^+$))
conclude ::= (CONCLUDE {wff | wff-list})
use-resource ::= (USE-RESOURCE {simple-term | ({simple-term}$^+$) })
wait-until ::= (WAIT-UNTIL {wff | wff-list })
require-until ::= (REQUIRE-UNTIL { wff | wff-pair })
wff-pair ::= (wff wff)
wff-list ::= ({wff}$^+$)
wff+acts ::= (wff ({act}$^+$))
act ::= *the name of an Act*

`TEST` goals specify a formula or list of formulas that must be true in the current database/world. `USE-RESOURCE` goals designate a set of resources required by the Act, and hence that must be available for the Act to be executed. `ACHIEVE` goals direct the system to accomplish some task by any means possible; `ACHIEVE-BY` goals are similar but specify a restricted set of Acts that can be used to accomplish the task. `WAIT-UNTIL` goals direct the system to wait until some specified condition holds. `REQUIRE-UNTIL` goals designate conditions that must be maintained over a specified interval, and `CONCLUDE` goals specify information to be added to the database.

For goal expressions, `OR` is equivalent to the first disjunct that succeeds (as in Lisp), rather than a logical disjunction that considers all disjuncts. Thus, for example, in evaluating the goal expression `(TEST (OR (A) (B) (C)))`, PRS will first determine whether `(A)` is true, if not then it will test `(B)`, if not then it will test `(C)`.

2.2 Act Environment

The Act environment conditions are defined as a series of fixed *slots*, listed in Table 2.1.

The Name and Comment slots are straightforward. The Properties slot provides a means to assign properties/attributes to an Act. The slots Cue, Precondition, Setting, and Resources, specify conditions that must be *satisfied* in order for the Act to be applicable in a given situation. For this reason, these slots are referred to as the *gating conditions* for an Act. The gating slots are filled with one or more goal expressions, which describe requirements on current goals and facts. The environment slots are discussed in detail below.

Name The name is a symbol that uniquely identifies the Act.

Comment The comment is a string that provides documentation.

Cue The Cue indicates the purpose of an Act. It is also used to index and retrieve Acts for possible execution. The Cue can contain either an `ACHIEVE`, `TEST` or `CONCLUDE` goal. An `ACHIEVE` goal indicates that the Act can be used to achieve some condition. A `TEST` goal indicates that the Act can be used to *actively* test some condition (as opposed to simply looking in the database). A goal

Environment Slot	Role
Name	identifier
Cue	purpose of the Act
Preconditions	situational conditions
Setting	bindings for variables
Resources	resources to allocate
Properties	user-defined characteristics
Comment	documentation

Table 2.1: Environment Slots and their Roles

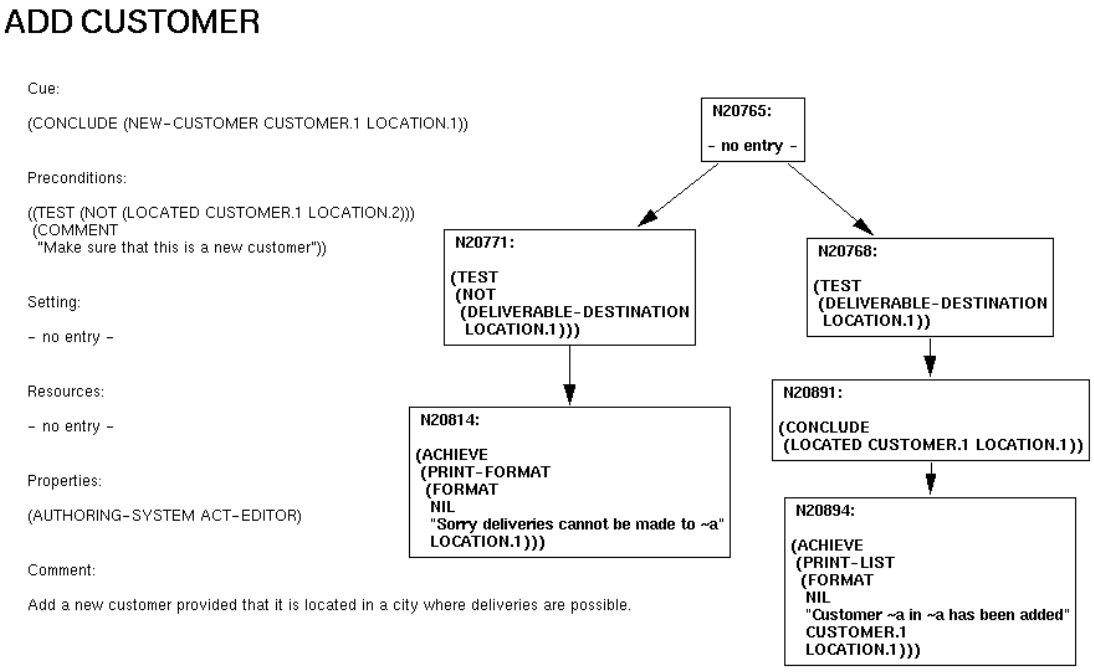

Figure 2.2: The Act `Add Customer`

`(CONCLUDE (P))` indicates that the Act can/should be invoked when `(P)` is added to the database. A Cue containing a `CONCLUDE` goal can thus be used to respond/react to events that arise during the course of execution. (For example, the Act `Add Customer` in Figure 2.2 can be used to take appropriate actions when a fact of the form `(NEW-CUSTOMER <customer> <location>)` is added to the PRS database).

An Act whose Cue contains an `ACHIEVE` or `TEST` goal is said to be *goal-invoked* while an Act whose Cue contains a `CONCLUDE` goal is said to be *fact-invoked*. Cue specifications should be short so that the system can rapidly identify a subset of potentially applicable Acts for a new goal or fact.

Precondition The Precondition slot specifies situational constraints that must be satisfied for the Act to be applicable. It can contain both `ACHIEVE` and `TEST` goal expressions. The meaning of `(ACHIEVE G)` in this context is that the system must currently have `G` as a goal in order for the Act to be applied. The meaning of `(TEST P)` is that `P` must be true in order for the Act to be applied.

Setting The Setting specifies additional ACHIEVE and TEST constraints for the applicability of an Act. This slot is equivalent functionally to the Precondition slot but typically is used to separate out those conditions that relate to instantiating variables.

Resources The Resources slot indicates resources that are to be allocated for the duration of the Act. This slot can be filled only with USE-RESOURCE goals. Resources are treated as non-sharable objects. In order for an Act containing an expression of the form (USE-RESOURCE (A B C)) in its Resources slot to be applicable, the resources (A B C) must not currently be allocated to another Act. These resources would then be unavailable for use by other processes until execution of the Act finishes.

Properties The properties slot is a list of property/value pairs. Certain properties have special significance in PRS. For example, the property (DECISION-PROCEDURE T) designates an Act that reorders the intention graph (as described further in Chapter 9). Other properties are recommended, such as AUTHORING-SYSTEM, which designates the system used to create the Act (usually the Act-Editor), and AUTHOR. The user is free to supply additional properties, as desired.

The properties of an Act are generally context-independent. At times, however, it is convenient to have properties whose values depend on the situation in which an Act is invoked. The Act formalism supports the use of property values that are defined in terms of variables from the gating environment conditions. Property values can even include evaluable forms, as in the following property list:

```
((SAFETY-HANDLER T)
 (PROBLEM-ID PID.1)
 (PRIORITY (+ 1 (URGENCY-OF Y.1))))
```

Here, the value of the property PROBLEM-ID is whatever the variable PID.1 is bound to by the gating conditions for the Act. Similarly, the value for the property PRIORITY is set to be one greater than the value returned by the function URGENCY-OF, assuming that this function has been declared as *evaluable* (see Chapter 3 for details).

2.3 Act Plot

The plot of an Act consists of a directed graph, whose nodes represent actions and whose arcs impose a partial temporal order for execution. 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 Act Iterative Factorial in Figure 2.3.

Associated with each plot node is a list of goal expressions that specify a set of actions for the node. All Act goal operators except USE-RESOURCE can be used on plot nodes. However, at most one goal expression on each node can be built using the same operator. Furthermore, ACHIEVE and ACHIEVE-BY goals are mutually exclusive: if one is used on a node, the other is prohibited. Empty plot nodes are permitted.

On plot nodes, the CONCLUDE operator should not be applied to either an explicit disjunction such as (OR P_1 ...P_k), or an explicit disjunction embedded within a conjunction such as (AND (Q) (OR P_1 ...P_k)). This restriction is necessary because (a) PRS does not support the insertion of disjunctive facts into its database, and (b) when given a conjunctive fact to be concluded, PRS adds the individual conjuncts into the database (see Chapter 4 for details).

Execution of a plot requires successful execution of all nodes along some path from the start node to some terminal node. Successful execution of a node requires satisfaction of all of 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 N16702 and N16049

are parallel, while nodes `N16067` and `N16056` are conditional. The type of a node influences the meaning of the arcs that enter and leave the node.

Arcs coming into and going out of a parallel node are *conjunctive*, meaning that all of the arcs need to be executed. For the plot in Figure 2.1, both branches emanating from `N16702` would be executed, and execution of `N16049` cannot begin until both of its incoming branches have completed.

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 (referred to as a *split* node). PRS will execute individual successor nodes of the split node until it finds one whose goals are satisfied. At that point, it 'commits' to the branch headed by that successor node and ignores all other branches from the split node. For a disjunctive node with multiple incoming arcs (referred to as a *join* node), the node can be executed as soon as one of its ancestor nodes has been successfully executed. As an example, consider the act `Iterative Factorial` in Figure 2.3. After execution of the split node `N11817`, PRS will nondeterministically choose one of its successor nodes for execution. If the goal expression on its choice is satisfiable, then it will continue executing that branch. If not, it will try to satisfy the other node. In this case one of the two nodes will succeed. In general, that is not the case: if a split node has no successors that can be satisfied, then the split node is said to fail.

Two consequences of the typing conventions should be noted: (1) If a node has zero or one incoming edges and zero or one outgoing edge, it is irrelevant whether it is a conditional or a parallel node. (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. (3) Conversely, if one action is do be activated by all of its incoming edges and must only activate one of its outgoing edges, it must be represented by a parallel node that collects all the incoming edges followed by a conditional node that collects the outgoing edges.

2.4 Variables

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

It is sometimes necessary to change the bindings of variables within Act plots. One such situation arises when trying to write 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 of a gauge) and to take certain steps when the value crosses some threshold. Acts of these types are difficult to write using only logical variables.

For this reason, the Act language supports rebinding of variables during execution of an Act, provided the user explicitly specifies 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 very different from

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

which seeks to equate a value with a value that is 1 larger, and hence will always fail.

Variables that appear within the scope of a `REBIND` function are referred to as *dynamic* variables. The Act `Iterative Factorial` in Figure 2.3 illustrates how the `REBIND` function and dynamic variables can be used to specify plots with loops.

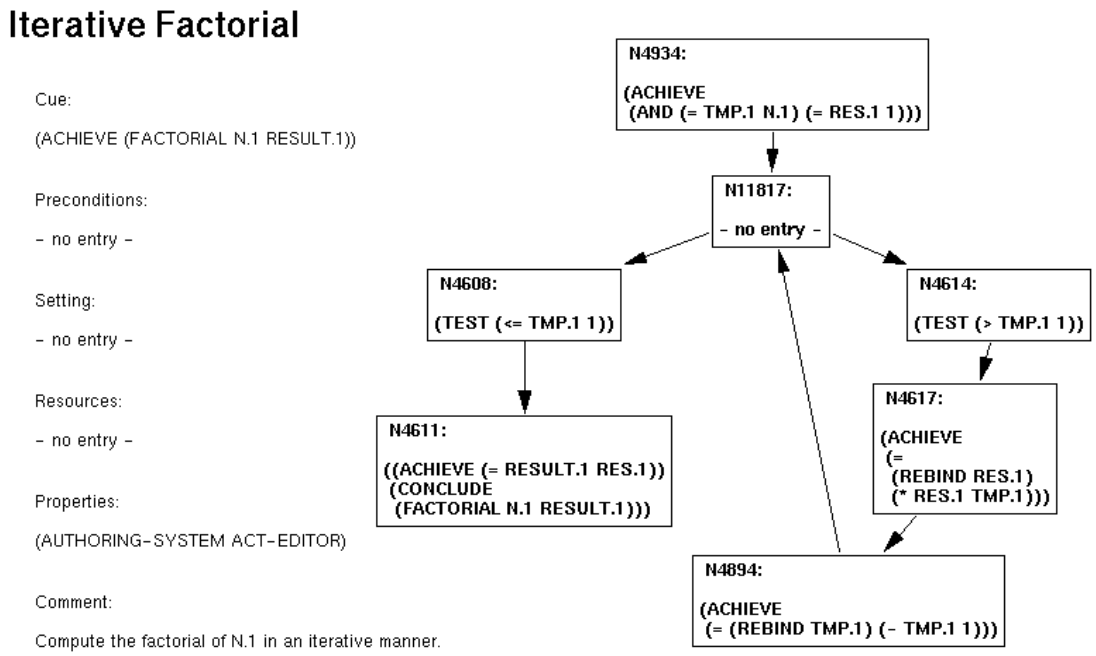

Figure 2.3: The Act `Iterative Factorial`

The `REBIND` function can be used only in plot nodes and only for variables that do not appear in the gating environment conditions. In addition, the `REBIND` function can only be used for goals of the form `(ACHIEVE (= (REBIND X.1) <expn>))`.

2.5 From ACTs to Actions

As described above, Acts contain goal expressions that describe objectives to be met. How are these goal expressions mapped to actions that will satisfy the goals? PRS contains three different types of basic activities that provide the underlying actions that the system can take: *primitive actions*, *evaluable predicates* and *evaluable functions*.

Primitive Actions

A primitive action is a LISP function that can be used to satisfy an `ACHIEVE` goal. A single primitive action can be associated with any user-defined predicate; in addition, there are a number of predefined primitive actions within PRS. A predicate with an associated primitive action is referred to as a *primitive action predicate*, or sometimes just a primitive action (for short).

When an `ACHIEVE` goal with a primitive action predicate is during execution, PRS will immediately evaluate the Lisp function rather than searching for Acts to satisfy the predicate. In this way, goals can

be mapped to actions for satisfying those goals. PRS includes a number of predefined primitive actions. One example is `(PRINT-LIST <format-stmt>)`, which can be used to print a LISP format statement.

To associate a primitive action with a predicate, it is necessary to define the Lisp function that serves as the primitive action and to declare the primitive action. By convention, the primitive action predicate and its corresponding LISP function must share the same name. Declarations are made using the following function call:

`(PRIMITIVE-ACTIONS <list-of-names-of-primitive-actions>)`

This function can be called either directly in Lisp or can be added to the *functions file* to be loaded initially by the system (see section 5.1.3 for details). The Lisp variable `*PRIMITIVE-ACTIONS*` is set to a list of primitive action predicates defined within the system at any point in time.

The value returned by a primitive action is important: if it returns any expression other than NIL, then the `ACHIEVE` goal is considered to be successful. Primitive actions should not return the value `:wait`, as it is a keyword with special meaning in PRS. Primitive actions should be used only when their arguments are guaranteed to be bound. The set of predefined primitive actions are described in Appendix D.

Evaluable Predicates

Evaluable predicates are similar to primitive action predicates in that they allow a function to be assigned to a predicate. They differ, however, in the manner in which those functions are used.

While primitive action functions are only invoked for a predicate in the scope of an `ACHIEVE` goal, functions for evaluable predicates are invoked for predicates appearing within any goal operator. However, because of the manner in which evaluable predicates are implemented *they should not have any side-effects.* In contrast, primitive actions are designed specifically to handle functions that change the world in some way (such as the `PRINT-LIST` primitive action described above).

As an example, the Act `Iterative Factorial` in Figure 2.3 makes use of the evaluable predicates $<=$ and $>$. These two predicates are predefined in PRS as evaluable predicates. Chapter 3 describes how to declare domain-specific evaluable predicates.

Evaluable Functions

Evaluable functions are similar to evaluable predicates. By making appropriate declarations, a user can associate a LISP function with any PRS logical function. Whenever the logical function is encountered during execution, the corresponding LISP function is used to return a value for the logical expression. As with evaluable predicates, evaluable functions should not have side-effects. Chapter 3 describes how to declare domain-specific evaluable functions.

The functions associated with primitive actions, evaluable predicates, and evaluable functions constitute the set of basic activities that the system can perform. The time taken to execute these functions influences the granularity of the real-time behavior of the system. PRS will not interrupt any of these basic actions; as such, it cannot respond to any new facts or goals until such an action has completed. If PRS is required to operate as a real-time system (and thus be able to furnish a guaranteed bound on reaction time), it is *critical* that any basic actions be guaranteed to return after some bounded (and preferably short) interval of time. Actions that are too complex to satisfy this requirement should be implemented as separate processes that communicate with PRS by message-passing.

Chapter 3

Attributes of Predicates and Functions

In PRS, it is possible to declare that predicates have certain attributes or properties. Predicates can be declared as *closed-world*, *functional* or *basic events*. In addition, both predicates and functions can be declared as *evaluable*. Declarations are made using attribute-specific Lisp function calls, as described below. These function calls can be made directly in Lisp or can be inserted in the *functions file* (see section 5.1.3) to be loaded during initialization of PRS.

3.1 Evaluable Predicates and Functions

As noted in Chapter 2.5, logical predicates can be assigned a Lisp function that executes actions intended to satisfy the predicate. When such a predicate is encountered within a `TEST` or `ACHIEVE` goal during execution, PRS will immediately evaluate the function rather than searching for Acts that can test or achieve the predicate. Such a predicate is said to be *evaluable*. In this way, goals can be mapped to actions for satisfying those goals.

To make a predicate evaluable, it is necessary to define a Lisp function with the same name as the predicate and declare that the predicate is evaluable. Declarations are made using the following function call, which adds new evaluable predicates to those currently declared:

```
(EVALUABLE-PREDICATES <list-of-names-of-evaluable-predicates>)
```

The Lisp variable `*EVALUABLE-PREDICATES*` contains a list of evaluable predicates within the system.

The value returned by an evaluable predicate is important: if it returns any expression other than NIL, then the `TEST` or `ACHIEVE` goal is considered to be successful. Evaluable predicates should not return the value `:wait`, as it is a keyword with special meaning to PRS.

PRS also supports the use of *evaluable functions*. Whenever a logical function that has been declared evaluable is encountered by PRS, the Lisp function with the same name is directly executed. To add new evaluable functions you should define the associated Lisp function and declare the function as evaluable using the following function:

```
(EVALUABLE-FUNCTIONS <list-of-names-of-evaluable-functions>)
```

The Lisp variable `*EVALUABLE-FUNCTIONS*` contains a list of evaluable functions within the system.

Evaluable predicates and evaluable functions are required to have the same name as their associated Lisp function. Evaluable functions and predicates in PRS *do not evaluate* their arguments, unless they are evaluable functions themselves. Evaluable and predicates should be used only when their arguments are guaranteed to be bound. Each PRS system includes a set of predefined evaluable predicates and functions, which are described in Appendix D (page 67).

3.2 Closed-World Predicates

A predicate can be declared as satisfying the *closed world assumption*, meaning that if any instance of the predicate is not found in the database, then the negation of the instance is assumed to be true. The closed world assumption is very useful when dealing with predicates that have many negative instances: rather than having to explicitly add all of the negative instances to the database, the closed-world declaration represents them implicitly. Closed-world declarations are made using the function:

```
(CLOSED-WORLD-PREDICATES <list-of-cw-predicates>)
```

The variable `*CLOSED-WORLD-PREDICATES*` stores a list of the closed-world predicates known within the system.

3.3 Functional Predicates

Predicates are often used to represent relations that are *functional* with respect to some subset of their arguments. Functionality means that for that subset of arguments, there is only one set of bindings for the remaining arguments that will make the predicate hold. For example, consider the relationship of fatherhood as expressed by the predicate `(FATHER <child> <man>)`. This predicate is functional in its first argument, since for any child there is at most one man who can be his or her father. More generally, for some predicate P the following relationship of functionality may hold:

$$((P\ x_1 ... x_n) \wedge (P\ y_1 ... y_n) \wedge x_1 = y_1 \wedge \ldots x_j = y_j) \supset x_{j+1} = y_{j+1} \wedge \ldots \wedge x_n = y_n$$

In this case, P is functional with respect to its first j arguments.

To declare predicates as functional, use the following function, which indicates that each `<predicate>` is functional in its first `<j>` arguments:

```
(FUNCTIONAL-PREDICATES '((<predicate> .  <j>) ...(<predicate> .  <j>)))
```

Note that the arguments have to be ordered so that this property holds of the *first* j arguments, rather than some other ordering of arguments (such as the *last* j arguments). The variable `*FUNCTIONAL-PREDICATES*` stores the set of declared functional predicates in the system.

3.4 Basic Events

Certain predicates are used to describe *basic events* within the system, corresponding to low-level operations within PRS. For example, the `SOAK` predicate (see Chapter 9) describes the set of currently applicable Acts/KAs. Typically, these facts are relevant only to the current cycle and may be used to trigger Acts or awaken some sleeping intention. For the sake of efficiency, it is useful to remove them from the database after the cycle in which they were added. A number of built-in predicates to PRS are declared as basic events by default. The user can declare additional predicates as basic events, using the following function:

```
(BASIC-EVENTS '((<predicate> .  <arity>) ...(<predicate> .  <arity>)))
```

The set of basic events is stored in the variable `*BASIC-EVENTS*`.

Chapter 4

The Database

The PRS database contains an agent's current information about the application domain. Information can be added or removed from the database using menu commands from the graphical interface, or by actions taken directly by the system (such as execution of `CONCLUDE` goals or the sending of a message by another process).

It is also possible to load database files directly into an agent. These files use Lisp syntax but do not contain evaluable Lisp expressions (i.e., the expressions are not evaluable by the Lisp interpreter, thus the file should not be compiled). The database filename should have the form `<name>.LISP`. When a database file is loaded, each fact in the file will be inserted into the database *without* checking to see if it makes any new Acts relevant. In contrast, when facts are concluded into the database in the normal course of running PRS (either via the CONCLUDE operator or via message passing) the system will check for new relevant Acts after adding the fact to the database.

Each fact in the database file should be a well-formed formula (or *wff*) that conforms to the following syntax. The set of formulas supported by the database is a strict subset of those used in goal expressions, but shares the same meanings.

Database Facts

wff ::= (pred-name {term}*) | `(NOT` wff`)` | `(AND` {wff}+`)`
term ::= function | individual | variable
variable ::= {class} . {integer}
function ::= (fn-name {term}*)

pred-name ::= *the name of a predicate*
fn-name ::= *the name of a function*
individual ::= *a domain object*
class ::= *the name of a class*
integer ::= *a positive integer*

All variables are interpreted as being universally quantified.

The database does not serve as a true *knowledge base* in the sense that it does not provide complete truth maintenance. The truth maintenance procedures used by the database will ensure only that there are not inconsistent literals: for any ground literal `(P)`, both `(P)` and `(NOT (P))` cannot be in the database at the same time. When one is in the database and the other is added, the original will be removed automatically from the database. The same does not hold for compound formulas (containing `AND` or `OR`) nor for non-ground literals. Thus, it is possible to have a database that is 'contradictory' in the sense that its contents are mutually inconsistent. For example, the facts `(NOT (AND (A) (B)) )`, `(A)` and `(B)` would

not be recognized as mutually inconsistent. If desired, the user can provide metalevel Acts that perform truth-maintenance for these more general cases.

The database does not implement `OR` as the logical disjunctive operator. For this reason, the database does not support the addition of either explicit disjunctions such as `(OR` P_1 `...`P_k`)`, or explicit disjunctions embedded within a conjunction such as `(AND (Q) (OR` P_1 `...`P_k`))`. PRS will prevent the user from directly adding such facts to the database. (NOTE: PRS will only flag the use of explicit disjunctions written using the operator `OR`. Implicit disjunctions such as `(NOT (AND (P) (Q)))` are not currently recognized.)

Chapter 5

Creating a PRS Application

Domain knowledge for a PRS application is typically stored in a collection of application files. These files come in three types. The *database files* describes the set of initial database facts or beliefs relevant to the application. The *act files* describe the application-specific Acts to be used by the system. The *functions file* contains any user-defined Lisp functions.

5.1 Application Files

To build a PRS application, a user must supply three types of domain knowledge. First of all, a set of facts or beliefs describing the relevant properties of the application domain and the initial state of the world is required. This information will be loaded into the PRS database. Second, the user must supply a set of Acts that defines the procedural knowledge of the domain. Finally, the user must provide any domain-specific functions that may be required during the course of execution. PRS stores its domain knowledge in a set of application files, with different files being used for the different types of knowledge.

5.1.1 Act Files

Individual PRS applications require domain-specific Acts written by the users. The library of application Acts is built using the Act-Editor and stored as one or more files called *graphs*. (The label *graph* is derived from Grasper-CL terminology.) Act graphs have names of the form `<name>.graph`. An individual agent can have any number of Act files associated with it.

5.1.2 Database Files

Database files can be used to specify information to be loaded directly into an agent's database. An agent can have any number of database files. A database file should contain only valid database expressions (as defined in Chapter 4). When a database file is loaded, each fact in the file will be inserted into the database *without* checking to see if it makes any new Acts relevant. A database file must have a name of the form `<name>.LISP`.

5.1.3 Functions Files

A *functions file* contains Lisp forms to be evaluated. One important class of forms are the user-defined functions required for an individual application. These will typically include the primitive actions and evaluable functions and predicates of the domain, as well as any auxiliary functions that are desired. Typically, *attribute declarations* for predicates and functions, as discussed in Chapters 3 and 11, are also

included in a functions file. The recognized declarations are summarized below:

```
(PRIMITIVE-ACTIONS <list-of-primitive-actions>)
(EVALUABLE-FUNCTIONS <list-of-evaluable-functions>)
(EVALUABLE-PREDICATES <list-of-evaluable-predicates>)
(CLOSED-WORLD-PREDICATES <list-of-closed-world-predicates>)
(FUNCTIONAL-PREDICATES '((<predicate> .  <number>) ...(<predicate> .  <number>)))
(BASIC-EVENTS '((<predicate> .  <arity>) ...(<predicate> .  <arity>)))
(MESSAGES-TYPE <list-of-predicates>)
```

A functions file is a Lisp file that has a name of the form `<name>.LISP` (or `<name>.BIN` if it has been compiled). *Note that definitions and declarations in this file apply to to all PRS agents.*

5.2 Loading an Application

Applications can be loaded in two ways: *interactively* or using the PRS *demo facility*. To load an application interactively, the user must explicitly create the desired agents using the menu-based interface and then load the appropriate application files. The demo facility allows the user to specify the agents and their application files ahead of time in a Lisp file. Loading the application simply involves loading that single file. The demo facility is described Chapter 12.

Interactive loading is generally useful during the development phase of an application. Once an application system has stabilized, the demo facility provides a more convenient means of initialization.

Chapter 6

Running PRS

6.1 Loading PRS

Initializaing PRS first requires that you enter the appropriate window system and run the appropriate executable. If you have been provided with a PRS image, start that image directly. Otherwise, start a LISP executable that contains CLIM and load the file

```
~prs/released/lisp/system.lisp
```

to initialize the system; then execute `(load-prs)`.

After the system is loaded, the display window for the graphical user interface (GUI) can be created by executing the function call `(run-prs)`. Figure 6.1 illustrates the initial GUI window for PRS.

Clicking on the `Application` button on the top left portion of the screen will pop up a menu of the Grasper-CL-based application systems that are currently loaded. At this point, the menu should contain only PRS and Grasper-CL. If the Act-Editor is subsequently loaded, it will also appear on this menu. Selecting any of these menu items will transfer control of the window to the chosen system. The name of the system that is currently connected to the interface appears in the banner on the top of the display window.

6.2 Organization of the PRS Interface

In general, PRS is run using a default graphical interface whose *display window* is depicted in Figure 6.1. This interface is layered on top of the Grasper-CL system. Alternative interfaces are possible; in this chapter, we describe only the default interface.

The PRS display window is divided into several regions called *panes*. Each pane presents different kinds of information relevant to interacting with and controlling PRS. Although the layout of panes within the PRS window is normally fixed, it can be altered to a degree (see Section 6.3.1).

The largest area in the PRS display window is the *graph pane*. This pane is used to graphically trace execution of individual Acts. This pane is scrollable in case the the user wishes to examine portions of an Act that is too large to be displayed fully in the pane.

The upper-left corner contains the *mode pane*, which displays the name of the current operational mode (initially, it simply reads `Top Level`). PRS has a separate mode of operation for different types of interaction. The PRS supports four modes: `Window`, `Agent`, `Knowledge` and `Execution`, each of which enables a different set of commands. `Window` mode supports operations on the windows of the interface. `Agent` mode contains commands for creating and manipulating agents. `Knowledge` mode provides commands for inputting and manipulating the knowledge of an individual agent. `Execution` mode supports the run-time

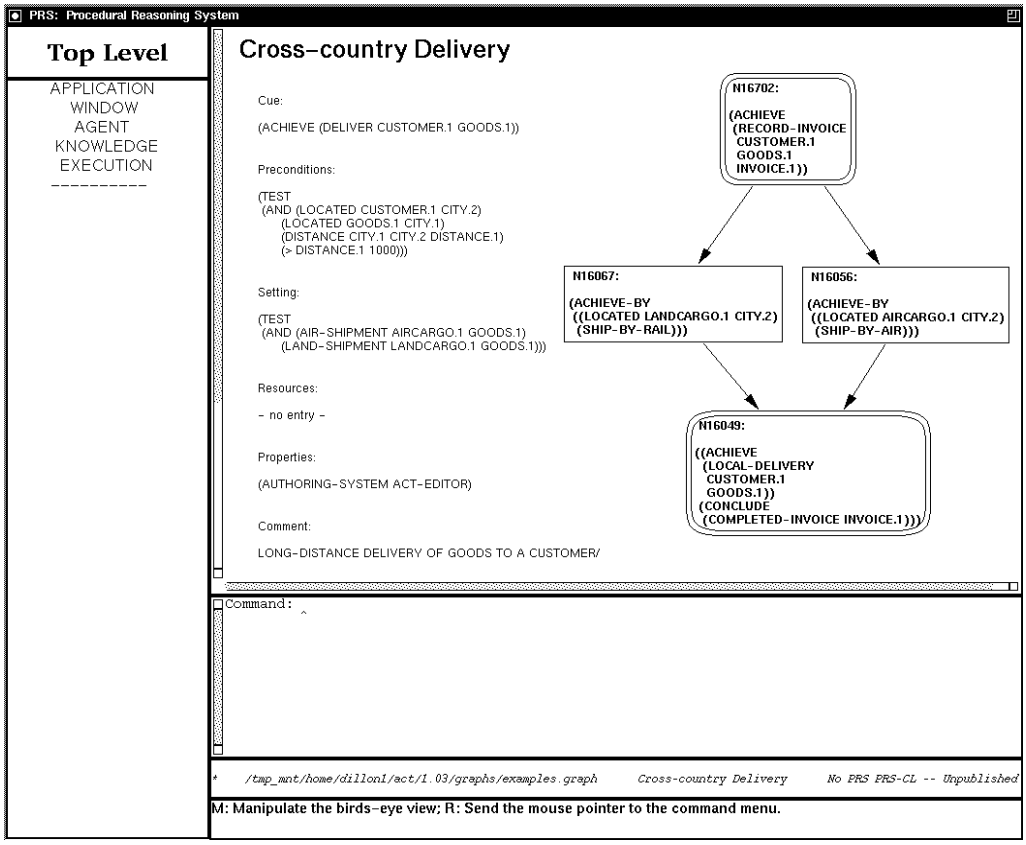

Figure 6.1: The PRS Display Window

activities of an individual agent. The *command pane*, situated beneath the mode pane, displays the PRS modes along with the menu of commands for the current mode.

Directly below the graph pane is the *interactor pane*. This pane is used to notify the user of warnings and certain system events. When PRS is waiting for the user to invoke a command it is said to be at the *top level*, and it displays the prompt `Command:` . When the user clicks on a command in the command pane, its name is typed out in the interactor pane. The interactor pane is also where various PRS commands print prompts for the user, and where input typed by the user in response to those prompts appears. This pane is scrollable. The interactor pane also serves as a Lisp listener. Any Lisp form entered into this pane (followed by RETURN) will be evaluated and its results printed there.

The *status pane*, below the interactor pane, displays information about the current state of the interface. Below the status pane is the *documentation pane*, whose contents change 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.

While PRS supports the use of multiple simultaneous agents, the interface is set up so that it is connected to at most one agent at a time. This agent is referred to as the *current agent*. The name of the current agent is displayed in the bottom of the status pane. Upon first entering the PRS interface or destroying the current agent, there will be no current agent. Various commands in `Agent` mode allow the identity of the *current agent* to be reset.

In addition to the *display window*, several other windows are defined within the PRS interface:

Birds-Eye View Window A smaller *birds-eye view window* can be activated that provides a low-

Figure 6.2: The Birds-Eye Window

resolution view of the entire graph that is scaled to fit completely in a single, small window. The birds eye is useful for visualizing large, complex Acts. A sample birds-eye window is displayed in Figure 6.2. Commands for activating and modifying the birds-eye window are available through the **Birds-eye** menu item on the **Window** menu, as described in the following section.

Trace Window The PRS interface allows certain activities within the system to be textually traced during execution. If desired, the user can direct the tracing information to a separate *trace window*. Each PRS agent has its own trace window. By default, tracing information is displayed in this window. However, the user can redirect tracing information to the interactor pane of the display window by using the **Aux Windows** command from the `Execution` menu.

Output Window Each agent has an associated output window. All non-trace information produced by an agent is sent to that output window, by default. As with the trace window, the user can instead force its output to the interactor pane by using the **Aux Windows** command from the `Execution` menu.

6.3 PRS Menus

Figure 6.3 shows the menus for the four command modes in PRS, namely `Window`, `Agent`, `Knowledge`, and `Execution`.

6.3.1 Window Menu

The `Window` menu provides commands to modify the display windows used by the PRS interface.

Pane-Layout This command can be used to alter the arrangement of the panes within the display window. The user can choose one of a fixed set of prespecified layouts from a pop-up menu.

Birds-Eye This command provides access to a pop-up menu that enables the user to turn on/off the Birds-eye window, as well as change various parameters (such as the scaling factor used and the parts of the Act to be displayed).

6.3.2 Agent Menu

The `Agent` menu provides access to commands that operate on the set of agents within PRS.

Select Select a PRS agent to become the current agent for the interface. If there is only a single agent, this command will set the current agent accordingly. Otherwise, a pop-up window will appear with the names of all active PRS agents from which the user can choose.

Create Create a new PRS agent and set it as the current agent. The user is prompted for a name for the new agent.

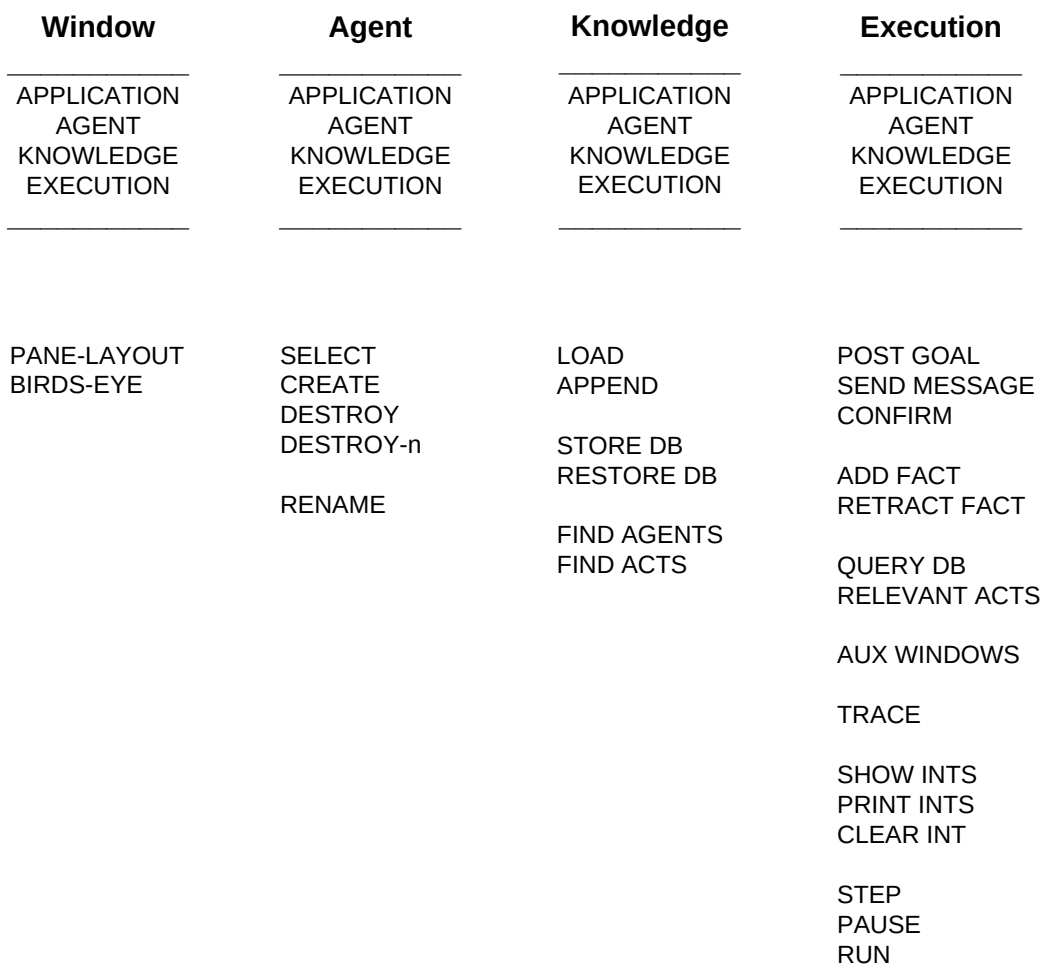

Figure 6.3: PRS Command Menus

Destroy Destroy the current PRS agent. A new agent must be designated as the current agent before any agent-specific commands can be executed.

Destroy-n Destroy multiple PRS agents. The user is prompted to select a set of agents from a pop-up menu.

Rename Give the current PRS agent a new name.

6.3.3 Knowledge Menu

The knowledge menu commands are used to manipulate the procedures, facts and functions of an agent. All commands from this menu other than the **Find** commands operate on the current agent; a current agent must be defined before this menu can be accessed.

Load Acts Load an Act file, a database file, or a functions file. All actions operate on the current PRS agent. After clicking on Load, you will be given a loading menu, from which to make your selection. When loading Acts, the system will discard all Acts that had been previously loaded (except for the Default Acts). Similarly, loading a database results in the old database being replaced.

Append Acts The Append Acts command is similar to the Load command, except that the Acts or database are added to the current PRS (rather than overwriting it). Note that Append is additive: if

you load the same set of Acts three times, those Acts will appear three times. If you wish to redefine an Act, you must use the Load command.

Store DB Create a copy of the current database of this PRS agent. This copy can be reinstalled as the current database using the command **Restore DB**. The command does not modify the current database.

Restore DB Install the most recently stored database as the current database. If no database has been stored, then the command is ignored. The command does not modify the stored database.

Show Acts This command displays a window containing a summary of Acts for the currently selected agent. The summary distinguishes fact-driven Acts from goal-driven Acts. The name and cue of each Act are displayed.

Find Agents This command will prompt the user for a symbol (corresponding to a function, predicate or constant), then will print a list of the current agents that use the symbol in one or more Acts that they load.

Find Acts This command is similar to **Find Agents**, but instead will print the name of the Acts and their corresponding graph files that access the user-supplied symbol. Any Acts that are loaded into some defined agent are considered.

6.3.4 Execution Menu

The execution menu contains the commands used to run a PRS agent. All execution commands operate on the current agent. Note that a current agent must be defined before this menu can be accessed.

Post Goal Post a goal for the current agent. See Section 2.1 for valid goal expressions.

Send Message Send a message to the current agent.

Confirm This command is used primarily while giving demonstrations. It pops-up a window containing a list of facts of the form `(OK <wff>)` for every corresponding fact `(NOT (OK <wff>))` in the database. Selecting items from this menu will result in those `(OK <wff>)` facts being added to the database. By writing applications that post `(NOT (OK <wff>))` facts then wait for the corresponding `OK` facts, the user can pause execution at critical points in an application to wait for 'confirmation' from the user before proceeding.

Add Fact Add a fact to the PRS database.

Retract Fact Remove a fact from the PRS database.

Query DB Query the database. The user will be promted to type in a fact. If it is in the database, that fact will be reprinted in the interactor pane. If it is not in the database, nothing is printed.

Relevant Acts Find the Acts whose Cue matches a user-supplied goal. Full unification is not performed, nor are Preconditions and Settings checked. As a result, *relevant* Acts are not necessarily *applicable* Acts.

Applicable Acts Find the Acts that could apply to a user-supplied goal.

Aux Windows Pop-up a menu for enabling/disabling the Output and Trace windows for the current agent.

Trace The trace command pops up a menu from which the user can activate or deactivate the following trace options.

Resources Trace the allocation and deallocation of resources.

Messages Trace the sending and receipt of messages.

Database Trace changes (additions and retractions) to the PRS database.

Procedure Text Textually trace the execution of Acts.

Procedure Graphic Graphically trace the execution of Acts by displaying them in the graph pane. During graphic tracing, the nodes in the Act will be highlighted as they are executed. Users may wish to adjust the duration of the highlighting on each node, depending on the nature of their application and the amount of tracing that has been activated. The variable `*FLASH-TIME*` can be used to this end; it specifies the number of seconds for each highlight (with default value .4).

Show Ints Display the intention graph for the current PRS agent in the interactor pane. A line is printed for each intention indicating: (1) the purpose of the intention (either a goal or fact), (2) the priority of the intention, and (3) the current state of the intention (see Section 7.1 for an explanation of the different states). The state is one of:

C designates the current intention,
N the intention is normal but not current,
S the intention is sleeping,
A the intention has been recently awakened.

Line indentation reflects the partial ordering within the intention graph.

Print Ints Print a summary of each intention, ignoring the partial order of the intention graph. Summaries are tree-structured, corresponding to the branching and subgoaling of Acts that are active for the intention. Each line of the summary contains (1) a subgoal/fact, (2) the KA/Act being executed for the subgoal/fact, and (3) the ID of the node being executed (but only when the node corresponds to a legitimate Act node, not some internal PRS node).

Clear Ints Reset the current PRS agent by removing all of its intentions. If the **Clear** command is executed while an agent is running, the agent will immediately halt its activities. *The current Acts and database are not reset.*

Step, Pause, Run This trio of commands control the running of PRS. Step causes a single step to be executed, Pause stops the execution, and Run allows a stopped agent to continue execution.

Chapter 7

The Intention Graph

An *intention* in PRS corresponds to a commitment to act, in response to either a posted goal or fact. Many intentions may be active simultaneously, some of which may be suspended or deferred, and some of which may be waiting for certain conditions to hold prior to activation.

The active intentions are organized in a data structure called the *intention graph*, which serves as a book-keeping tool for recording the run-time activity of an agent. In that regard, the intention graph plays a role similar to that of the activation stack in traditional programming languages. There are some important differences, however, the most notable of which is that intentions are not guaranteed to be executed. As will be described later, metalevel Acts can be used to alter the intention graph so that intentions may be altered or removed. For this reason, the intention graph is best thought of as the actions that an agent currently *believes* it will execute in the future.

7.1 Intentions

An intention records those specific actions that the system has chosen for execution in response to a goal or fact. This goal or fact is sometimes referred to as the *purpose* of the intention. A single intention consists of some initial Act together with all the sub-Acts being used to execute that Act. An intention has a tree structure in which successive layers in the tree correspond to levels of subgoaling within the Acts. Multiple branches from a single node indicate that several activities are being executed in parallel.

One important characteristic of intentions is their *state*, which is used to determine which intentions are available for activation. An intention can be in one of three possible states:

Normal: The most common state. Normal intentions at the root of the intention graph are eligible for activation.

Sleeping: An intention is in the sleeping state when its execution has been suspended, awaiting some condition to be satisfied. An intention can be put to sleep indirectly as the result of execution of a `WAIT-UNTIL` goal, or directly by the execution of an action of some intention. Each sleeping intention has an associated *activation condition*, which is a logical expression evaluable in the environment of the intention. The intention remains asleep until the activation condition becomes true, at which time it enters the *awake* state.

Awake: An intention is in the awake state immediately after it emerges from the sleeping state. The awake state is similar to the normal state except that when there is more than one intention eligible for activation, the system prefers the most recently awoken one. As soon as an intention in the awake state has been activated, the intention is put in the normal state.

```
o (ACHIEVE (DELIVER ACME-FLORIST FLORAL-GOODS))  Priority: 0  State: (C)
  * (ACHIEVE (DELIVER ACME-FLORIST FLORAL-GOODS)) {Cross-Country Delivery, N16702}
    * (ACHIEVE-BY ((LOCATED (TULIPS ROSES) TORONTO) (SHIP-BY-AIR)))
      * (ACHIEVE (LOCATED (TULIPS ROSES) TORONTO)) {SHIP-BY-AIR, N66596}
    * (ACHIEVE-BY ((LOCATED (CHRISTMAS-TREES) TORONTO) (SHIP-BY-RAIL)))
      * (ACHIEVE (LOCATED (CHRISTMAS-TREES) TORONTO)) {SHIP-BY-RAIL, N66590}
        * (ACHIEVE (MOVE-TRAIN TRAIN-25 TORONTO)) {MOVE TRAIN}
          * (CONCLUDE (LOCATED TRAIN-25 TORONTO))
```

Figure 7.1: A Sample Intention Summary

PRS provides a facility to display a textual summary of intentions (namely, the Print Ints command from the **Execution** menu). Figure 7.1 displays an intention from a delivery domain, using (among others) the Act `Cross-Country Delivery` from Figure 2.1. This intention was generated in response to the goal `(ACHIEVE (DELIVER ACME-FLORIST FLORAL-GOODS))`. The first line in the summary describes the purpose of the intention, along with its priority and state. Each line in the remainder of the summary contains (1) a subgoal/fact, (2) the Act being executed for the subgoal/fact, and (3) the name of the last successfully executed node in that ACT. The ACT name and node are omitted when a primitive action/evaluable predicate is being used in an attempt to satisfy a subgoal. The node name is also omitted in situations where the ACT has not yet begun execution.

7.2 Intention Graph

The intention graph consists of a partial order over the set of defined intentions. The use of a partial order enables the establishment of priorities and other relationships among intentions. The intention graph may contain zero, one or many tree structures. The intention comprising the root of each intention tree is referred to as the *root intention* for that tree. Within the intention graph, there is always a designated *current intention*, which is the intention to be executed during the current cycle.

Intentions are established and manipulated in two ways. The system interpreter (in the top level main loop) modifies the intention graph by adding intentions that have been chosen for execution and removing those intentions that have finished execution. Meta-Acts can also be used to modify the intention graph, generally for two different reasons. The principal application is to implement some alternative control scheduling strategy that reorders the intentions. Additionally, metalevel Acts can be used to modify the intention graph in response to some event in the domain. For example, suppose a number of intentions have been established as part of responding to some set of events. While these intentions are being executed, some new event may arise that makes it necessary to completely switch activities. In such a case, it may be appropriate to kill some (or all) of the intentions currently in the intention graph in order to halt the activities that are no longer appropriate.

Figure 7.2 displays a textual overview of an intention graph that was generated while running in the delivery domain mentioned above. Displays of this sort can be obtained using the command Show Ints from the **Execution** menu. Each line corresponds to an intention, with the indentation indicating links in the partial order. The information for each intention consists of: (1) the purpose of the intention (either a goal or fact), (2) the priority of the intention, and (3) the current state of the intention. The state is one of:

C designates the current intention,
N the intention is normal but not current,

```
** INTENTION GRAPH **
 o (NEW-CUSTOMER HISTORIC-INN BOSTON)  Priority: 0   State: (N)
 o (NEW-CUSTOMER NATIONAL-INN TORONTO)  Priority: 0   State: (N)
 o (SOAK (#<KA-Op-Instance ADD CUSTOMER> #<KA-OP-Instance ADD CUSTOMER>))
        PRIORITY: 0   STATE: (C)
     o (ACHIEVE (DELIVER ACME-FLORIST FLORAL-GOODS))  Priority: 0   State: (N)
```

Figure 7.2: A Sample Intention Graph Summary

S the intention is sleeping,
A the intention has just been awakened.

This intention graph contains four intentions, three of which are root intentions. Two of these root intentions were created in response to posted facts of the form `(NEW-CUSTOMER <customer> <location>)` using the ACT from Figure 2.2. The remaining root intention was created to respond to a fact of the form `(SOAK <list>)` (this is a meta-fact, as described in Chapter 9.) This intention is deciding how to respond to two new KAs/Acts that have been triggered. The fourth intention was created to in response to a goal of the form `(ACHIEVE (DELIVER ACME-FLORIST FLORAL-GOODS))` using the Act `Cross-Country Delivery` from Figure 2.1. This last intention is a descendant of the `SOAK` intention, indicating that the `SOAK` intention has higher priority for execution. Intuitively, that means that PRS will decide what to do about the new customers before addressing the delivery task.

Chapter 8

PRS Execution

PRS execution is controlled by the system *interpreter*. The main objectives of the interpreter are to respond to posted goals while simultaneously taking steps to satisfy its current goals. These steps may involve posting subgoals or performing *primitive actions* that modify either the external world or internal PRS structures.

The basic activities of the interpreter are outlined in Figure 1.3. The interpreter repeatedly cycles through three phases of operations: *selecting* an Act in response to the current goals and facts, *intending* the Act, and *activating* an intention to carry out some designated action. These three phases are described in detail below.

8.1 Select Phase

The objective of the selection phase is to identify a single Act to be added to the intention graph in response to any new facts or goals posted during the last execution cycle. The interpreter will choose the Act randomly from among all Acts that are *applicable* in the current cycle, as described below. The restriction of choosing a single Act may seem unduly limiting, since it can be useful to intend multiple Acts in parallel for some applications. As described in Chapter 9, the simple scheme described here can be used to provide highly complex control policies (such as intending Acts in parallel) through an appropriate use of Meta-Acts.

Applicability is determined by considering the goals and facts posted to the PRS agent in the last cycle, and the gating environment conditions of the individual Acts. A *fact-invoked* Act is applicable when the Setting, Precondition and Resources constraints are satisfied by the current database and the `CONCLUDE` condition in its Cue has been added to the database in the previous cycle. A *goal-invoked* Act is applicable when the Setting, Precondition and Resources constraints are satisfied by the current database and the `ACHIEVE` or `TEST` goal listed in its Cue was added to the database in the previous cycle.[1] In determining Act applicability, the interpreter will not automatically perform deductions. Rather, both beliefs and goals are matched directly with environment conditions using unification only. This allows applicable Acts to be selected quickly, thus guaranteeing a certain degree of reactivity. If arbitrary deductions were made, this guarantee could not be made. (PRS can be made to perform any necessary deductions by invoking appropriate metalevel Acts. These metalevel Acts are themselves interruptible, so that the reactivity of the system is retained.)

After determining the set of applicable Acts, the interpreter posts several meta-facts to the database that describe certain characteristics of the set. One such fact has the form

[1] Recall that a *fact-invoked* Act has a Cue containing a `CONCLUDE` operator while a *goal-invoked* Act has a Cue containing an `ACHIEVE` or `TEST` operator.

`(SOAK <act-list>)`

where `<act-list>` is a list of the applicable Acts. The acronym `SOAK` stands for *Set Of Applicable KAs/Acts*. A complete list of the meta-facts posted after determining the applicable Acts is given in Figure 9.1

As described in Chapter 9, the meta-facts can be used by meta-acts to control system execution. In particular, the addition of these meta-facts to the PRS database may cause certain meta-acts to become applicable. Thus, before the interpreter selects an Act to be added to the intention graph, it first needs to re-check applicability in light of the new meta-facts. For this reason, the interpreter invokes the applicability-testing process again using the new meta-facts that have been added to the database. At this stage, only Acts that make use of the newly-posted meta-facts will be considered applicable (provided all of their gating environment conditions are met). At the end of this second cycle, the interpreter will again post a new set of meta-facts to describe the new set of applicable Acts. This applicability-testing loop continues until no new Acts are applicable. At that point, a single Act is chosen from the applicable Acts of the previous cycle. If there is no applicable Act, the execution phase begins immediately. Otherwise, the chosen Act is first inserted into the intention graph, then the execution phase begins.

8.2 Intend Phase

During the *intend phase*, the applicable Act chosen during the selection phase is inserted into the intention graph. This process is referred to as *intending* the Act.

PRS distinguishes two types of goals. An *external goal* is one that is not a means to an already intended end but rather results from some external event (such as a goal or fact posted by a user, or a message sent by another process). An *operational goal* is a subgoal of an existing intention. If the selected Act responds to an external goal or a new belief, it will be inserted into the intention graph as a new root intention. For example, any fact-invoked Act that became applicable as the result of information added to the database will result in a new root intention. For an Act that responds to an operational goal of some existent intention, that intention is expanded to reflect the decision to apply the new Act.

At first sight, this seems unduly restrictive – one often wants to attend to more than one task, and one often wants to order these tasks for later execution rather than have them executed immediately (which placing them at the root of the intention graph entails). The way around this problem lies in the meta-acts: *they provide the means to place multiple intentions on the intention graph and order intentions arbitrarily.* The use of meta-acts to modify the intention graph is explored in detail in the next chapter.

8.3 Activate Phase

Execution takes place during the *activate phase*. During this phase, some individual intention is chosen from the set of *eligible* intentions and a single step is executed in accordance with the current goals of the leaf nodes of the intention. Any non-sleeping root intention is eligible for activation. If there are eligible intentions in the *awake* state, the most recently awakened intention is given priority. Otherwise, one of the eligible intentions is selected randomly.

As described above, each leaf of an intention has a single goal associated with it. Goals based on the operators `CONCLUDE`, `RETRACT`, and `TEST`, can be directly executed with the interpreter performing the appropriate action. `REQUIRE-UNTIL` goals are similarly satisfied by internal interpreter actions. Goals of the form `(ACHIEVE (P t1 ...tn) (a1 ...an))` or `(ACHIEVE-BY (P t1 ...tn))` where `P` is an evaluable predicate are also directly executable, with the corresponding Lisp function for the predicate being evaluated. When a goal is successful, the intention must be updated to reflect the satisfaction of the the goal.

The successor of the Act node that lead to the goal on the intention node must then be posted. When a goal fails, the intention must also be updated and corresponding actions taken.

All other achievement goals result in the posting of appropriate subgoals that will be considered during the next cycle through the main interpreter loop.

Prioritized Activation of Intentions

It is possible to prioritize the activation of intentions in domain-specific ways by defining a function that selects the intention to be activated from among all eligible intentions. The slot `activation-sort-predicate` of the `intention-list` structure for each PRS agent contains the function that is used to choose among the eligible intentions.

The value for this slot defaults to the value of the variable `*DEFAULT-ACTIVATION-SORT-PREDICATE*`. This variable is originally set to a function that chooses randomly among the eligible intentions (thus producing the behavior described above). The slot can be set by changing `*DEFAULT-ACTIVATION-SORT-PREDICATE*` prior to creation of the PRS agent or by using the agent parameter `:activation-sort-predicate` when the PRS demo facility is used (see Chapter 12). The slot value can be any function that takes a list of intentions as argument and returns one of those intentions.

To further assist in prioritizing intention activation, activation-sort-predicates can make use of a slot `priority` in the class `intention`. This slot can be used to assign priorities to intentions directly. The default value for the priority is 0, with larger numbers indicating increased priority. There is no ceiling on the priority values. The priority slot can be filled by passing a `:PRIORITY` keyword argument to the function `INTEND`.

Chapter 9

Metalevel Acts

One of the most interesting and powerful features of PRS is its use of metalevel Acts to control execution of the system. Metalevel and baselevel Acts are the same structurally: they share the same formalism and specification. The difference between the two types lies in their contents. Baselevel acts are defined over domain facts and invoke actions that modify the application world or database. Metalevel acts may additionally use meta-level facts about the system operation and may modify the internal structures of an agent.

9.1 The Use of Meta-Acts

The PRS interpreter uses a simple, general-purpose strategy for controlling execution of an agent. The generality of the approach provides users with the flexibility to encode complex control strategies for individual domains using meta-acts. It is important to understand that meta-acts *don't change* the underlying execution mechanisms described in Chapter 8. Rather, they exploit those mechanisms to make the system behave in manner that appears as if some more complex execution strategy were being used.

To illustrate this point, we consider a situation where meta-acts can be employed to extend the functionality of the basic system. As noted above, the basic behavior of the interpreter is to choose *one* applicable Act to add to the intention structures at each cycle. Without any further actions, all other applicable Acts from the cycle are ignored. In particular, they are *not* considered in the following cycle. So, for example, if the facts G1 and G2 are posted in a single cycle PRS will select an applicable Act for at most one of the new goals. The other goal is discarded.

The Setting binds `G.1` and `F.1` to be the goal-invoked and fact-invoked Acts on the SOAK list. The variables `G.1` and `F.1` store all applicable Acts in this cycle, thus making it possible to intend these Acts in future cycles of the interpreter.

In certain domains though, discarding goals is not always desirable. The meta-act `Meta Selector: all goals & all facts` in Figure 9.1 can be used to force the system to act on all goals and facts posted in a given cycle. The Cue for this Act is `(CONCLUDE (SOAK X.1))`, with Precondition `(TEST (> (length X.1) 1))`. As such, this Act becomes applicable whenever there is more than one applicable Act.

The first node in the plot binds the variable `F.1` to a list containing all applicable fact-invoked KAs/ACTs in the current cycle. The next node has the goal

```
(ACHIEVE (INTENDED-ALL F.1)),
```

which has the effect of intending an Act for each of the fact-invoked KAs/Acts in `F.1`. In particular, there is a predefined meta-act called `Meta Intend All` that responds to goals of the form `(INTENDED-ALL goals.1)` and causes each of the members of `goals.1` to be intended. The goal

```
(ACHIEVE (= G.1 (GOAL-INVOKED-KAS X.1)))
```

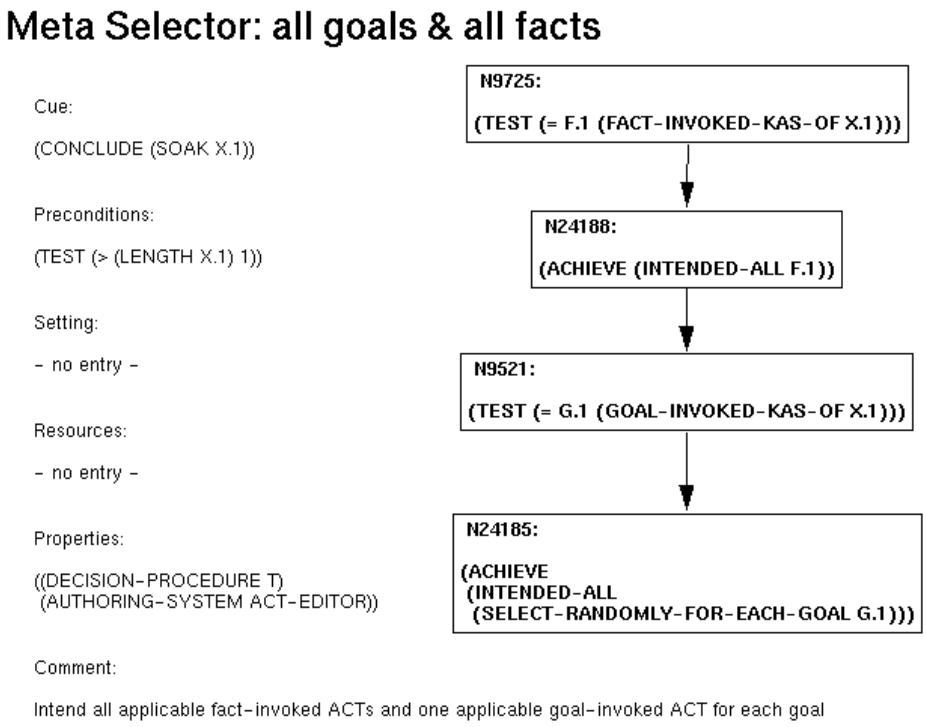

Figure 9.1: The Meta-Act `Meta Selector: all goals & all facts`

on the next node binds the variable `G.1` to a list of the goal-invoked KAs/Acts for the current cycle. This variable is used in the goal

```
(ACHIEVE (INTENDED-ALL (SELECT-RANDOMLY-FOR-EACH-GOAL G.1)))
```

on the ensuing node. This goal has the effect of intending an Act for each goal that has been posted in the current cycle. `SELECT-RANDOMLY-FOR-EACH-GOAL` is a predefined evaluable function that selects one applicable Act from each set of Acts in `G.1` that apply to different goals.

Figure A.4 provides a second example of a meta-act. This meta-act can be used to make an informed choice among multiple applicable Acts for achieving a goal, rather than having the interpreter choose randomly. See Appendix A.3.3 for a description of how it works.

Meta-acts can be used to produce a wide array of complex behaviors. The principal reasons for using meta-acts are to:

- Intend multiple Acts that apply in a single interpreter cycle.
- Make informed choices among multiple applicable Acts.
- Reorder the intention graph in response to new information.
- Kill intentions for activities that are no longer appropriate.

9.2 Writing Meta-Acts

As noted above, the structure of a meta-act is identical to that of a domain-level act. The difference between the two lies in their contents: what conditions are used in their environments and plots, the

- `(SOAK <ka-list>)` the set of applicable KAs/Acts.
- `(FACT-INVOKED-KAS <ka-list>)` the set of fact-invoked KAs/ACTs.
- `(GOAL-INVOKED-KAS <ka-list>)` the set of fact-invoked KAs/ACTs.
- `(APPLICABLE-KAS-FACT <fact> <ka-instances>)` One such fact is concluded for each fact that has a non-empty set of applicable KAs/Acts.
- `(APPLICABLE-KAS-GOAL <goal> <act-instances>)` One such fact is concluded for each goal that has a non-empty set of applicable KAs/Acts.
- `(FAILED-GOAL <goal>)` One such fact is concluded for each failed goal.
- `(DB-SATISFIED-GOAL <goal> <act-instances>)` One such fact is concluded for each goal that is satisfied without applying any further actions (i.e., is true in the database or by virtue of evaluable predicates and functions).

Table 9.1: Meta-predicates

- `(DECISION-PROCEDURES-OF <ka-list>)` the set of decision procedures in `<ka-list>`.
- `(FACT-INVOKED-KAS-OF <ka-list>)` the set of fact-invoked KAs/ACTs in `<ka-list>`.
- `(GOAL-INVOKED-KAS-OF <ka-list>)` the set of fact-invoked KAs/ACTs in `<ka-list>`.
- `(PROPERTY-P <property> <ka-instance>)` returns the value of a specified property from the property list of the ACT associated with a ka-instance.

Table 9.2: Meta-functions

actions that they take, and the properties that they have.

Consider first the gating environment conditions for a meta-act. The two meta-acts mentioned above both made use of meta-facts of the form `(CONCLUDE (SOAK <ka-list>))` in their Cues, meaning that they are invoked when the set of applicable KAs/ACTs have certain characteristics. Other meta-facts describing the internal system state are used to trigger different kinds of meta-acts. For example, beliefs about failed goals could trigger a metalevel Act to deliberate on the utility of retrying the goal. Table 9.1 summarizes the meta-predicates used in the system-generated meta-facts. Meta-facts built from these predicates are posted by the system during each cycle of applicability testing. These meta-facts are classified as *basic events*, meaning that they can trigger Acts, but are not retained in the database after the current loop of applicability testing. Meta-predicates can also be used throughout the plots of a meta-act.

A limited set of *meta-functions* are also predefined in PRS for use in writing meta-acts. Table 9.2 presents a collection of these functions.

The subgoals or actions taken by meta-acts could change any aspect of the agent's internal structures. Typically, they are used to modify the intention graph in some manner. Several predefined meta-acts are provided in the default-processes graph that modify the intention graph directly. The most commonly used of these meta-acts are summarized in Figure 9.2; for a complete list, refer to Appendix B.

For meta-acts that choose among multiple applicable Acts for a posted goal, it is essential that the intention created for that Act be activated (and completed) before the lower-level intention is again activated. The simplest way to guarantee this ordering is to insert the new intention ahead of the old intention in the intention graph. To assist in this process, PRS recognizes an Act property called *decision-procedure*.

`Meta Intend`
Cue: `(ACHIEVE (INTEND X.1))`
Calls Function `(intended-from-ka x.1)`
Effects: *Intend an applicable Act.*

`Meta Intend All`
Cue: `(ACHIEVE (INTENDED-ALL LIST.1))`
Calls Function: `(intended-all-from-ka list.1)`
Effects: *Intend a list of applicable Acts.*

`Meta Intend All As Root`
Cue: `(ACHIEVE (INTENDED-ALL-AS-ROOT LIST.1))`
Calls Function: `(intended-all-as-root list.1)`
Effects: *Insert a list of intentions at the root of the current intention.*

Figure 9.2: Selected Default Meta-Acts for Modifying the Intention Graph

When an Act with `(DECISION-PROCEDURE T)` in its Property slot is to be intended, a new intention is created and set as the only root of the graph (that is, it has all other intentions as descendants). Since interactions among intentions can be somewhat subtle, it is generally recommended that users add the property *decision-procedure* to any meta-acts that they write.

It is up to the user to ensure that the invocation of higher and higher levels of his/her meta-acts eventually terminates. It is sometimes quite easy to violate this requirement. For example, consider a meta-act that has the following goal expression in its Cue:

```
(CONCLUDE (AND (SOAK X.1))
               (EQUAL 1 (LENGTH X.1)))
```

Such an Act will generally cause PRS to loop indefinitely because the Act will repeatedly become the single applicable Act, thus reactivating itself. (The loop could conceivably be broken by other meta-acts, if appropriately defined.)

Chapter 10

Useful System Definitions

This chapter discusses various system-defined ACTs and functions that provide built-in functionality in specific areas.

10.1 Input/Output

Several default ACTs and primitive actions are provided to enable input and output by an agent during execution.

`(ACHIEVE (PRINT <expn>))` print the value of `<expn>` to the Output Window.

`(ACHIEVE (PRINT-LIST <format-stmt>))` print a LISP format statement.

`(ACHIEVE (PRINT-WARNING <format-stmt>))` print a LISP format statement, and beep to notify the user.

`(ACHIEVE (READ <var>))` read an expression from the Input Window and bind it to the variable `<var>`.

`(ACHIEVE (QUERY <format-stmt> <var>))` read an expression from the Input Window and bind it to the variable `<var>`, using a LISP format statement to prompt.

`PRINT`, `QUERY` and `READ` are implemented by default Acts, while `PRINT-LIST` and `PRINT-WARNING` are implemented as primitive actions.

10.2 Equality, Unifiability, Assignment

Several forms of equality-related reasoning are provided within the system.

The equality of two objects can be tested using the primitive action predicates `EQ` and `EQUAL`. The associated primitive action functions have the same semantics as the LISP functions of the same name.

Unifiability of two expressions is determined by either of the following goals:

```
(TEST (= <expn1> <expn2>))
(ACHIEVE (= <expn1> <expn2>))
```

These two goal expressions are equivalent; both are supported simply for the sake of generality. For these goal expressions, default ACTs apply that will attempt to unify the two expressions. When the expressions are unifiable, the bindings for any variables in the expressions are modified to reflect the unification. Thus, for example, the goal

```
(TEST (= (P X.1 5) (P 3 Y.1))
```

will suceed, binding the variable `X.1` to the value `3` and `Y.1` to the value `5`. The system does not currently support the use of evaluable functions within `<expn1>`.

The above goal expressions for equaliting testing can lead to multiple invocations of embedded evaluable functions, since there may be multiple attempts to satisfy the overall equality-based goal. The goal expression `(ACHIEVE (*=* <var> <fn-call>))` provides an additional mechanism for variable binding that circumvents this problem. The first argument must be a variable, the second argument a ground function call. This expression is guaranteed to invoke any evaluable functions in `<fn-call>` only once, as such, it is the recommended way to evaluate functional expressions that may have side-effects, or that may be expensive or time-consuming to compute.

10.3 Manipulating Sets of Objects

One often needs to manipulate sets of objects that satisfy some particular properties or to apply given actions to such sets. To do this, PRS provides two mechanisms: the first for selecting sets of objects, and the second for abstracting actions so that they can be applied correctly to the elements of a set.

A set of objects There is a predefined evaluable function `ALL` that can be used to collect a set of objects that satisfy some condition. `ALL` takes two arguments. The first argument is a variable X and the second argument is a `wff` W containing the variable X. The function returns all values of X that are currently believed by the system to satisfy W (i.e., all values of X that, when substituted for X in W, yield a formula contained in the PRS database).

For example `(ALL X.1 (AND (TYPE VALVE X.1) (POSITION X. CLOSED)))` returns the list of all closed valves. As a second example, the function

```
(ALL X.1 (AND (TYPE SWITCH Y.1) (TYPE VALVE X.1) (ASSOCIATED X.1 Y.1)
           (POSITION X.1 CLOSED) (POSITION Y.1 GPC)))
```

returns all closed valves for which the associated switch is in the GPC position.

Action Abstraction The `Apply to All` Meta Goal, together with the `lambda` operator, enables the user to specify a goal to be achieved over a set of objects or individuals. It succeeds if and only if the goal succeeds for all elements of the set.

For example:

```
(ACHIEVE (APPLY-TO-ALL (LAMBDA VALVE.1 (ACHIEVE (POSITION VALVE.1 CLOSED)))
                 (ALL MANIFOLD.1
                      (AND (TYPE MANIFOLD-VALVE MANIFOLD.1)
                         (POSITION MANIFOLD.1 OPEN)))))
```

will try to achieve the goal of closing the valve for all manifold valves that are open.

The Meta acts that fulfill the goal `Apply to All` are provided in the default Acts file.

Chapter 11

Multiple PRS Agents and Communication

For many applications, it is useful to employ more than one PRS agent. Doing so provides a means of partitioning differing functionalities of the application into modular units. For example, in constructing a PRS application to control the operation of a mobile robot it might be convenient to have one agent to supervise motor activity, a second agent to process perceptual information and a third for interacting with human supervisors.

Each agent has the basic architecture described in Figure 1.1, with its own set of ACTs, database and intention graph. Coordination of activity among the various agents is achieved through message-passing, as described below.

11.1 Inter-agent Communication

PRS agents can communicate among themselves and with other processes (such as a user or a simulator) by passing messages. An agent can be made to send a message by posting a goal

```
(ACHIEVE (send-message recipient message))
```

where `recipient` is the name of the PRS agent (or process) to whom the message is being sent and `message` is the message itself. Alternatively, one can directly use the Lisp function `(send-message recipient message)` from a Lisp window, or the `Message` command from the `AGENT` menu.

A message is a proposition represented in predicate calculus form: it must conform to the syntax of facts as described in Chapter 4. A message sent to a PRS agent is, upon receipt, added to that agent's database. As with any new fact, this can trigger ACTs that respond to the incoming message. Currently, there are no standard or default ACTs for processing incoming messages—these must be written by the user as part of the application domain.

If all PRS agents and processes are assumed trustworthy, the messages passed among them can be interpreted as facts about the current world state. In more complex domains where, for example, agents may have inaccurate information, it is preferable to use some special predicate to denote the fact that the message is an assertion made by some particular agent rather than a fact about the world. For example, agents could use a predicate of the form

```
(asserted sender fact)
```

when sending messages. The recipient of this message then has the opportunity to decide how to treat this assertion based on factors such as the identity of the sender, situational information, or the fact itself (using appropriate ACTs). The recipient could accept the asserted fact and add it to its database, reject

it as unreliable, or combine it with other evidence in some other way. For example, if a PRS agent called `INTERFACE` wished to advise a PRS agent called `RCS` that the valve `valve1` was closed, it could send `RCS` the message

```
(asserted INTERFACE (position valve1 closed))
```

The agent `RCS` could then apply its internal Acts in order to decided to do with this message.

Messages can also be used to encourage the recipient to adopt a particular goal. Supposed some PRS agent (or process) A wants another PRS agent B to adopt some particular goal. A cannot directly establish a goal for B — the best A can do is to request that B adopt the given goal, using some special message format. A possible syntax for such requests is

```
(requested sender goal)
```

where `goal` is the goal that the sender wants the recipient to adopt. For example, if the PRS agent `RCS` wished the PRS agent `INTERFACE` to close a valve `valve1`, `RCS` could send `INTERFACE` a message with message content:

```
(requested RCS (ACHIEVE (position valve1 closed)))
```

There are no standard or default ACTs for responding to such messages—it is the responsibility of the user to write ACTs for establishing such goals as appropriate.

In addition to the use of messages to assert facts (beliefs) and request the adoption of goals, one can similarly use them to perform other kinds of communication acts: committing, promising, informing, etc.

11.2 Message Types

There is a variable `*MESSAGES-TYPE*` in PRS that is used to identify the types of messages being used in the current application. This variable is initially set to the list

```
(request achieved failed)
```

but the user can add types by executing the function

```
(MESSAGES-TYPE <list-of-predicates>).
```

`MESSAGES-TYPE` declarations are used to support the passing of variables among agents. For all messages whose first element is a member of this list, any unbound variable contained in the message will be replaced by the symbol `what`. The following example shows how this mechanism can be used to transmit values between agent.

Suppose that agent A wants to know the pressure in the tank T, and it knows that the agent B knows or can compute such knowledge, then A can send to B a message such as:

```
(request A (TEST (pressure T x.1)))
```

`x.1` is unbound in this expression but we expect it to be instantiated by B. After this message has been sent, A should wait to receive a message from B that indicates the pressure. In other words, A should have the goal similar to:

```
(WAIT-UNTIL (achieved B (TEST (pressure T x.1))))
```

Meanwhile, B will receive the message:

```
(request A (TEST (pressure T WHAT)))
```

This message could trigger some ACT in B that will determine the pressure of T and send back the message:

```
(achieved B (TEST (pressure T 245)))
```

Since A is waiting for `(achieved B (TEST (pressure T x.1)))`, it will bind `x.1` to 245 upon receipt of the message from B.

By default, the binary predicates `request` and `achieved` are declared as *basic events* (see section 3.4). Thus, instances of these predicates are removed at the end of one PRS cycle. The declaration is made primarily to keep the databases free of extraneous facts related to outdated messages. Should additional message-passing predicates be defined, the user may also wish to declare them as basic events.

Chapter 12

The Demo facility

PRS provides a facility for defining an application (or *demo*). It enables a PRS application to be specified easily and further provides commands to start, stop, trace, or reinitialize the agents in the system.

12.1 Creating a Demo

The demo facility is best described by example. We use the call presented below for illustration:

```
(make-instance 'prs-demo
               :name 'AIR-SEA-DEMO
               :prs-agents-description
               `((TIME-MANAGER
                  :databases ("~prs/air-sea-demo/shared-database.lisp"
                              "~prs/air-sea-demo/time-manager-database.lisp")
                  :processes ("~prs/air-sea-demo/time-manager.graph"
                              "~prs/air-sea-demo/meta-acts.graph'')
                  :activation-sort-predicate ,(function PREFER-NONUPDATE-INTENTIONS)
                 (AIRPLANE
                  :databases ("~prs/air-sea-demo/shared-database.lisp"
                              "~prs/air-sea-demo/airplane-database.lisp")
                  :processes ("~prs/air-sea-demo/airplane.graph"
                              "~prs/air-sea-demo/meta-acts.graph")
                  :activation-sort-predicate ,(function PREFER-NONUPDATE-INTENTIONS)
                 (SUBMARINE
                  :databases ("~prs/air-sea-demo/shared-database.lisp"
                              "~prs/air-sea-demo/submarine-database.lisp")
                  :processes ("~prs/air-sea-demo/submarine.graph"
                              "~prs/air-sea-demo/meta-acts.graph")
                  :activation-sort-predicate ,(function PREFER-NONUPDATE-INTENTIONS)
                 ))
               :function-file "~prs/air-sea-demo/shared-functions.lisp"
               :select-prs-agent 'TIME-MANAGER)
```

This call would lead to the creation of a PRS demo system called AIR-SEA-DEMO containing agents name TIME-MANAGER, AIRPLANE, and SUBMARINE. The appropriate database and ACT graph files would be loaded into these agents initially. In addition, each agent has been set to use the function `PREFER-NONUPDATE-INTENTIONS` to determine which one of the eligible intentions should be activated (see

Section 8.3 for more information on prioritized execution of intentions). The TIME-MANAGER would be set as the current agent and the designated function file loaded into the Lisp environment.

The above call demonstrates the most commonly used parameters of the demo facility but others are supported. The full set of parameters for the demo facility is summarized below.

Demo-level Parameters

:name a symbol that uniquely identifies the demo

:prs-agents-description a list of agent descriptions (defined below)

:function-file the function file for the demo

:select-prs-agent what should be the current agent

Note that if no value is provided for parameter **:select-prs-agent** then by default the current agent is set to be the last agent in the list of agent descriptions.

Agent Descriptions

An agent-description is a list whose first element is a symbol that serves as the name of the agent, and whose remainder consists of a selection from the following list of agent parameters.

:databases a list of the databases to be loaded into the agent

:processes a list of the procedures (Acts or KAs) to be loaded into the agent

:activation-sort-predicate a function for choosing an intention to activate from the set of eligible intentions

:type-existence-fn a function for determining if a variable is typed

:alternative-consultation-sources a list of names of consultation functions

:external-db-function a list of database initialization functions

Note that the slot `:alternative-consultation-sources` takes a list of *names* of functions. All other slots for functions require the actual function definitions.

12.2 Manipulating a Demo

There are number of commands available for manipulating PRS demos, as described below. The variable `*current-demo*` is set to the currently active prs demo while the variable `*prs-demos*` contains the list of all active PRS demos. The function `(FIND-PRS-DEMO-NAMED <name>)` can be used to located a PRS-demo object from its name.

- (`REINITIALIZE <prs-demo>`) Reinitialize the demo by restoring the databases for all PRS agents, and reinitializing all simulators defined for the demo.
- (`STOP <prs-demo>`) Halt all PRS agents and simulators in the demo, as well as simulated time.
- (`STOP-BUT-NOT-SIMULATORS <prs-demo>`) Similar to STOP but does not stop the simulators.

- (RESTART <prs-demo>) Restart a demo after it has been halted by STOP or STOP-BUT-NOT-SIMULATORS.
- (KILL <prs-demo>) Kill the demo.

Chapter 13

Trouble-shooting PRS Execution

Certain unexpected difficulties may arise during the use PRS. This section presents suggestions for dealing with such problems.

13.1 Recovering from Problem States

During operation of PRS, the user may occasionally enter states in which the menus and mouse-sensitive items on the screen do not respond. This state of affairs can arise for different reasons, with the appropriate action to be taken being dependent on the cause.

The first step to take is to determine whether PRS hit an error. The combination of Lisp and CLIM can make errors difficult to recognize. If an error occurs, an error message will be sent to the window from which the Lisp process was first *but nothing will appear in the GUI window.* So, if the GUI seems to have stopped, check the Lisp window for an error. Errors will generally include a continuation option for returning to the command level of the GUI (it first aborts the problematic command). Note that any PRS functions called from the break in the Lisp window will have their output go to the default output window, which is the interactor pane. To send the output to the Lisp window, execute the Lisp function call `(setq *standard-output* *terminal-io*)`.

At times, the GUI may seem not to respond to any input despite the fact that no error has been noted in the Lisp window. PRS is probably performing some time-consuming operation, such as displaying a large ACT as part of the graphic tracing.

Sometimes, it may appear that PRS is not running because you have posted a goal and are getting no response. If PRS execution has been paused, then the system can not continue running until you directly command it, using either the **Step** or **Run** commands from the **Execution** menu. Another check to make is to ensure that the interface is currently connected to the agent that you think is executing.

If PRS should ever get genuinely stuck, it is best to kill the window and create a new one. Simply interrupt the Lisp process (type two controls-Cs to the Lisp window), abort to the top level, and reinvoke the function `(user::run-prs)`.

13.2 Points of Confusion

There are certain aspects of PRS that often confuse users. The following list covers some of them.

- Relevant ACTs (as returned by the Relevant ACTs command in the Execution Menu) are not always *applicable* ACTs. Relevance implies only that the Cue of the act matches the designated goal expression.

- Definitions in the Functions file are not limited to a single PRS agent but rather apply to the entire Lisp environment, and hence all agents.
- The `REBIND` function can only be used for goals of the form `(ACHIEVE (= (REBIND X.1) ...))` and only on plot nodes.
- The **Clear Ints** command from the **Execution** menu resets only *intentions*, and only for the *current agent*. The database and Act libraries remain unchanged.
- When multiple goals are posted in the same cycle, only the one which the agent attempts to achieve will remain. All others are discarded unless a Meta-Act such as `All Goals and All Acts` has been loaded (see Chapter 9).

13.3 Limitations and Known Bugs

Below is a summary of limitations and known problems with this version of PRS.

- The *negation as failure* capability is currently disabled within the system.
- Goals of the form

  ```
  (TEST (= <expn1> <expn2>))
  (ACHIEVE (= <expn1> <expn2>))
  (TEST (== <expn1> <expn2>))
  (ACHIEVE (== <expn1> <expn2>))
  ```

  do not support the use of evaluable functions within `<expn1>`.
- The first few lines of text sent to auxiliary windows are not always displayed, due to problems with CLIM.
- The graphic trace of procedures may leave residue from previously highlighted nodes on the graph pane, due to problems with CLIM.
- At times, the interactor plane may display the message:

  ```
  The input ‘‘’’ is not a complete Lisp Expression.  Please edit your input.
  ```

  This situation generally results from having typed extra Returns in the interactor pane. At this point, you have entered an odd editing mode of CLIM. Continuing to type Return will not help. Instead, the user should mouse-click right on the interactor pane to exit the mode and continue working in the pane. Alternatively, selecting any mouse-activated commands from the graph or command panes will simultaneously initiate the command and exit the editing mode.

Appendix A

Demonstration Domains

This appendix describes three applications of PRS to sample problems in relatively limited domains. These applications have been designed to help new users of the system familiarize themselves with the workings of PRS. The Delivery application provides a very basic agent for responding to events and goals posted by a user. The Simulation application shows a more sophisticated example of PRS usage, involving the synchronized coordination of several PRS agents. The Factorial application illustrates the use of metalevel reasoning.

A.1 Delivery Demonstration

This demonstration creates a single PRS agent named DELIVERY that processes requests to move goods between various sites. The initial database for the agent contains information about certain known companies (such as their their location) and products.

A.1.1 Loading the Demonstration

To load the demo, execute the following in the Lisp Listener window of the PRS GUI:

```
(load "~prs/beta/demos/delivery/demo.lisp)
```

The files used in this demo are:

```
delivery.graph          - the Acts that implement the DELIVERY agent
delivery-db.lisp        - the initial database for the agent
delivery-fns.lisp       - functions and declarations for the agent
```

A.1.2 Running the Demonstration

To request a delivery by the system, post a goal to the DELIVERY agent of the form:

```
(ACHIEVE (DELIVER customer.1 goods.1))
```

where customer.1 is the name of a customer, and goods.1 is the name of the goods to be delivered. Additional customers can be added to the demo by asserting facts of the form

```
(NEW-CUSTOMER customer.1 location.1)
```

to the DELIVERY agent's database.

A.2 Simulation Demonstration

This demonstration provides a simple multi-agent application of PRS. The demo contains three agents: an AIRPLANE agent, a SUBMARINE agent, and a TIME-MANAGER agent. The TIME-MANAGER is responsible for maintaining a simulated clock, and keeping the other agents synchronized with that clock. The AIRPLANE and SUBMARINE agents have initial speeds and headings defined in their databases, and update their trajectories based on this information as the simulation progresses.

A.2.1 Loading the Demonstration

To load the demo, execute the following in the Lisp Listener window of the PRS GUI:

```
(load "~prs/beta/demos/simulation/demo.lisp)
```

The files used in this demo are:

```
airplane.graph                  - the Acts for the AIRPLANE agent
airplane-database.lisp          - the AIRPLANE agent's initial database

submarine.graph                 - the Acts for the SUBMARINE agent
submarine-database.lisp         - the SUBMARINE agent's initial database

time-manager.graph              - the Acts for the TIME-MANAGER agent
time-manager-database.lisp      - the TIME-MANAGER agent's initial database

shared-database.lisp            - database facts shared by all agents
shared-functions.lisp           - functions and declarations shared by all agents

message-handler.graph           - Acts that define procedures to process messages received
                                  from other agents (used by all agents)
```

A.2.2 Running the Demonstration

The following goals can be posted to the TIME-MANAGER agent to interact with the demo:

`(ACHIEVE (BEGIN-SIMULATION))` begin the simulation, or restart the simulation if it has been halted.

`(ACHIEVE (HALT-SIMULATION))` halt the simulation

`(ACHIEVE (ADVANCE-TIME time.1))` advance the simulated clock to some future time.

A.3 Factorial Demonstration

This demonstration illustrates the use of metalevel reasoning for controlling the application of Acts. It consists of a single agent named FACTORIAL, which can compute factorial in both iterative and recursive fashion. A metalevel Act decides which approach to use, based on user preferences that are stated in the agent's database.

A.3.1 Loading the Demonstration

To load the demo, execute the following in the Lisp Listener window of the PRS GUI:

```
(load "~prs/beta/demos/factorial/demo.lisp)
```

The files used in this demo are:

```
factorial.graph          - the Acts that implement the FACTORIAL agent
factorial-db.lisp        - the initial database for the agent
factorial-fns.lisp       - functions and declarations for the agent
```

A.3.2 Running the Demonstration

To request that the agent compute the factorial of a number, post the goal:

```
(ACHIEVE (PRINT-FACTORIAL n.1))
```

To indicate a preference for either the ITERATIVE or RECURSIVE algorithms, assert one of the following into the agent's database:

```
(PREFER-ITERATIVE)
(PREFER-RECURSIVE)
```

A.3.3 Discussion

Figure A.1 shows the Act for this domain that can be used to compute the factorial of a number `N.1`. Actually, the previous statement is not quite true. Rather, this Act tries to satisfy the goal `(ACHIEVE (factorial N.1 RESULT.1))`. If you post the goal `(ACHIEVE (factorial 5 120))`, it will succeed, but if you post `(ACHIEVE (factorial 5 0))` it will fail (unless the fact `(FACTORIAL 5 0)` has been added to the PRS database). If you post the goal `(factorial 5 X.1)` it will succeed with a binding of 120 for `X.1`.

The Cue for this Act is `(ACHIEVE (factorial N.1 RESULT.1))`, indicating that the Act can be used in response to goals involving the factorial of a number. The body implements an iterative algorithm for computing the factorial of a number, which involves repeatedly considering successively smaller values for the value of the factorial function. Dynamic variables are required both to keep track of the countdown from `N.1` to 1 (i.e., the variable `TMP.1`) and the result value being constructed (i.e., the variable `RES.1`). Note the use of `REBIND` in the nodes `N4617` and `N4894`.

If this Act is invoked twice for the same goal, it will not recompute the factorial value (because the previous value has been concluded in the database and is still there). However, the Act does not store intermediate values.

Figure A.2 shows an Act that computes Factorial in a recursive manner. It is similar to the iterative Act, but uses a recursive subgoal rather than dynamic variables. When invoked, this Act concludes `Factorial` facts into the database for all of the intermediate values. When you invoke it again with higher values for `N.1` and `RESULT.1`, the previous values are not recomputed.

Figure A.3 shows the Act that can be used to print the value of factorial. It simply posts the goal `(ACHIEVE (factorial X.1 N.1))` and prints `N.1`.

After loading the domain, try posting the goal `(ACHIEVE (print-factorial X.1)` for different values of `X.1`. The factorial Act that is used will sometimes be the recursive one and sometimes the iterative one. What makes the system choose one instead of the other? The system at this point chooses *randomly*. The choice of Act does not matter for such a simple scenario but in many cases the user may want to control which of the applicable Acts is selected.

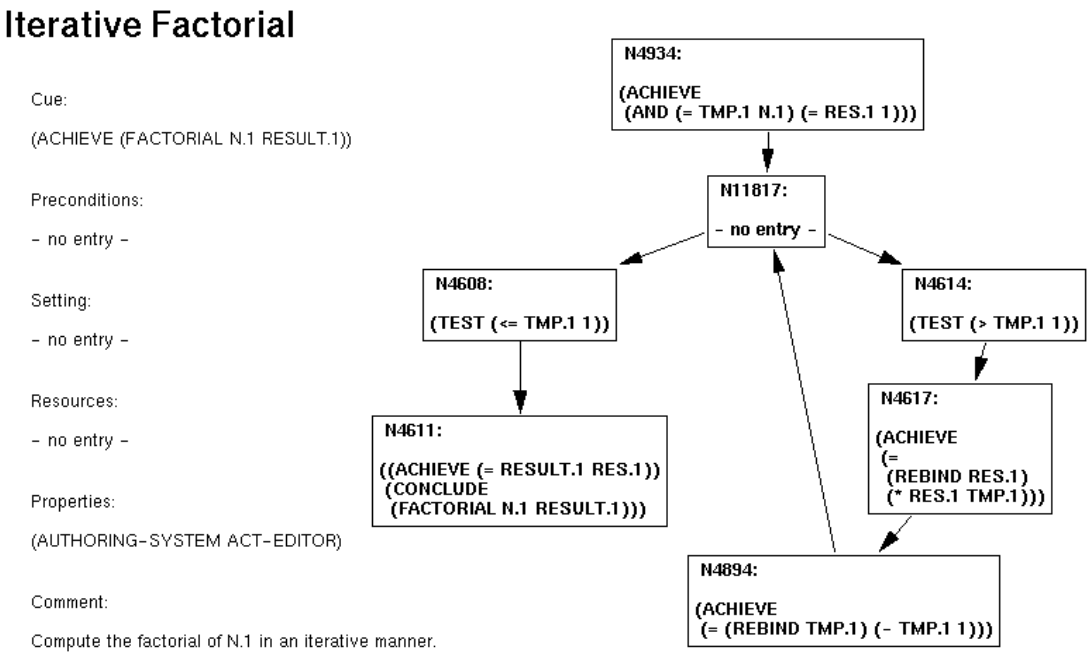

Figure A.1: The Act `Iterative Factorial`

Suppose one wishes to let the user specify whether he or she wants to use the iterative or recursive Act by adding appropriate information to the database: if `(prefer-recursive)` is in the database then the recursive Act should be used, and if `(prefer-iterative)` is in the database then the iterative Act should be used. If no preference is stated, then PRS should simply choose randomly. This strategy is implemented by the Act `Meta Selector: Iterative vs Recursive` in Figure A.4.

This Act is applicable whenever there are two applicable KAs/Acts with the same goal. The plot has three branches, corresponding to the three situations: `(prefer-recursive)`, `(prefer-iterative)` and no preference. In this last case, we select one of the two Acts randomly through the subgoal `(ACHIEVE (INTEND (SELECT-RANDOMLY X.1)))`. Otherwise, we test which one has the property `(RECURSIVE T)` (note that `PROPERTY-P` is a predefined evaluable predicate) and select it or the other depending on whether the preference is for a recursive or iterative solution. Note that the Act `Recursive Factorial` has the property `(RECURSIVE T)` while the Act `Iterative Factorial` has no such property. In order to produce the meta-selection behavior described above, it is also necessary to declare `(prefer-recursive)` and `(prefer-iterative)` as closed-world predicates.

Recursive Factorial

Cue:

(ACHIEVE (FACTORIAL N.1 RESULT.1))

Preconditions:

- no entry -

Setting:

- no entry -

Resources:

- no entry -

Properties:

((RECURSIVE T) (AUTHORING-SYSTEM ACT-EDITOR))

Comment:

Compute the factorial of N.1 in a recursive manner.

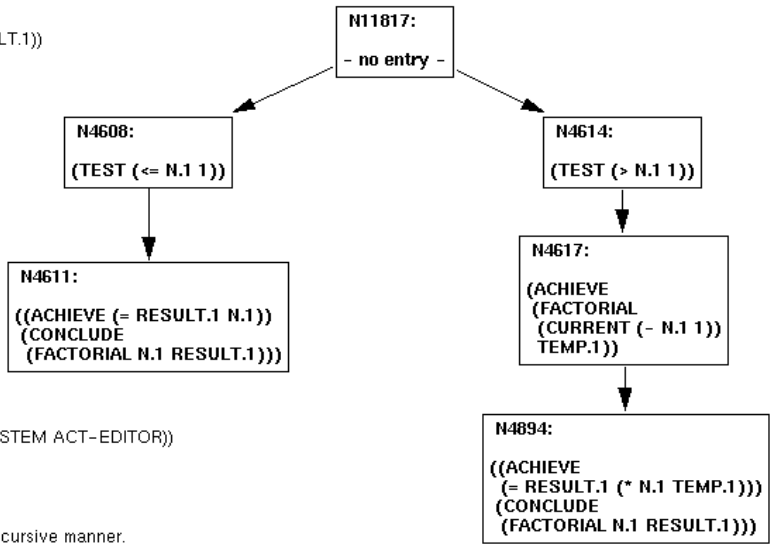

Figure A.2: The Act `Recursive Factorial`

Print Factorial

Cue:

(ACHIEVE (PRINT-FACTORIAL N.1))

Preconditions:

- no entry -

Setting:

- no entry -

Resources:

- no entry -

Properties:

(AUTHORING-SYSTEM ACT-EDITOR)

Comment:

Compute the Factorial of N.1 and print the result.

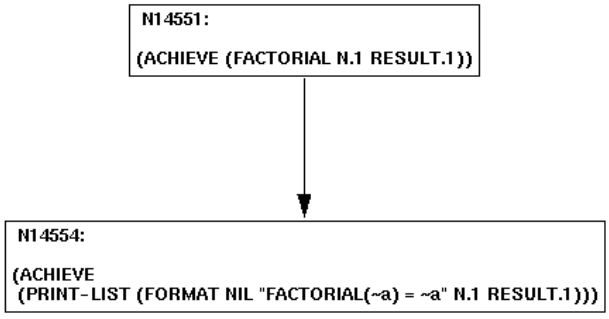

Figure A.3: The Act `Print Factorial`

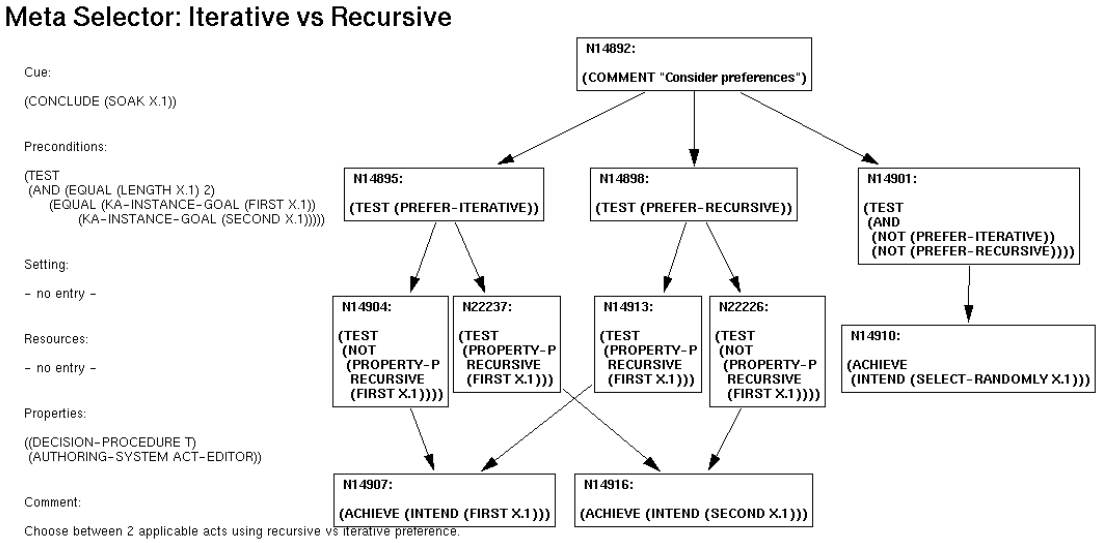

Figure A.4: The Act `Meta Selector: Iterative vs Recursive`

Appendix B

Default Acts

The ACTs described below are a default part of every PRS system. They are stored in the file:

~prs/released/graphs/act-default-processes.graph.

Each Act is summarized by indicating its Name, Cue, Preconditions (if any) and Documentation slots. None of the default Acts have entries in their Setting or Properties slots.

- **= (ACHIEVE)**

  Cue: `(ACHIEVE (= X.1 Y.1))`

  Documentation: Bind variable X.1 to the current value of variable Y.1

- **= (TEST)**

  Cue: `(TEST (= X.1 Y.1))`

  Documentation: Bind variable X.1 to the current value of variable Y.1

- **== (ACHIEVE)**

  Cue: `(ACHIEVE (== X.1 X.1))`

  Documentation: Unification: this act always succeeds if it is invoked (the unification for the CUE performs the necessary bindings).

- **== (TEST)**

  Cue: `(TEST (== X.1 X.1))`

  Documentation: Unification: this act always succeeds if it is invoked (the unification for the CUE performs the necessary bindings).

- **Apply ACHIEVE**

  Cue: `(ACHIEVE (APPLY (LAMBDA VAR.1 (ACHIEVE STATE.1)) VALUE.1))`

  Documentation: This Act makes a syntactic change in the goal by replacing the variable with its value, then posts the resulting Goal.

- **Apply TEST**

  Cue: `(ACHIEVE (APPLY (LAMBDA VAR.1 (TEST STATE.1)) VALUE.1))`

  Documentation: This Act makes a syntactic change in the goal by replacing the variable by its value, then posts the resulting Goal.

- **Apply to All**

  Cue: `(ACHIEVE (APPLY-TO-ALL ACTION.1 LIST.1))`

  Documentation: Apply the same goal to a list of variables.

- **Apply to All in Parallel**

  Cue: `(ACHIEVE (//-APPLY-TO-ALL ACTION.1 LIST.1))`

  Documentation: Apply the same goal to a list of variables, intending them in parallel.

- **Apply to All in Parallel (with priority)**

  Cue: `(ACHIEVE (//-APPLY-TO-ALL-WITH-PRIORITY ACTION.1 LIST.1 PRIORITY-LIST.1))`

  Documentation: Apply the same goal to a list of variables, intending them in parallel with priorities.

- **Meta Conjunction (ACHIEVE)**

  Cue: `(ACHIEVE (AND X.1 Y.1))`

  Documentation: Satisfy a conjunction of ACHIEVE goals by satisfying each ACHIEVE goal in sequence.

- **Meta Conjunction (TEST)**

  Cue: `(TEST (AND X.1 Y.1))`

  Documentation: Satisfy a conjunction of TEST goals by satisfying each TEST goal in sequence.

- **Meta Conjunction 3 Args (ACHIEVE)**

  Cue: `(ACHIEVE (AND X.1 X.2 X.3))`

  Preconditions: `NIL`

  Documentation: Satisfy a conjunction of three ACHIEVE goals by satisfying each ACHIEVE goal in sequence.

- **Meta Disjunction (ACHIEVE)**

  Cue: `(ACHIEVE (OR X.1 Y.1))`

  Documentation: Satisfy a disjunction of ACHIEVE goals by trying the ACHIEVE goals in sequence until one is satisfied.

- **Meta Disjunction (TEST)**

  Cue: `(TEST (OR X.1 Y.1))`

  Documentation: TEST a disjunction by testing the individual disjuncts until one is satisfied.

- **Meta Negation as Failure (ACHIEVE)**

  Cue: `(ACHIEVE (NOT X.1))`

  Preconditions: `(TEST (KNOW-HOW-TO-NEGATE-P X.1))`

  Documentation: Try to fail a TEST goal for a predicate that can be negated.

- **Meta Negation as Failure (TEST)**

  Cue: `(TEST (NOT X.1))`

Preconditions: `(TEST (KNOW-HOW-TO-NEGATE-P X.1))`

Documentation: Try to fail an ACHIEVE goal for a predicate that can be negated.

- **Print**

  Cue: `(ACHIEVE (PRINT X.1))`

  Documentation: Print the value of X.1 to the Output Window.

- **Query (format prompt)**

  Cue: `(ACHIEVE (QUERY X.1 Y.1))`

  Documentation: Query the user for a binding for variable Y.1, using the format statement X.1 to prompt.

- **Read**

  Cue: `(ACHIEVE (READ X.1))`

  Documentation: Read a value from the Input window and bind it to variable X.1

- **Send and Wait**

  Cue: `(ACHIEVE (SEND-AND-WAIT REC.1 REQ.1 GOAL.1))`

  Documentation: Send a request to another agent, then wait for a response.

- **Sleep**

  Cue: `(ACHIEVE (SLEEP X.1))`

  Documentation: Put the current intention to sleep for X.1 seconds.

- **Test and Set**

  Cue: `(ACHIEVE (TEST-AND-SET GOAL.1 RESULT.1))`

  Documentation: Try to achieve a goal and bind RESULT.1 to T or NIL depending on whether the goal succeeds.

- **Timed Wait and Set**

  Cue: `(ACHIEVE (TIMED-WAIT-AND-SET CONDITION.1 T.1 VALUE.1))`

  Documentation: Wait for CONDITION.1 to be true or T.1 seconds to expire. If the condition is satisfied, return VALUE.1 else NIL.

- **Timed Wait Until**

  Cue: `(ACHIEVE (TIMED-WAIT-UNTIL CONDITION.1 T.1))`

  Documentation: Wait for CONDITION.1 to hold. If T.1 seconds elapse first, then the ACT fails.

Appendix C

Primitive Actions

The predefined primitive actions in PRS are summarized below, grouped according to their functionality.

Input/Output

These predefined actions are used for input/output purposes.

`(print-format <format-str> &rest arguments)` PRS version of the LISP format function.

`(print-list <format-stmt>)` Print a LISP format statement to the Output Window. (Note: this primitive action has been kept for backward compatibility, but `print-format` is the recommended function for printing format statements.)

`(print-warning <format-stmt>)` Print a LISP format statement to the Output Window and beep to notify the user.

Intention Manipulation

The following primitive actions can be used to manipulate intentions and the intention graph. They are used primarily in meta-Acts written to implement domain-specific control mechanisms.

`(add-activation-condition-current-intention <condition>)` Add an activation condition to the current intention.

`(asleep-current-intention-from-ka)` Put the current intention to sleep (If the activation condition of the intention is T, then the intention will not go to sleep.)

`(intend <ka-instance>)` `Intend a ka-instance.`

`(intend-with-priority <ka-instance> <priority>)` Intend a ka with a specific priority (see Section 8.3 for details on prioritizing intentions).

`(intended-all <ka-instance-list>)` Intend a list of ka-instances.

`(intended-all-as-root <ka-instance-list>)` Intend a list of ka-instances as roots of the intention graph.

`(intended-all-as-root-before-me <ka-instance-list>)` Intend a list of ka-instances as roots of the intention graph and before the current intention.

(intended-all-last <ka-instance-list>) Intend a list of ka-instances, putting them at the end of the intention graph.

(intended-all-before-n <ka-instance-list> <intention>) Intend a list of ka-instances, putting them before <intention> but after all predecessors of ¡intention¿.

(intended-all-after-n <ka-instance-list>) Intend a list of ka-instances, putting them after <intention> but before all successors of ¡intention¿.

(intended-all-goals-//-before-me <goal-list>) Create new root intentions for each goal.

(intended-all-goals-//-before-me-with-priority <goal-list> <priority-list>) Create new prioritized root intentions for each goal.

(kill-intention ¡intention¿ &OPTIONAL (prs *current-prs*)) Eliminate a given intention, along with all of its descendants. (NOTE: caution is required when using this function because of possible interactions with other intentions. For example, another intention may be waiting on the killed intention to complete.)

Others

The following predefined actions perform an assortment of basic activities within the system.

(*=* <var> <fn-call>) For assignment to variables. The first argument must be a variable, the second argument a ground function call. This primitive action is guaranteed to invoke any evaluable functions in ¡fn-call¿ only once; as such, it is the recommended way to bind variables. Using the default Acts to bind variables can lead to multiple invocations of evaluable functions, since there may be multiple attempts to satisfy the overall equality-based goal.

(send-message <agent> <message>) Send a message to a PRS agent.

(try-to-fail-achieve <goal>) Try to fail an ACHIEVE goal.

(try-to-fail-test <goal>) Try to fail a TEST goal.

Internal Predefined Actions

The remaining predefined actions predicates are employed internally by the default ACTs in PRS. As such, they are not for general use.

(print-io-pane <value>) Print a value to the Output Window.

(selected <arc-list>) When the PRS variable *TURBO-PRS* is set to T, a goal of the form (ACHIEVE (SELECTED <arc-list>) is posted by PRS whenever there is more than one outgoing arc from a disjunctive node. The primitive action will choose from among these arcs.

Appendix D

Evaluable Predicates and Functions

Predicates

`EQ, EQUAL, >, <, >=, <=, NULL, LISTP, ATOM` (as in Lisp)

`(KNOW-HOW-TO-NEGATE-P <goal>)` Determines if the predicate in the goal has been declared as one for which negation-as-failure applies.

`(ELAPSED-SECONDS-P <time> <seconds>)` Tests whether time elapsed since `<time>` is greater than `<seconds>` seconds.

`(PROPERTY-P <ka>)` Return the property slot for KA/Act

`(VAR-PROPERTY-P <ka>)` Return the property slot for KA/Act, with all variable bindings in place

`(NO-OP)` Dummy predicate that always evaluates to T (used internally by PRS)

`(AVAILABLE-RESOURCE <resource>)` Determines resource availability (for internal use only)

Functions

`+, -, *, ABS` (as in Lisp)

`LENGTH, FIRST, SECOND, THIRD, REST, CONS, CAR, CDR, REVERSE, TIME` (as in Lisp)

`MY-TIME` returns the simulated time.

`(SECONDS-TIME <seconds>), (TIME-SECONDS <time>)` converts between numbers of seconds and times (which may be in arbitrary units).

`(ALL <var> <expression>)` returns all values of `<var>` for which `<expression>` is true

`(DECISION-PROCEDURE-OF <ka>)` determines whether a KA has the decision procedure property.

`(FACT-INVOKED-KAS-OF <ka-list>)` returns the fact-invoked kas from a list of ka instances

`(GOAL-INVOKED-KAS-OF <ka-list>)` returns the goal-invoked kas from a list of ka instances

`(MY-TIME)` returns the current PRS time

`(SELECT-RANDOMLY <list>)` randomly chooses an element of `<list>`

`(SELECT-RANDOMLY-FOR-EACH-GOAL <ka-instance-list>)` for each distinct goal addressed by KA-Instances in the list, select one of the KA-instances randomly.

`(SELECT-KA-FOR-EACH-GOAL <ka-instance-list> <selection-fn>` for each distinct goal addressed by KA-Instances in the list, select one by applying `selection-fn`.

Appendix E

Important Variables and Functions

This appendix describes some important variables, and functions that the user might want to consult or employ.

E.1 Variables

PRS-OBJECTS: List of PRS objects (agents) in the system.
PRS-DEMOS: List of PRS demo objects in the system.
EVALUABLE-FUNCTIONS: List of evaluable functions.
EVALUABLE-PREDICATES: List of evaluable predicates.
CLOSED-WORLD-PREDICATES: List of closed world predicates.
FUNCTIONAL-PREDICATES: List of functional predicates.

NEGATION-AS-FAILURE: List of predicates for which negation as failure applies.

MONITOR-MESSAGES: If set to T, the messages sent to each agent are traced in the Trace Window. (Default is NIL).

MESSAGES-TYPE: List of recognized message classes (see section 11.2).

BASIC-EVENTS: List of predicates recognized as basic events (see section 3).

NOT-SUBSUMED-GOAL: If T, goals that are subsumed in the same intention stack are not tried again. (Default is nil)

STANDARD-PROCESSES-FILE: The name of the default processes (ACTs) file. The standard file is ~prs/released/graphs/act-default-processes.graph.

UNIQUE-ACT/KA-NAMES When NIL, multiple Acts/KAs with the same name can be loaded into a single agent; when non-NIL, newly loaded Acts/KAs will overwrite previously loaded versions. (Default is T.)

FLASH-TIME: The number of seconds that an Act node flashes during graphic tracing. (Default is .4)

TURBO-PRS: Enable (nil) or disable (t) the posting of a goal to select which arc to choose after one node.

E.2 Functions

All functions and methods described in this section are given without their body. Only their name and the necessary declaration part is given.

`(EVALUABLE-FUNCTIONS new-functions)`: Append the list new-functions to the list of evaluable functions.
`(EVALUABLE-PREDICATES new-predicates)`: Append the list new-predicates to the list of evaluable predicates.
`(CLOSED-WORLD-PREDICATES new-cw-predicates)`: Append the list new-cw-predicates to the list of closed world predicates.
`(NEGATION-AS-FAILURE new-predicates)`: Append the list new-predicates to the list of negation as failure predicates.
`(FUNCTIONAL-PREDICATES new-predicate-pairs)`: Append the list new-predicates-pairs to the list of functional predicates.

`(FIND-PRS-NAMED name)`: Return the PRS Object named with name.
`(INTERFACED-PRS)`: Return the PRS that is currently interfaced to the PRS Window.
`(LOAD-DEFAULT-KAS &optional (file *STANDARD-PROCESSES-FILE*))`: Load the Default Processes.

`(INIT-TIME :START-TIME <time> :EXPANSION-RATE <rate>`
`:UNITS-PER-SECOND <units> :SAFE <safe?>)`: Initialises the PRS time (MY-TIME) and specifies an initial expansion rate (number of simulated seconds per real second), the units for the simulated time (how many simulated time units per second), and whether you want to ensure that reported times are non-decreasing, even when the clock is adjusted back using (SETF MY-TIME). A :UNITS-PER-SECOND of 60 (the default) means that each second of simulated time is represented by an increase of 60 in the value of (MY-TIME). For conversions between times and seconds, use the evaluable functions SECONDS-TIME and TIME-SECONDS. `(MY-TIME)`: Returns the PRS time (this should be the only reference to the time).
`(SETF (MY-TIME) <adjusted time>)`: Make small adjustments to keep the time in sync with an external clock (for resetting the time use INIT-TIME).
`(TIME-EXPANSION-RATE)`: Time evolution of the system: 1 mean same rate as real time, 2 means time goes twice as fast, 0.5 time slows down to half as fast. (Default is 1)
`(SETF (TIME-EXPANSION-RATE) <new rate>)`: Set expansion rate
`(STOP-TIME)`: Stop the PRS time.
`(RESTART-TIME)`: Restart the PRS time.

`(TIME-STOPPED-P)`: Test whether PRS time is stopped.

`(ASSERT-FROM-KA <wff>)` Implements the CONCLUDE operator.
`(RETRACT-FROM-KA <wff>)` Implements the RETRACT operator.
`(ACHIEVE-BY-FROM-KA <expn>)` Implements the ACHIEVE-BY operator.
`(REQUIRE-UNTIL-FROM-KA <expn>)` Implements the REQUIRE-UNTIL operator.

Appendix F

Application Program Interface (API)

F.1 Specifying the Agent

`CREATE-PRS-AGENT (agent-name)`
Creates an initialized PRS agent of the given name.

`DESTROY-PRS-AGENT (agent-name)`
Destroys the given PRS agent.

`AGENT? (name)`
Predicate returns `T` if an agent exists of the given name. Otherwise, returns `nil`.

`AGENT-NAME (agent)`
Given a PRS agent, returns its name

`SET-OF-AGENTS ()`
Returns list of the names of known PRS agents

`GET-PROCESS-FILES (prs-agent)`
Returns the list of processes used by a given agent

`INIT-PROCESSES-FROM-FILE (&optional filename prs-agent aux pathname package)`
Initializes the given PRS agent with processes from `filename`. If `prs-agent` is not specified, it uses the currently selected agent.

`INIT-PROCESSES-FROM-GRASPER (&optional prs-agent)`
Similar to `INIT-PROCESSES-FROM-FILE` except the processes come from the current Grasper space.

`APPEND-PROCESSES-FROM-FILE (&optional filename prs-agent aux package)`
Appends processes from `filename` to the given PRS agent If `prs-agent` is not specified, it uses the currently selected agent.

`APPENDS-PROCESSES-FROM-GRASPER (&optional prs-agent)`
Similar to `APPEND-PROCESSES-FROM-FILE` except the processes come from the current Grasper space.

`INIT-DATABASE-TO-EMPTY (&optional prs-agent-name)`
Initializes the database of the PRS agent with the given name. If no agent name is specified, it uses the currently selected agent.

`APPEND-DATABASE-FROM-FILE (prs-agent-name filename)`

Appends the contents of `filename` to the database of the specified PRS agent name or, if no agent is specified, the currently selected agent.

`APPEND-FUNCTIONS (filename)`
Loads functions from the LISP file specified by `filename`.

`CACHE-DATABASE (prs-agent)`
Caches database of given PRS agent

`DECACHE-DATABASE (prs-agent)`
Decaches database of given PRS agent

F.2 Execution

F.2.1 Facts, Messages, Goals, and Intentions

`RESET (prs-agent)`
Clears intentions and database of PRS agent

`RESET-INTENTION-LIST (prs-agent)`
Resets intention list of PRS agent

`RESET-FACT-DATA-BASE (prs-agent)`
Resets database of PRS agent

`POST-FACT-WITH-NOTICE (statement &optional prs)`
Posts a fact to specified or current PRS agent and prints a notice of that fact.

`CONSULT-DB (statement &optional prs)`
Returns a list of facts which satisfy the statement in the database of the given or current PRS agent.

`POST-NEW-GOAL (goal-statement &optional prs-agent`
Posts goal given by statement to specified or current PRS agent.

`PRINT-INTENTION-LIST (&optional prs)`
Prints the intention list of the given PRS agent. If none is specified, uses the current PRS agent.

`PRINT-INTENTION-GRAPH (&optional prs)`
Prints the intention graph of the given PRS agent. If none is specified, uses the current PRS agent.

`CLEAR-ALL-INTS-ALL-AGENTS ()`
Clears all intentions in all known PRS agents.

`CLEAR-AGENT-MODE (&optional prs)`
Clears intentions of specified or current PRS agent.

`SEND-MESSAGE (to-name message)`
Send message to PRS agent with the given name

`BROADCAST-MESSAGE (to-list message)`
Send message to multiple PRS agents (given by a list of names).

F.2.2 Pausing, Stepping, and Going

`STEP-EXECUTE (&optional prs-agent)`
Executes one step of the specified PRS agent. Executes the currently selected agent if `prs-agent` is not specified.

`HALT-EXECUTE (&optional prs-agent)`
Pauses execution of a specified PRS agent. Pauses the current agent if `prs-agent` is not specified.

`GO-EXECUTE (&optional prs-agent)`
Continues execution of the specified PRS agent. Uses the currently selected agent if `prs-agent` is not specified.

`GENERIC-STEP-EXECUTE (&optional prs-agent)`
Like STEP-EXECUTE only without messages printed to PRS.

`GENERIC-HALT-EXECUTE (&optional prs-agent)`
Like HALT-EXECUTE only without messages printed to PRS.

`GENERIC-GO-EXECUTE (&optional prs-agent)`
Like GO-EXECUTE only without messages printed to PRS.

F.2.3 Tracing

`TRACE-KAS-FROM-GRAPH (graphs &optional (graphic-trace t) (text-trace t) prs-agent)`
Turn on tracing for given graphs in current or specified PRS agent.

`TRACE-KA-PROPERTY (predicate &optional (graphic-trace t) (text-trace t) prs)`
TRACE-KA-PROPERTY: turns on tracing for whichever acts have a property for which the predicate returns true.

`AGENT-TRACE-SELECTIONS (&optional prs)`
Returns list of current trace selections (Procedure-Graphic Procedure-Text ...)

`SET-INTENTION-TEXT-TRACE-OFF (&optional prs-agent)`
Turns off intention text tracing for specified or current PRS agent.

`SET-INTENTION-TEXT-TRACE-ON (&optional prs-agent)`
Turns on intention text tracing for specified or current PRS agent.

`SET-DATABASE-TRACE-OFF (&optional prs-agent)`
Turns off database tracing for specified or current PRS agent.

`SET-DATABASE-TRACE-ON (&optional prs-agent)`
Turns on database tracing for specified or current PRS agent.

`SET-RESOURCE-TRACE-OFF (&optional prs-agent)`
Turns off resource tracing for specified or current PRS agent.

`SET-RESOURCE-TRACE-ON (&optional prs-agent)`
Turns on resource tracing for specified or current PRS agent.

`SET-MESSAGE-TRACE-OFF (&optional prs-agent)`

Turns off message tracing for specified or current PRS agent.

`SET-MESSAGE-TRACE-ON (&optional prs-agent)`
Turns on message tracing for specified or current PRS agent.

`SET-PROCEDURE-TEXT-TRACE-OFF (&optional prs-agent)`
Turns off procedure text tracing for specified or current PRS agent.

`SET-PROCEDURE-TEXT-TRACE-ON (&optional prs-agent aux kas)`
Turns on procedure text tracing for specified or current PRS agent.

`SET-PROCEDURE-GRAPH-TRACE-OFF (&optional prs-agent)`
Turns off procedure graph tracing for specified or current PRS agent.

`SET-PROCEDURE-GRAPH-TRACE-ON (&optional prs-agent aux kas)`
Turns on procedure graph tracing for specified or current PRS agent.

`SET-INTENTION-GRAPH-TRACE-OFF (&optional prs-agent)`
Turns off intention graph tracing for specified or current PRS agent.

`SET-INTENTION-GRAPH-TRACE-ON (&optional prs-agent)`
Turns on intention graph tracing for specified or current PRS agent.

`TURN-OFF-TRACING (&optional agents)`
Turn off all tracing for PRS agents specified by agents. Default is all agents.

`NO-AGENT-TRACE (&optional prs)`
Turn off tracing for the specified PRS agent. If none is specified, turns off tracing for the current PRS agent.

F.3 Input and Output

`INITIALIZE-POPUP-WINDOW (label)`
Returns a stream to which one can write (using `format` statements. See also `DISPLAY-POPUP-WINDOW`.

`DISPLAY-POPUP-WINDOW (window-stream width height)`
Displays the window from the given stream (see `INITIALIZE-POPUP-WINDOW`) with dimensions width and height.

`PRS-DISPLAY (title string-list key (width 800) (height 240))`
Pops up a window with title with the list of strings (on consecutive lines) with the given dimensions. Left click closes the window.

`PRS-NOTIFY (thing &optional beep)`
Prints thing to the LISP listener window and beeps if specified.

`PRS-PRINT (item)`
Prints `item` to the I/O window of current PRS agent.

`PRINT-IO-PANE (item)`
Prints item to the IO window of current prs.

`PRS-TRACE (string)`

Prints string to the trace window of current PRS agent

`PRINT-LIST (expression)`
Evaluates LISP expression and prints the result in I/O window of current PRS agent.

F.4 The Demo Facility

`FIND-PRS-DEMO-NAMED (name)`
Finds a PRS demo given its name

`GET-PROCESS-FILES (prs-demo)`
Returns the list of processes used by the agents in a PRS demo system.

`REINITIALIZE (prs-demo)`
Reinitializes all agents and simulators in a given demo

`STOP-BUT-NOT-SIMULATORS (prs-demo)`
Stop all agents in a demo

`STOP (prs-demo)`
Stop all agents and simulators in a demo

`KILL (prs-demo)`
Kill a demo

`RESTART (prs-demo)`
Restart a demo.

F.5 Global Parameters

`PRIMITIVE-ACTIONS (new-primitive-actions)`
Add primitive actions to PRS.

`EVALUABLE-FUNCTIONS (new-functions)`
Add evaluable functions to PRS.

`FUNCTIONAL-PREDICATES (new-functions)`
Add functional predicates to PRS.

`BASIC-EVENTS (new-functions)`
Add basic events to PRS.

`EVALUABLE-PREDICATES (new-predicates)`
Add evaluable predicates to PRS.

`CLOSED-WORLD-PREDICATES (new-cw-predicates)`
Add closed world predicates to PRS.

`NEGATION-AS-FAILURE (new-predicates)`
Add predicates to negate as failure. What does that do?

`MESSAGES-TYPE (new-predicates)`

Add new message type

`APPEND-ACT-GRAPH-FROM-KA (graph-name &optional prs)`
Append a graph file to the current or specified agent

F.6 Misceallaneous Functions

`PRS-PKG (x)`
Places and returns the symbol `x` in the PRS package.

`RUN-LOADED-PRS (key width height top left)`
Runs the PRS GUI from a LISP image. Default dimensions are `(width 1050) (height 850) (top 10) (left 50)`

`FIND-PRS-NAMED (name)`
Given the name of a PRS agent, returns the object of that agent

`WITH-PRS-AGENT (agent rest rest)`
Performs actions with given PRS agent

`KA-LIST (prs-agent)`
Returns the list of KA structures in given PRS agent

`REINITIALIZE (prs-agent)`
Clears the given agent of all intentions and restores its database from the cache.

Bibliography

[1] M. P. Georgeff. Actions, processes, and causality. Technical note, Artificial Intelligence Center, SRI International, Menlo Park, California, U.S.A, 1986.

[2] M. P. Georgeff. Future impact of intelligent machines on space operations. In *24th Goddard Memorial Symposium*, Washington, DC, 1986. American Astronautical Society.

[3] M. P. Georgeff and F. F. Ingrand. Research on procedural reasoning systems. Final Report, Phase 1, for NASA Ames Research Center, Moffet Field, California, U.S.A, Artificial Intelligence Center, SRI International, Menlo Park, California, U.S.A, October 1988.

[4] M. P. Georgeff and F. F. Ingrand. Decision-making in an embedded reasoning system. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 972–978, Detroit, Michigan, U.S.A, 1989.

[5] M. P. Georgeff and F. F. Ingrand. Monitoring and control of spacecraft systems using procedural reasoning. In *Proceedings of the Proceedings of the Space Operations-Automation and Robotics Workshop*, Houston, Texas, 1989.

[6] M. P. Georgeff and F. F. Ingrand. Real-time reasoning: The monitoring and control of spacecraft systems. In *Proceedings of the Sixth IEEE Conference on Artificial Intelligence Applications*, Santa Barbara, California, U.S.A, March 1990.

[7] M. P. Georgeff and F. F. Ingrand. Research on procedural reasoning systems. Final Report, Phase 2, for NASA Ames Research Center, Moffet Field, California, U.S.A, Artificial Intelligence Center, SRI International, Menlo Park, California, U.S.A, June 1990.

[8] M. P. Georgeff and A. L. Lansky. Development of an expert system for representing procedural knowledge. Final Report, for NASA Ames Research Center, Moffet Field, California, U.S.A, Artificial Intelligence Center, SRI International, Menlo Park, California, U.S.A, December 1985.

[9] M. P. Georgeff and A. L. Lansky. Procedural knowledge. *Proceedings of the IEEE Special Issue on Knowledge Representation*, 74:1383–1398, 1986.

[10] M. P. Georgeff and A. L. Lansky. A system for reasoning in dynamic domains: Fault diagnosis on the space shuttle. Technical Note 375, Artificial Intelligence Center, SRI International, Menlo Park, California, U.S.A, 1986.

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

[12] M. P. Georgeff, A. L. Lansky, and P. Bessiere. A procedural logic. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, California, U.S.A, 1985.

[13] M. P. Georgeff, A. L. Lansky, and M. Schoppers. Reasoning and planning in dynamic domains: an experiment with a mobile robot. Technical note 380, Artificial Intelligence Center, SRI International, Menlo Park, California, U.S.A, 1987.

[14] F. F. Ingrand and M. P. Georgeff. Managing deliberation and reasoning in real-time ai systems. In *Proceedings of the 1990 DARPA Workshop on Innovative Approaches to Planning*, Santa Diego, California, U.S.A, November 1990.

[15] F. F. Ingrand, J. Goldberg, and J. D. Lee. SRI/Grumman Crew Members' Associate Program: Development of an Authority Manager. Final Report, Artificial Intelligence Center, SRI International, Menlo Park, California, U.S.A, 1989.

[16] P. D. Karp, J. D. Lowrance, and T. M. Strat. *The Grasper-CL Documentation Set.* Artificial Intelligence Center, SRI International, 333 Ravenswood Avenue, Menlo Park, CA, June 1993.

[17] P. D. Karp, J. D. Lowrance, T. M. Strat, and D. E. Wilkins. The Grasper-CL graph management system. *LISP and Symbolic Computation: An International Journal*, 7:251–290, 1994.

[18] J. D. Lowrance. *Evidential Reasoning with Gister-CL: A Manual.* Artificial Intelligence Center, SRI International, 333 Ravenswood Avenue, Menlo Park, CA, June 1996.

[19] K. L. Myers. *The ACT Editor User's Guide.* Artificial Intelligence Center, SRI International, Menlo Park, CA, March 1997.

[20] D. E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm.* Morgan Kaufman, 1988.

[21] D. E. Wilkins. *Using the SIPE-2 Planning System: A Manual for Version 4.3.* Artificial Intelligence Center, SRI International, Menlo Park, CA, August 1993.