UM-PRS V3.0
Programmer and User Guide

Marcus J. Huber, Jaeho Lee, Patrick Kenny, Edmund H. Durfee
Artificial Intelligence Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, MI 48109

{marcush,jaeho,pkenny,durfee}@engin.umich.edu

August 14, 1995

Abstract

The University of Michigan Procedural Reasoning System (UM-PRS) is an object–oriented (C++) implementation of PRS as described in [1, 2, 3, 4, 6] and other papers. This document's primary purpose is to explain everything that is required to create an UM-PRS application. To a lesser extent, the document will familiarize readers to PRS's concepts. For a more complete discussion of the theoretical and philosophical issues, reading the original research papers and the UM-PRS paper [5] is recommended.

[0]This research was sponsored in part by ARPA under contract DAAE-07-92-C-R012 and in part by ORINCON under contract 951546.

Contents

1 Introduction

UM-PRS is composed of four components: a database representing the current world model, a library of plans called Knowledge Areas (KAs), a primitive function library and primitive function interface for performing low–level functions, and an intention structure that maintains the runtime state of the set of currently active goals. The user deals explicitly with the first three of these components. This document describes the purpose of each of these components, how they interact, any files that are necessary, and any related representation and syntax. The system's functionality, which in part depends upon the intention structure, is also described. Throughout the document we describe any of the constraints and limitations of the current implementation. A demonstration system, including of all of the necessary components for a simple, but complete, running system, is included with the UM-PRS package.

The UM-PRS execution cycle is very similar to that originally discussed for PRS in [1, 2, 3, 4, 6]. Changes to the world model or posting of new goals triggers the system to search for all valid (based upon the goal and context) KAs that might be used to deal with the situation. The system interpreter may then select one KA from this collection, called the SOAK (Set Of Applicable Kas), and *intend* it, which places the instantiated KA (it has a variable binding particular to the situation) onto the UM-PRS *intention structure*, the system's multi-goal runtime stack. Alternatively, the system may continue to execute the intention currently on the intention structure. This is discussed in more depth in Section 10.

2 World Model

The world model holds the facts that represent the current state of the world. Information that might be kept there includes state variables, sensory information, conclusions from deduction or inferencing, modeling information, etc. Each world model entry is of the form:

relation-name *argument1 argument2 ... argumentN;*

Each relation with the string identifier **relation-name** has an user-specified implicit ordering and semantic label for each argument in the relation. For example, a relation with the name **tree** might specify the height, species, and color of a tree, such as in:

tree *20 "Maple" "Red";*

with the implicit parameter ordering of height, then species, then color. The order and content of a relation is completely up to the discretion and control of the user. Arguments may consist of strings, floating point numbers, integer numbers, or pointers to arbitrarily complex structures (`void` pointers).

Specification of the world model consists of creating a text file containing the keyword **FACTS:** followed by the list of initial world model relations. UM-PRS parses the world model specification before execution and is furnished to the planning system as described in Section 9. An example world model specification, drawn from a robotics application, might be as shown below:

```
FACTS:
        robot_status "Ok";
        partner_status "Ok";
        robot_initialized "False";
        robot_localized "False";
```

```
robot_registered "False";
robot_position 10000 10000 0;
robot_location "Unknown";
self "CARMEL";
partner "BORIS";
object_found "False";
object_delivered "False";
comm_status "Ok";
plan_empty "False";
destination "Room4";
next_room "Room3";
next_node "Node12";
```

3 Knowledge Areas

A Knowledge Area (KA) defines a procedural method for accomplishing a goal. Its applicability is limited to a particular PURPOSE, or goal, and may be further constrained to a certain CONTEXT. The procedure to follow in order to accomplish the goal is given in the BODY. Each KA has a NAME component that can be used to distinguish between procedures. Another component of a KA is the optional DOCUMENTATION, which provides a generic place for the programmer to put commentary. The PRIORITY section is an optional component that can be used to influence selection of certain procedures over others through the meta-level reasoning mechanism of UM-PRS. Another optional component is the EFFECT section, which is identical in syntax to a KA BODY section, but is executed when UM-PRS is run in simulation mode (see Section 9.2). A third optional section, called the FAILURE section, can be used to specify a procedure to execute when the KA fails for some reason.

Throughout KAs, variables are represented by a text identifier preceded by a **$** symbol. Variables may be used throughout a KA, and are primarily used in actions in the KA BODY, in the KA CONTEXT, and in a KA's PURPOSE expression. Note that variables should never be used in the initial world model (the reason for this will become evident later). Each of the parts of a KA is described below in more detail. Before this, though, we discuss *expressions*, an important UM-PRS representation scheme.

3.1 Expressions

Expressions can be used in various places for run-time evaluation. First of all, *values* and *variables* are expressions. A value can be either an integer number, a floating point number, a `void` pointer, or a string. A function is also an expression. The syntax of function expressions follows the Lisp function format closely where the first element in them is a function to call while the rest are arguments to the function, all enclosed within parenthesis. For example, `(+ 3 1)` adds the numbers 3 and 1 together, while `(> $x 5)` returns `Value::True` if the variable `$x` is greater than 5, and `Value::False` if it is less than or equal to 5. More examples of valid `Expressions` are shown below.

```
$variable_name_1
12345
12.45
"abc def"
```

```
//
// arithmetics with a floating point number and an integer
//
(+ 0.123 -123)                // plus
(- 1 2.0 3 4.0)               // minus
(/ 30 2.0 3 -4)               // division
(* 2.3 3.4 4.5 5.6 6.7)       // multiplication
(% 12 3)                      // modulo

//
// string concatenation
//
(+ "abc" "efg" "\\" "\n" "\r" "\t" "\f")
(+ "\b" "\a" "\v" "Hex \xff" "Octal \77")

//
// Boolean
//
(== $a 10 $b)                 // equal
(!= "abc" $b "abc")           // not equal
(< 10 5)                      // less than
(<= 10.5 5)                   // less than or equal
(> $a 10 $b $c)               // greater than
(>= $a "hello" $b "world")    // greater than or equal

//
// Logical
//
(and (> $a 10) (< $a 100))    // and
(&& (> $a 10) (< $a 100))     // and
(or (> $a 10) (== $a 0))      // or
(|| (> $a 10) (== $a 0))      // or
(not (== $a 10))              // not
(! (== $a 10))                // not
```

Note that most functions allow mixed (integer and floating point) number operations. Boolean conditionals, however, cannot have mixed argument types, in general. It is illegal, for example, to compare a string with an integer.

Furthermore, most functions can have single or multiple arguments, and quite often more than two arguments. An example of this can be seen in the string concatenation examples above.

3.2 Body

The body of a KA describes the sequence of actions, a procedure, to be taken in order to accomplish a goal. The body may contain provided UM-PRS actions and user-defined primitive functions, and can be organized into branches which are conditional executed and loops which can be conditionally repeated. An example of a KA is shown below. It is not complete followed by a description of each of its components.

```
KA {
NAME:
        "Example KA"
DOCUMENTATION:
        "This is a nonsensical KA that shows all of the possible actions"
PURPOSE:
        ACHIEVE ka_example;
CONTEXT:
        FACT task_complete "False";
BODY:
  QUERY determine_task $task;
  FACT problem_solved $task $solved;
  OR
  {
    TEST (== $solved "YES");
    WAIT user_notified;
    RETRACT working_on_problem "True";
  }
  {
    TEST (== $solved "NO");
    ACHIEVE problem_decomposed;
    ATOMIC
    {
        ASSERT working_on_problem "True";
        MAINTAIN problem_decomposed;
    };
    ASSIGN $result (* 3 5);
  };
  UPDATE (task_complete) (task_complete "True");

FAILURE:
  UPDATE (ka_example_failed) (ka_example_failed "True");
  EXECUTE print "Example failed.  Bailing out"
}
```

3.2.1 Actions

Each action in a KA, one line of a KA's BODY, can specify a goal or condition to ACHIEVE, WAIT for, MAINTAIN, or QUERY. In addition, a KA action can be a low–level function to EXECUTE directly, an ASSERTion of a fact to the world model, a RETRACTion of a fact from the world model, an UPDATE of a fact in the world model, a FACT or a RETRIEVE statement that retrieves relation values from the world model, or an ASSIGN statement that assigns variables the results of run–time computations. Furthermore, iteration and branching are accomplished through WHILE, DO, OR, and AND actions. For convenience when testing KA failure (or for other reasons) there is also a FAIL action which is an action that always fails. An example of quite a few of these actions is included in the example KA above.

ACHIEVE *goal_name parameter1 parameter2 ... parameterN :PRIORITY expression;*

An ACHIEVE action causes the system to subgoal from the currently executing goal. This then triggers the system to search for KAs in the KA library that can satisfy the goal *goal_name* given the current context.[1] Parameters can be used as arguments to the goal to be achieved and can also be used to get return values back from the invoked KA. If the invoked KA fails, and the ACHIEVE was within a KA body, the ACHIEVE action fails as would any other KA body action. The failure semantics for system goals (i.e. top level goals) is different, however, and is explained in Section 5.

An ACHIEVE action can optionally specify the priority of the goal using the keyword :PRIORITY *expression* after the parameters. If the priority is not specified, it has the default priority 0. The details of how the goal priority is interpreted will be explained in Section 10.

Example: **ACHIEVE robot_homed $x $y $orient;**

Example: **ACHIEVE robot_homed $x $y $orient :PRIORITY (+ $x $y 10);**

AND { *active-sequence1* } ... { *action-sequenceN* }*;*
where *action-sequence* (also called a *branch*) is one or more of any of the actions listed here and $N > 0$. The semantics of an AND action is that the interpreter will execute each of the branches in turn, with the first branch listed executed first. If all of the branches succeed, then the action succeeds. If any branch fails, then the entire action fails.[2]

Example:

```
AND
{
    EXECUTE print $x;
}
{
    EXECUTE print $y;
};
```

ASSERT *relation_name argument1 argument2 ... argumentN;*
Information is placed into the world model using an ASSERT action. Assertion causes the system to search the world model for the given relation name. If it finds a world model entry with the same relation name and the same argument values, the action does nothing. Otherwise, the action *appends* the asserted fact, with all of the arguments evaluated into *values*, onto the world model. All variables and expressions must be evaluable to either a string, floating point number, an integer number, or an address (an integer).

Example: **ASSERT goal_in_view "True";**

NOTE: To *modify* a relation in the world model, use the UPDATE action, to be explained below.

[1] See Section /refsec:interpreter for more details about the UM-PRS interpreter execution cycle.

[2] Given the current execution semantics, there is little need, if any, to ever use an AND branch.

ASSIGN *variable expression;*
Variables can also be assigned values based on such things as mathematical computations using the ASSIGN action. The given expression is evaluated and the variable is assigned the resulting string, floating point number, integer number, or address (pointer). This action always succeeds.

Example: **ASSIGN $diff (- $x2 $x1);**

ATOMIC { *action1; action2; ... actionN;* }*;*
The sequence of actions *action1, action2, ... actionN* are all executed before control is given back to the interpreter. Normally, the interpreter performs several functions between each action (see Section 10). This might, in some cases, cause the system to interrupt execution of a sequence of actions in the middle of the sequence. Using the ATOMIC action bypasses this interpreter activity, guaranteeing that the sequence of actions will complete. This action succeeds when the entire sequence executes successfully, and fails if the sequence fails.

Example:

```
// A typical situation to use an ATOMIC construct - where early
// actions in the sequence might have otherwise caused the
// interpreter to prematurely interrupt execution of the sequence.
ATOMIC
{
    UPDATE (debug_mode) (debug_mode "False");
    UPDATE (run_mode) (run_mode "True");
    EXECUTE print "Now entering run mode.\n";
};
```

DO { *action1; action2; ... actionN;* } **WHILE** *: action0;*

The sequence of actions *action1, action2, ... actionN* are executed and then, if the *action0* (typically a TEST action) succeeds, the sequence of actions are repeated. This continues until *action0* fails, and the action after the DO is then executed, or until one of *action1, action2,... actionN* fails, whereby the entire DO action fails.

Example:

```
DO
{
    EXECUTE print $x;
    ASSIGN $x (+ $x 1);
} WHILE : TEST (> $x 10);
```

EXECUTE *function_name parameter1 parameter2 ... parameterN;*
Primitive functions may be executed by using an EXECUTE action. Primitive functions implement the low–level functionality of a complete system. Certain primitive functions, such as PRINT are already implemented. The primitives that come with UM-PRS are listed in Section 8.3. Parameters can be passed as arguments and can also be used to return values. See Section 8 for documentation on how to write new primitive functions.

Example: **EXECUTE process_image $img_ptr $num_objs $objlist_ptr;**

FACT *relation_name argument1 argument2 ... argumentN;*
Information from the world model can be accessed using the FACT action. The world model is searched for the first entry with the given relation name that has matching constant and bound variable arguments. Any arguments that are unbound variables are assigned the appropriate values from the matched world model entry. If no match is found, the action fails.

Example: **FACT robot_location $x $y $orient;**

FAIL *;*
This action *always* fails. It can be used to explicitly cause a branch or a KA to fail without having to resort to some more obscure method.

Example: **FAIL;**

LOAD *file_list;*
This action provides the functionality of bringing in additional KAs, goals, world model facts, etc. from a file in the middle of execution.where `file_list` is a list of filenames to load and parse.

Example: **LOAD "file1.kas" "file2.kas" "file3.kas";**

MAINTAIN *goal_name parameter1 parameter2 ... parameterN;*
A *maintain* goal instructs the system that the given goal must be reattained if it ever becomes unsatisfied. Parameters can return values. If there are no applicable KAs for goal *goal_name*, then the action fails. MAINTAIN goals are not currently implemented.

Example: **MAINTAIN objects_avoided;**

OR { *active-sequence1* } ... { *action-sequenceN* };
where *action-sequence* (also called a *branch*) is one or more of any of the actions listed here, and $N > 0$. The semantics of an OR action is that the interpreter will execute each of the branches in turn, with the first branch listed executed first. If any of the branches succeed, then the action succeeds. If all branches fail, then the entire action fails.

Example:

```
OR
{
    TEST (< $z 0);
    EXECUTE print "Error: value < 0!\n";
}
{
    TEST (>= $z 0);
    ASSIGN $altitude $z;
};
```

POST *goal_expr;*
This has the effect of adding a so-called top-level goal (i.e. a goal that has no parent goal) to the system's goal list. The goal expression `goal_expr` is a standard ACHIEVE, QUERY, MAINTAIN, or WAIT KA action consisting of the goal type, goal name, goal arguments, and the optional priority.

Example: **POST ACHIEVE robot_homed $x $y $orient :PRIORITY (+ $x $y 10);**

QUERY *goal_name parameter1 parameter2 ... parameterN;*
A QUERY action is functionally identical to an ACHIEVE action. It is provided to allow the programmer to be more explicit about the semantics of the action's goal (information acquisition, in particular). If the invoked KA fails, the QUERY action fails.

Example: **QUERY vision_processed $object_list;**

RETRACT *relation_name argument1 argument2 ... argumentN;*
One or more entries in the world model can be removed from the database using a RETRACT action. The system searches the world model for all entries with the given name, *and the given parameter values*, and removes them if any exist. If the arguments do not match, then no action is taken. This action never fails.

Example: **RETRACT goal_in_view "True";**

RETRIEVE *relation_name argument1 argument2 ... argumentN;*

Information from the world model can also be accessed using the RETRIEVE action. But RETRIEVE action gets new values from the world model for the variable arguments regardless of the current value of the variables. Like FACT, this action searches for the first world model entry, in this case matching only against the relation name. This action should be used carefully because, if the action is executed and it fails to get new values, the variable arguments will *lose* their values. This action succeeds if there is an entry with the given relation name in the world model, and fails otherwise.

Example: **RETRIEVE robot_location $x $y $orient;**

TEST *expression;*
The action succeeds if the expression evaluates to true, otherwise it fails. Note that an expression is true if it has either a non-zero value or is a non-null string.

Example: **TEST (== $x $y);**

UNPOST *goal_expr;*
The UNPOST action looks through the system's goal list for the goal specified in `goal_expr`. If it finds the goal, and the goal is not currently being pursued, it simply removes the goal from the goal list. However, if a KA for the goal *is* being executed, the goal is not immediately removed. The goal will be removed when the system's interpreter tries to execute the goal's next KA action resulting in it and all of its subgoals being removed from the system's intention structure.

The goal expression `goal_expr` is a standard ACHIEVE, QUERY, MAINTAIN, or WAIT KA action. The goal expression may only partially specify all of the arguments for goals on the system's goal

list, indicating that any goal(s) matching the partial specification should be removed. For example, if only the goal name is given in the UNPOST action, any goals in the system's goal list with that name will be removed. Similarly, if only the goal name and a priority are specified, any goals with identical name and priority, regardless of any arguments, are removed. Concrete examples are shown below.

Example: **UNPOST ACHIEVE robot_homed;** // Will remove all goals with name `robot_homed`

Example: **UNPOST ACHIEVE robot_homed $x $y;** // Will remove all goals with name `robot_homed` and with the exact same values for its first two arguments as the values of the variables.

Example: **UNPOST ACHIEVE robot_homed $x $y $orient :PRIORITY 10;** // Will remove all goals with name `robot_homed` and with the exact same values for its first two arguments as the values of the variables, and with the exact same priority.

UPDATE *(relation_name arg1 arg2 ... argN) (relation_name arg1 arg2 ... argM);*

The system searches the world model for all entries with the same name *and the same parameter values* as given in the first relation and removes them if any exist. The second relation is then asserted in the world model. If no arguments are given in the first relation then all but one of the world model relations with the matching relation name are removed and the remaining relation is updated with the information given in the second relation. This action never fails.

Example: **UPDATE (goal_in_view) (goal_in_view "False");**
or **UPDATE (goal_in_view "True") (goal_in_view "False");**

WAIT *goal_name parameter1 parameter2 ... parameterN;*
WAIT actions cause the system to suspend execution of the KA until the specified goal is achieved. As with ACHIEVE goals, parameters can return values. This action cannot fail. WAIT goals are not currently implemented.

Example: **WAIT messages_processed $message_list;**

WHEN *: action0 { action1; action2; ... actionN; };*
The *action0* (typically a TEST action) is executed and, if it succeeds, the sequence of actions *action1, action2, ... actionN* are executed. Use of this action simplifies programming many conditional branch situations. If more complex conditional branches need to be programmed, use the OR construct.

Example:

```
WHEN : TEST (< $z 0)
{
    EXECUTE print "Error: value < 0!\n";
}
```

WHILE *: action0 { action1; action2; ... actionN; };*
The *action0* (typically a TEST action) is executed and, if it succeeds, the sequence of actions *action1, action2, ... actionN* are executed. When *actionN* has been sucessfully executed, *action0* is again checked. This continues until eventually *action0* fails, and the action after the WHILE is then executed, or until one of *action1, action2,... actionN* fails, whereby the entire WHILE action fails.

Example:

```
WHILE : TEST (> $x 10)
{
    EXECUTE print $x;
    ASSIGN $x (+ $x 1);
};
```

3.2.2 Comments

Comments can be added to any parsed file (those specifying top-level goals, World Model entries, KAs, etc.) through the use of "//" characters, which indicate that the remainder of the line is a comment, and the pair "/*" "*/", which is interpreted as making a comment out of whatever text is between them (can extend to multiple lines of text). This is identical to the commenting scheme of C++.

3.3 Name

The name of the KA is simply for ease of human perusal and is not used at all during execution of the system. It is merely a label for the Knowledge Area.

3.4 Documentation

The documentation slot may be used to store descriptive text of any type. Notes about special characteristics of the KA, a description of how the KA works, etc. can be placed here. It does not impact upon system execution in any way.

3.5 Purpose

The *purpose* of a KA specifies the goal that successful execution of the KA body will satisfy. The syntax of the purpose is the same as that for a KA body ACHIEVE, MAINTAIN, or WAIT goal. A purpose may also be a QUERY action.

During execution of PRS, this purpose will be matched against top–level goals (see Section 5), as specified by the user before the system is started, and against KA body actions that specify a goal or query. If the KA's purpose matches, and the context is satisfied (as explained below), it is considered an applicable approach to solving the goal and may be *intended* (see below, in Section 10) and executed.

3.6 Context

The context of a KA specifies exactly in which situations the KA may be useful. The context is checked when a KA is being considered for inclusion in a Set of Applicable Kas (the SOAK), a list of KAs that are applicable toward solving a goal. The context is also continually checked

throughout the execution of a KA once the KA starts execution, to make sure that the the KA is still applicable to the intended situation.

The context specification is described in terms of a list of expressions. Each expression is evaluated in turn, and its return value checked (a context that has no expression, a KA without a CONTEXT defined, always succeeds.) The context may contain world model actions (RETRIEVE and FACT), which can be used to check for particular world states. Predefined UM-PRS expressions (e.g. <, >= != , etc.) can be used in the context to check relationships of variables to constants and other variables. The user can also define primitive functions which can be used in expressions (see Section 8). If every expression returns "TRUE" (non-zero, a non-null pointer, a non-nil string, or `Value::True`), then the context is considered to have passed. If any of the expressions returns "FALSE" (zero, null string, a null pointer, or `Value::False`), then the entire context is considered to have failed.

It is important to understand when expressions fail, and when they succeed. For example, failure for a FACT predicate (for more information on predicates, see below) results from the inability of the system to find a correct match in the world model for the given world model relation and *instantiated* arguments. The FACT action uses the instantiated arguments during matching and does not overwrite those values with new values. Care must be taken that variables used in FACTs in the context are not changed in the KA body to values that would cause the context to fail.

An example of a valid context expression is shown below. It demonstrates the use of the UM-PRS action FACT, the relationship != , and the user-defined predicate (primitive function) `task_complete`, which might check to see if the given task has been completed.

```
CONTEXT: FACT task $task;
         (!= $task "Do Nothing");
  (task_complete $task);
```

During SOAK generation, if the context check is not satisfied, the KA is not considered applicable to the situation and is dropped from consideration for execution. Also, if the context of a currently executing KA fails, the entire KA, and any of its subgoals, are deemed invalid and removed from the intention structure.

As was mentioned earlier, the list of context entries is implicitly considered a conjunction (an AND), so that each of the three entries cannot fail. The context will fail its initial check if there is no world model entry with the name "task". If the value for the variable has already been established in the PURPOSE of the KA in which this context is found, the context will also fail if there are no world model entries with the name "task" with a first parameter that matches the value of the variable `$task`, even if there are world model entries that do share the same name of "task". On the other hand, if no value has yet been bound to the variable `$task`, then the world model is searched for matching world model relation names and a value is bound to the variable. The context will also fail if this value is not the string "Do Nothing", or if the user-defined function `task_complete` fails with particular variable binding of `$task`. Similar functionality could be achieved through the simplified context:

```
CONTEXT: FACT task_complete "False";
         FACT task "Do Nothing";
```

Notice that variable bindings can be established through the context. World model action such as FACT and RETREIVE can be used to get values from the world model that can then be used in expressions later in the context. Note that particular attention must be paid to the failure semantics of each of the entries in the context. In the first context example, the variable `$task`

will be established with a particular value during the initial context generation if it hasn't already been bound a value. If the KA starts then starts executing, the context will be checked with the current bindings of the variables. If the binding for `$task` *ever* changes, whether to the string "Do Nothing" or otherwise, the context will fail if a world model entry with a matching value does not exist. If the variable changes to the string "Do Nothing", the context will fail too, of course.

Primitive functions may also be used (as an Expression, e.g. "¿" and "¡=") within the CONTEXT of a KA. The primitive function will be executed normally. The return value of the primitive function is used to determine whether the context is satisfied. A return value of `Value::False`, `Value(0)`, `Value(0.0)`, or `Value("")` are interpreted as a failure of a primitive, so care must be taken when designing primitive functions that may be used in KA contexts so that they do not return these values if that is not the desired interpretation. The syntax of using a primitive function in this manner is the same as that for other Expressions, namely surrounded by parentheses.

An example of a KA CONTEXT incorporating a primitive function is shown below. This example demonstrates how a primitive function can bind a value to a variable that is used later in the context (it can be used within the KA body too, of course). This example context, then, is satisfied when the distance between two points is less than 5.

```
KA {
PURPOSE: ACHIEVE example_completed $x1 $y1 $x2 $y2;
CONTEXT:
        (calc_distance $x1 $y1 $x2 $y2 $distance);
        (< $distance 5);
BODY:
      EXECUTE print "Within 5 units.\n";
}
```

3.6.1 Predicates

Within UM-PRS, several actions have parallel semantics to their "normal" KA BODY semantics. This parallel occurs when the actions are used as predicates (i.e. used within an expression rather than as a KA action). Each predicate returns a boolean value (i.e. `Value::True` or `Value::False`) which can be used in expressions located within contexts or such actions as TEST or WHILE. The semantics of each predicate is given below.

ACHIEVE *goal_name parameter1 parameter2 ... parameterN :PRIORITY expression;*

This predicate checks for the presence of the the given goal in the system's goal list. It uses the current variable values within the goal's parameter list and checks for an exact match. If an exact match is found, the predicate returns true, otherwise it returns false.

Example: **TEST (ACHIEVE robot_homed $x $y $orient :PRIORITY (+ *x*y 10));**

FACT *relation_name argument1 argument2 ... argumentN;*
A FACT acts as a predicate once variables in the FACT action have values bound to them. This predicate tests to see if the given world model relation with the given arguments exists in the world model. It returns true if one is found, false otherwise.

Example: **TEST (FACT robot_location $x $y $orient);**

RETRIEVE *relation_name argument1 argument2 ... argumentN;*

The RETRIEVE action acts as a predicate when found in a context expression or a TEST action (also in such cases as the TEST component of a WHILE or DO action). This predicate tests to see if there are *any* world model entries with the given relation name and the correct number of arguments. If a world model entry exists, this predicate returns true, otherwise it returns false.

Note that a side-effect of this predicate is to bind new values to the variables if a world model match is found. Any variable bindings in the RETRIEVE predicate will be replaced.

Example: **TEST (RETRIEVE robot_location $x $y $orient);**

3.7 Priority

The priority field of a KA is optional and specifies the priority of the instantiated KA. The default KA priority is 0 when it is not specified. The priority can be any valid expression including an expression with variables as in the following example.

```
PRIORITY: (+ $a (- $b $c));
```

The details of how the KA priority and the goal priority are interpreted will be explained in Section 10.

3.8 Effect Section

The EFFECT section is identical in format to a KA BODY, but is used to represent a simplified and abstracted procedural representation of the real procedure. There are no limitations or constraints upon the content of the EFFECT section with respect to a KA BODY. Everything that can be done within a KA BODY can be done within the EFFECT section.

This section of a KA is executed when UM-PRS is run in simulation mode (see Section 9.2). World model updates and subgoaling are the typical KA actions contained in the EFFECT section. Loops and branches are constructs that we foresee as being abstracted out of the BODY when writing the EFFECT section. If the EFFECT section is empty, the interpreter considers the KA to have succeeded.

An example KA with an EFFECT section is shown below:

```
KA {
NAME:
        "Test Interpreter in simulation mode"
PURPOSE:
        ACHIEVE sim_done;
CONTEXT:
        FACT test_done "False";
BODY:
        EXECUTE print "\nNormal execution started.\n";

        EXECUTE print "Simple subgoaling.\n";
        ACHIEVE system_initialized;
        UPDATE (system_init) (system_init "True");
```

```
        EXECUTE print "More complex subgoaling.\n";
        ACHIEVE indirect_communicated "Execution started indirect.\n";

EFFECT:
        ACHIEVE system_initialized;
        UPDATE (system_init) (system_init "True");
        ACHIEVE indirect_communicated "Simulation started indirect.\n";
}
```

3.9 Failure Section

An optional body of a KA, called the FAILURE section, specifies a procedure to be executed when a Knowledge Area fails. If the KA fails, because of context failure or failure of the KA BODY, the interpreter executes the actions found in the FAILURE section before doing anything else (see Section 10 for more information on the interpreter and what it does). Execution of this section is performed as if the entire procedure were an ATOMIC action (i.e. without interruption or the normal interleaving of execution of actions with interpreter activity).

When a KA fails due to its CONTEXT failing, it and all of that KA's subgoals are deemed to have failed. In this case, the FAILURE sections of each of the failed KA's subgoals are executed, from the "leaf" KA (the KA for the goal at the bottom of the subgoal stack - i.e. that which does not have a subgoal itself) back up to the failed KA. The following KAs demonstrate the use of FAILURE sections and, when executed, will illustrate how the FAILURE sections are executed when a KA's CONTEXT fails.

```
GOALS:
ACHIEVE testing_done;

FACTS:
test_done "False";
force_failure 0;


//-----------------------------------
// KA
//-----------------------------------
KA {
NAME:
"Test Interpreter on subgoal failure"
DOCUMENTATION:
"Test Interpreter on subgoal failure"
PURPOSE:
ACHIEVE testing_done;
CONTEXT:
FACT test_done "False";
BODY:
     EXECUTE print "Just before top level ACHIEVE.\n";
OR
{
    ACHIEVE communicated "In Failure branch1.\n";
    ACHIEVE subgoal_failed1;
}
{
```

```
    ACHIEVE communicated "In branch2.\n";
};
}


//-----------------------------------
// KA
//-----------------------------------
KA {
NAME:
"Subgoal to failure1"
DOCUMENTATION:
"Intermediate level subgoal failure."
PURPOSE:
ACHIEVE subgoal_failed1;
CONTEXT:
FACT test_done "False";
FACT force_failure 0;
BODY:
     EXECUTE print "In subgoal_failed1\n";
ACHIEVE subgoal_failed2;
}


//-----------------------------------
// KA
//-----------------------------------
KA {
NAME:
"Subgoal to failure"
DOCUMENTATION:
"Make sure subgoal failure 'works'."
PURPOSE:
ACHIEVE subgoal_failed2;
CONTEXT:
FACT test_done "False";
BODY:
     EXECUTE print "In subgoal_failed2\n";
OR
{
    ACHIEVE communicated "In Failure branch1 #2.\n";
    ACHIEVE ka_failed;
}
{
    ACHIEVE communicated "In Failure branch2 #2.\n";
    ACHIEVE ka_failed;
};
FAILURE:
EXECUTE print "\nsubgoal_failed2 failed.\n\n";
}


//-----------------------------------
// KA
```

```
//-----------------------------------
KA {
NAME:
"Simply fail."
DOCUMENTATION:
"Make sure that KA fails"
PURPOSE:
ACHIEVE ka_failed;
CONTEXT:
FACT test_done "False";
BODY:
EXECUTE print "Just before failure.\n";
     UPDATE (force_failure) (force_failure 1);
EXECUTE print "Just after failure.  Should never get here.\n";
FAILURE:
EXECUTE print "\nka_failed failed.\n\n";
}


//-----------------------------------
// KA
//-----------------------------------
KA {
NAME:
"Communicate the text to the user"
DOCUMENTATION:
"Print and speak the text"
PURPOSE:
ACHIEVE communicated $TEXT;
CONTEXT:
FACT test_done "False";

BODY:
EXECUTE print $TEXT;
FAILURE:
EXECUTE print "\ncommunicated failed.\n\n";
}
```

4 Cyclic functionality

In addition to the primitive functions that can be called through execution of KAs, UM-PRS permits arbitrary processing to occur every cycle through the interpreter. This functionality, typically used to update (assert, retract) the world model external to the normal KA execution cycle, is provided in procedural body called CYCLE. The CYCLE is specified in a syntax identical to that of a KA body in one of the files parsed during initialization (or using the LOAD action). This procedure may contain any KA action or construct that a normal KA body can contain *except* for subgoaling. Subgoaling is not supported as execution of the CYCLE procedure is outside of the normal execution of the interpreter (SOAK generation, element selections, etc.). Subgoal actions are detected during execution, and generate a warning message. The actual subgoal action is simply skipped.

An example of a complete UM-PRS program which includes a CYCLE (which simply maintains and prints out the number of times the interpreter has cycled) is shown below:

```
GOALS:
```

```
        ACHIEVE cycle_tested;
FACTS:
        test_done       "False";
        system_init     "False";
        cycle_number    0;
CYCLE {
        RETRIEVE cycle_number $N ;
        EXECUTE print "\nCycle#" $N "\n";
        UPDATE (cycle_number $N) (cycle_number (+ $N 1));
}

KA {
NAME:
        "Test CYCLE"
PURPOSE:
        ACHIEVE cycle_tested;
CONTEXT:
        FACT test_done "False";
BODY:
        EXECUTE print "\nNormal execution started.\n";

        UPDATE (system_init) (system_init "True");

        WHILE : TEST (== 1 1)
        {
                EXECUTE noop;
        };
}
```

5 System Goals

The initial, top-level goals that UM-PRS is to satisfy are specified in a text format in a syntax identical to KA goal actions. System goals, however, may only be either ACHIEVE or MAINTAIN goals, and must contain only constant values (integer numbers, floating point numbers, or strings).[3]. The list of goals is preceded by the keyword **GOALS:** when provided to UM-PRS (see Section 9). A simple example of a system goal specification might look something like:

```
GOALS:
        ACHIEVE cone_demo :PRIORITY 10;
        MAINTAIN obstacles_avoided;
```

Top level goals are persistent goals. That is, they are pursued until they are satisfied. In contrast, goals within KA bodies are *not* persistent (see Section 3.2.1).

6 Intention Structure

The *intention structure* maintains information related to the runtime state of progress made toward the system's top-level or "system" goals. The system will typically have more than one of these top-level goals, and each of these goals may, during execution, invoke subgoals to achieve lower-level goals. With the current implementation, a goal that is being pursued may be interrupted by a higher priority goal and then later

[3]There are no variable bindings yet to establish values for any variables

resumed (if possible). To manage this, the intention structure keeps track of which top-level goals are being pursued, and which action in each KA is currently being executed.

When a goal gets suspended, due to a higher level goal becoming applicable, the current state of execution of the current goal is stored. When the suspended goal becomes the highest priority goal again, it is "reactivated". There is nothing unusual in the reactivation process. It is identical to normal execution. However, due to the possibility that the world model has been changed during the pursuit of other goals, the contexts of the resumed top-level goal and its subgoals may no longer be valid. If they are all still valid, execution resumes at the exact place where it was suspended. However, if the context of one of the KA's context fails, that KA, and any of its subgoals, is considered to have failed. This is the normal behavior for KAs whose context fails. See Section 3.6 for more details.

7 The main() Function

The UM-PRS system library does not define the main function, but it defines the `long um_prs(int argc, char* argv[])` function. Thus the user should define his/her own main function. Typically, it just needs to call `um-prs` function as shown below. Note, however, that the `argc` and `argv` arguments should be passed to the `um-prs` function to pass runtime options to be explained in Section 9.

```
#include "user.h"

int main(int argc, char* argv[])
{
  return um_prs(argc, argv);
}
```

UM-PRS also allows users to override the default priority evaluation function (see Section 10.1). Users can do this by assigning the new function to `program.priority_evaluation_function_hook` as in Figure 1. The user defined priority evaluation function should have two `double` arguments, `goal_priority` and `ka_priority` and return a `double` priority value. An example definition of this function is also given in Figure 1.

8 Primitive Function Interfacing

Each primitive action referenced in a KA body must have a corresponding C++ or C language function that can be directly executed. An interface function that deals with converting UM-PRS variables to/from C/C++ representation must be specified for each primitive function.

The UM-PRS system is composed of a C++ library which can be linked directly to the user-defined interface functions and primitive executable functions. In order to make it easy for the user to define interface functions, we provide users with a set of constructs defined in the "user.h" header file. Thus, "user.h" should be included in every interface function file. The details of how to make a UM-PRS application are demonstrated in the "example" directory. Users are advised to copy the "Makefile" and the example C++ file and to modify it for their applications.

8.1 Interface Functions

All the interface functions have the same parameters as defined as follows:

```
#define PRIMITIVE_FUNCTION(name) \
        Value name(long arity, ExpList* args, Binding* binding=0)
```

```
#include "user.h"

double user_priority_evaluation(double goal_priority, double ka_priority)
{
  return goal_priority * ka_priority;
}

int main(int argc, char* argv[])
{
  // Point to the user priority evaluation function
  // Uncomment if such functionality is desired.
  // program.priority_evaluation_function_hook = user_priority_evaluation;

  return um_prs(argc, argv);
}
```

Figure 1: User priority evaluation function

The primitive function interface arguments are: *arity*, which specifies the number of arguments to the primitive function; *args*, which are the arguments of the primitive, and *binding*, which holds the variable bindings associated with the passed arguments. A simple print interface function is shown in Figure 2.

To extract the arguments from *args*, the user just needs to define an iterator, say `arg_list` (the expression list iterator class `ExpListIterator` is already defined by UM-PRS), and call the member function `get_next()` sequentially until it returns 0 (indicating that there are no more arguments). The value referred by the expression can be extracted by calling the `eval` function with `binding`, as in `exp->eval(binding)`.

The user also can set a value to the variable arguments using `binding->set_value` function as in the code sample shown in Figure 3. Returning values *and* passing arguments can, of course, be performed in the same primitive function.

Figure 4 is an example of a primitive function that receives a two structures as arguments and returns the result of a calculation based upon the contents of these two structures. Complex data structures must be passed as pointers to `void`, and then cast to the correct form before using them. This is shown in this figure. Although not shown, binding structures back to variables is also done using `void` pointers within the `binding->set_value` function call.

Each primitive function must return a `Value`. Since conversion functions from integers, floating-point numbers, strings, pointers to `Value` are already defined, the user can return any of these types of values. If the return is of type `Value::False`, the primitive function will be considered to have failed by the UM-PRS interpreter.

8.2 Function List

Finally, the user needs to define a list of primitive UM-PRS function name and the corresponding C++ interface function as follows:

```
FunctionNamePair user_function_list[] = {
{ka_print_args,              "print_args"              },
{ka_get_dead_reckoning,      "get_dead_reckoning"      },
{ka_calc_euclidean_distance, "calc_euclidean_distance" },
{0,                          0                         }
};
```

```
PRIMITIVE_FUNCTION(ka_print_args)
{
  if (arity < 0) return Value::False;

  ExpListIterator arg_list(args);
  Expression* exp;

  // Go through each argument and print it individually
  for(exp = arg_list.get_next(); exp; exp = arg_list.get_next())
    cerr << exp->eval(binding);

  // This primitive function has successfully completed.
  return Value::True;
}
```

Figure 2: Primitive function that gets passed arguments.

```
PRIMITIVE_FUNCTION(ka_get_dead_reckoning)
{
  int status, x, y;

  // If there aren't enough arguments then this
  // primitive function fails.
  if (arity < 2) return Value::False;

  ExpListIterator arg_list(args);

  // This parameter will be used for return values.
  Expression* robotx = arg_list.get_next();
  Expression* roboty = arg_list.get_next();

  // determines robot's position
  get_dead_reckoning(&x, &y);

  // Bind the newly determined values to return arguments
  binding->set_value(robotx,Value(x));
  binding->set_value(roboty,Value(y));

  // This primitive function has successfully completed.
  return Value::True;
}
```

Figure 3: Primitive function that returns values.

```
PRIMITIVE_FUNCTION(ka_calc_euclidean_distance)
{
  double dist;
  double euclidean_distance(Point* p1, Point* p2);

  // If there aren't enough arguments then this
  // primitive function fails.
  if (arity < 3) return Value::False;

  ExpListIterator next_exp(args);

  // Get the parameters for the primitive function.
  // These can be int, float, char *, void *
  void* p1 = (void *) next_exp()->eval(binding);
  void* p2 = (void *) next_exp()->eval(binding);

  // This parameter will be used for return values.
  Expression* distance = next_exp();

  // call the actual function that performs the calculation
  dist = euclidean_distance((Point*) p1, (Point*) p2);

  // Bind the result to the return (3rd) argument
  binding->set_value(distance,Value((float) dist));

  // This primitive function has successfully completed.
  return Value::True;
}
```

Figure 4: Primitive function that gets and returns values.

where the first entry is the name of the interface function and the second entry is the name used within a Knowledge Area.

8.3 Predefined functions

Currently there are several predefined primitives. They are defined below:

print *expression1 ... expressionN;*
This primitive function permits output to cout (typically the screen). The expressions can be any valid expression, including constant values (e.g. numbers, strings), variables, calculations, etc. Most formatting characters, such as "\n", "\t", etc. (as in C/C++ languages) is permitted.

Example: **EXECUTE print "\nHello " 12 " " $some_variable "\n.";**

sleep *sleeptime;*
This primitive pauses for the number of seconds indicated by the integer argument *sleeptime*.

Example: **EXECUTE sleep 5;**

noop *;*
This primitive function performs no action at all, simply returning successfully.

Example: **EXECUTE noop;**

read_integer *var_list;*
This primitive function accepts the input of one or more integer numbers from the user.

Example: **EXECUTE read_integer $var1 $var2 $var3;**

read_float *var_list;*
This primitive function accepts the input of one or more floating point numbers from the user.

Example: **EXECUTE read_float $var1 $var2 $var3;**

read_string *var_list;*
This primitive function accepts the input of one or more strings from the user. The current maximum length of each string is fixed, and is defined in the file "primitive.h" as READ_STRING_INPUT_SIZE.

Example: **EXECUTE read_string $var1 $var2 $var3;**

remove_ka *string_list;*
This primitive function will cause all of the KAs with a name that matches one of the given strings to be considered invalid, and therefore no longer applicable to be selected for execution. The example given below shows how to remove KAs with the name "Test CYCLE" (defined in Section 4).

Example: **EXECUTE remove_ka "Test CYCLE";**

abs *num_arg;*
This primitive function returns the absolute value of the numeric argument passed to it. If a non-numeric argument is passed to this function, ... If more than one argument is passed to this function, ...

Example: **ASSIGN $absvalue (abs $value);**

get_time *$time;*
This primitive function returns the value returned from the Unix time() function.

Example: **EXECUTE get_time $time;**

generate_soak *$soak;*
This primitive function generates a set of applicable KAs (a SOAK) based on the current world model, system goals, and KA library. It returns it in the first argument to the primitive function. This function also returns the size (the number of elements) of the SOAK so that it may be ASSIGNed to another variable. Note that this function allocates memory for the generated SOAK. It is the responsibility of the user to free that memory when the SOAK is no longer needed (see the primitive function below).

Example: **ASSIGN $num_elements (generate_soak $soak);**

delete_soak *$soak;*
This primitive function frees the memory associated with a SOAK generated by the `generate_soak` primitive function.

Example: **EXECUTE delete_soak $soak;**

print_soak *$soak;*
This primitive displays the information within the SOAK generated by a call to the `generate_soak` primitive function.

Example: **EXECUTE print_soak $soak;**

9 Building and Running a UM-PRS Based Application

Before UM-PRS is executed, the UM-PRS library must be built and linked with the compiled primitive action functions and primitive action interface functions as described above in section 8. Then, when UM-PRS is invoked, it needs to know where to look for the KA library, the world model, and the system's goals. The file containing this information is specified as a command line argument. UM-PRS parses the text from the file argument and creates internal storage representations for all of the information before proceeding with execution. If there are any parsing problems, UM-PRS will indicate this with appropriate error messages. UM-PRS will then begin to execute KAs until either there is an error (such as an error in a primitive action) and/or all of the system's goals have been achieved or have failed.

To create a complete UM-PRS system, first compile UM-PRS and build the UM-PRS library, compile user-created files that define the domain specific primitive actions and primitive action interface for the system, and link them together. The given `makefile` can be used without modification for UM-PRS library, but the `makefile` for the application should be modified appropriately.

UM-PRS recognize following runtime options for debugging purposes.

`-i` : show changes to the intention structure.

`-g` : show changes to the system's goal list.

`-p` : specifies the limit number of parsing errors before the UM-PRS gives up. The default value is 10.

`-s` : show SOAK generation and selection information.

`-w` : show the world model whenever a change occurs to the world model.

`-y` : show the process of parsing in detail. This is only useful to debug parsing errors.

`-S` : run UM-PRS in simulation mode.

9.1 Parser Messages

UM-PRS's first activity is parsing the initial definitions of the World Model, system Goal List, CYCLE procedure, and KA Library. If there are any problems found while parsing, UM-PRS will output a line of text of the form:

```
<filename>:error:<line#> (<character#>):parse error
```

where `<filename>` is the filename where the error was found, `<line#>` is the line number in that file, and `<character#>` is the character number in that file.

9.2 Execution Modes

There are currently two modes of execution for UM-PRS. The standard mode of operation is for the KA BODY procedure to be executed. This is the default. Another mode of operation executes the EFFECT section of a KA. This section of a KA was introduced to facilitate plan prespecification, simulated forecasting, or similar functionality. UM-PRS is placed in simulation mode by the using the run-time switch "-S" (see above). See Section 3.8 for complete details on the EFFECT section of a KA.

10 UM-PRS Interpreter

The UM-PRS interpreter is responsible for selecting and executing Knowledge Areas based upon the goals and context of the current situation. Associated with the interpreter is the *Intention Structure*, a run-time stack (stacked based upon subgoaling) of instantiated KAs, and the *Goal List*, the set of the system's current top-level goals. The interpreter operates using the basic cycle shown below in pseudocode:

```
while (1) {

  execute_cycle();

  generate_SOAK(soak);

  // If a new SOAK, or if last SOAK was non-nil, add it to the
  // SOAK list
  if (soak || previous_soak) soak_list(add,level++);

  // Continue to top of loop an see if a meta-level SOAK is generated.
  if (soak) { previous_soak = soak; continue; }

  // If no new KAs were relevant
  if (!previous_soak) {

    if (all_goals_achieved) exit;

    else {
      // Execute something (in the leaf KA) in the intention structure.
      if (intention_structure.execute() == FAILED) {
        intention_structure.unintend();
        level = 0;
        soak = 0;
      }
    }
  }
```

```
    else {

      // Randomly select one of the new KAs of highest priority
      // and add it to the intention structure
      intention_structure.intend(soak.highest_priority_random());

      // Execute something (in the leaf KA) in the intention structure.
      if (intention_structure.execute() == FAILED) {
        intention_structure.unintend();
        level = 0;
        soak = 0;
      }
    }
  }
```

Note that the `get_new_facts_goals_messages()` function is user definable. See Section 7 for how to define this. Also note that The system currently chooses randomly between applicable KAs that share the highest priority.

10.1 Priority-based Meta–level Reasoning

In our current implementation, the interpreter selects one KA based on the priority of the goal and the KA when there are multiple elements in the SOAK. When a goal is posted the optional goal priority (Section 3.2.1) is evaluated and associated with the goal. If no explicit goal priority is specified, its default value becomes 0.

The priority of the instantiated KA in SOAK is the sum of the goal priority (Section 3.2.1) and the KA priority (section 3.7) in default. If users want to override the default priority evaluation function, please refer to Section 7. Note that since the KA priority of the KA is calculated at runtime, the same KA with different bindings can have different priorities.

The interpreter selects the KA with highest priority and, if there are multiple KAs with the highest priority, it selects one among them randomly.

10.2 Reactivity

UM-PRS is very reactive to changes in the environment. When a new fact about the world is recorded in the world model through sensing, communication, or internal conclusions, the UM-PRS interpreter will look for KAs that are applicable to the new situation. Likewise, if a new goal (or goals) is added, UM-PRS looks for applicable KAs to satisfy the new goal(s). In either case, UM-PRS selects one of the applicable KAs, weighs its importance against that which it is actively working on, and if found to be of more importance, intends it (places it on the intention structure), and then executes something in the active "intention".

11 Changes/Modifications from UM-PRS V3.0

In this section we discuss those changes that we have made from the previous version. These are listed below:

- Run-time semantics during multi-threaded execution were changed a little so that Intention Stacks that were active, but "blocked" waiting on a subgoal in which there are no applicable KAs, will permit other, runnable stacks to execute.
- New primitives `abs`, `get_time` implemented, which return the absolute value and the Unix time() function, respectively.

- New primitives `generate_soak`, `delete_soak`, and `print_soak` implemented, which generates a new SOAK, deletes such a generated SOAK, and prints out such a generated SOAK.
- Fixed the remaining memory leaks. There is now zero memory leaked during execution.
- The RETRACT KA action was changed so that it actually causes one or more World Model entries to be removed from the World Model rather than simply flagged as being invalid. This saves a great deal of memory over long or complex runs.
- New primitive `shell`, which passes its string argument directly to the external shell for processing.
- KAs no longer need to have entries in their CONTEXT slot.
- Much improved performance during SOAK generation due to the use of hash tables for searching and matching.

12 Known bugs

In this section we describe any known shortcomings of the current version of the implementation. At this point in the development, there are no known problems with UM-PRS code. However, please contact the authors if any problems are encountered and it appears to be a UM-PRS software bug.

References

[1] Michael Georgeff and Amy L. Lansky. Reactive reasoning and planning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 677–682, Seattle, Washington, August 1987.

[2] Michael P. Georgeff and Amy L. Lansky. Procedural knowledge. *Proceedings of the IEEE Special Issue on Knowledge Representation*, 74(10):1383–1398, October 1986.

[3] Francois Ingrand, Michael Georgeff, and Anand Rao. An architecture for real-time reasoning and system control. *IEEE Expert*, 7(6):34–44, December 1992.

[4] Francois. F. Ingrand and Michael P. Georgeff. Managing deliberation and reasoning in real-time AI systems. In *Proceedings of the 1990 DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control*, pages 284–291, Santa Diego, CA, November 1990.

[5] Jaeho Lee, Marcus J. Huber, Edmund H. Durfee, and Patrick G. Kenny. UM-PRS: an implementation of the procedural reasoning system for multirobot applications. In *Conference on Intelligent Robotics in Field, Factory, Service, and Space (CIRFFSS '94)*, pages 842–849, Houston, Texas, March 1994.

[6] SRI International, Menlo Park, California. *Procedure Reasoning System User Guide*, March 1992.