PRODIGY4.0: The Manual and Tutorial [1]

The PRODIGY Research Group,
under the supervision of Jaime G. Carbonell:
Jim Blythe, Oren Etzioni, Yolanda Gil, Robert Joseph, Dan Kahn,
Craig Knoblock, Steven Minton, Alicia Pérez, Scott Reilly,
Manuela Veloso, and Xuemei Wang. [2]

June 1992
CMU-CS-92-150

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

PRODIGY is a general-purpose problem-solving architecture that serves as a basis for research in planning, machine learning, apprentice-type knowledge-refinement interfaces, and expert systems. This document is a manual for the latest version of the PRODIGY system, PRODIGY4.0, and includes descriptions of the PRODIGY representation language, control structure, user interface, abstraction module, and other features. The tutorial style is meant to provide the reader with the ability to run PRODIGY and make use of all the basic features, as well as gradually learning the more esoteric aspects of PRODIGY4.0.

[1]This research was sponsored by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

[2]The initial design was fully developed and implemented by Jim Blythe, Xuemei Wang, and Dan Kahn. The authors acknowledge Steven Minton, Jaime Carbonell, Craig Knoblock, and Dan Kuokka in designing and implementing the initial PRODIGY system, PRODIGY2.0. Manuela Veloso and Daniel Borrajo designed and implemented the first nonlinear problem solver for PRODIGY, NOLIMIT. Our past experience with PRODIGY2.0 and NOLIMIT drove our design and implementation of PRODIGY4.0. Most of this manual was written by Jim Blythe, Yolanda Gil, Robert Joseph, Alicia Pérez (editor), and Xuemei Wang.

Keywords: planning, problem solving, machine learning, Prodigy

1 Introduction

PRODIGY is a domain-independent problem-solving architecture used primarily as a testbed for research in planning, machine learning, and knowledge acquisition. Although PRODIGY could be employed as a theorem prover or an expert system, it is primarily intended to be used as a general planner. In particular, the PRODIGY architecture is well suited for:

1. A basic means-ends analysis nonlinear planner in high-level, symbolic domains.
2. A tool for exploring the space of planners in complex and unusual domains (via domain-oriented control rules and other adaptations).
3. A testbed for machine learning research, primarily speed-up learning (EBL, analogy, abstraction, etc.), but also learning at the knowledge-level (experimentation), and interactive knowledge acquisition (see Appendix A for an overview of these mechanisms).

Points two and three are important: PRODIGY provides a platform for research in planning and learning that eliminates the need for each researcher to start from scratch and design her or his own planning system. Although PRODIGY does not address certain issues (such as representing uncertainty), it can be used as an initial model, to aid in the design or serve as the basis for a more complex planner. PRODIGY is a powerful and well-engineered system. One of our main concerns is to make PRODIGY a *usable* system.

PRODIGY has been applied to many different domains: robotic path planning, the blocksworld, an augmented version of the STRIPS domain, multirobot task planning, matrix algebra multiplication, discrete machine-shop planning and scheduling, process planning, computer configuration, logistics planning, and several others.

1.1 PRODIGY4.0

The current version of the PRODIGY problem solver incorporates several features with respect to previous versions. PRODIGY4.0 is a nonlinear problem solver allowing full interleaving of goals. It works with a *set* of goals, as opposed to the top goal in a goal stack, as PRODIGY2.0 does. Improving the matcher efficiency has been also a major effort of the project. The multi-level abstraction planning capability is fully integrated with the problem solver. In addition, PRODIGY4.0 provides interruption handling, implements a simple form of dependency-directed backtracking, and allows multiple problem spaces.

1.2 How to Use This Manual

Section 2 contains a very simple example of how PRODIGY solves a problem. This section is a good introduction to how the system works. Its subsections introduce the rest of the manual. Section 3 shows how to represent domain knowledge, including how to define operators and inference rules. Section 4 describes PRODIGY's control structure in detail, showing exactly how the search proceeds. Section 5 explains how to use control rules to guide the search, and how to write them.

To run the example from Section 2, skip the sections mentioned above and go directly to Section 6. Sections 6.2 and 6.5 describe how to load and run the system, and how to use the trace produced by it. Section 7 gives some hints on how to debug a domain.

Section 8 contains some additional features of PRODIGY4.0. Section 9 gives an overview of the abstraction module, that allows PRODIGY to do hierarchical planning in new domains. Section 10 provides information about how the system files are organized, and how to load a domain.

Section 11 provides answers to frequently asked questions. Appendix A provides an overview of PRODIGY's learning components. Appendix B summarizes the syntax of PRODIGY4.0's description Language (PDL4.0). Finally, Appendix C contains a sample domain, the extended-STRIPS world, from which some of the manual examples have been taken.

2 PRODIGY in a Nutshell

Simply stated, PRODIGY searches for a sequence of actions that transform an initial state (the current state) into a final state (the goal state). To illustrate how PRODIGY works, in this section we will step through the process of PRODIGY solving a simple problem. The problem is taken from the familiar blocksworld domain described in many introductory AI textbooks (such as [Nilsson, 1980, Rich and Knight, 1991]). The domain consists of an idealized robot arm, a table, and blocks, that can be manipulated by the robot arm. First, we will define the blocksworld domain and the specific problem to be solved. Then we will look at a brief description of the trace produced by the problem-solving episode in order to introduce PRODIGY's control structure.

2.1 Defining a Domain

The domain theory (or "domain" for short) specifies the legal actions that can be performed in terms of *operators*. Although PRODIGY is used for planning in many domains, some quite complex, we use here a pedagogically simple domain, the blocksworld. In this domain (see Figure 1), there are only four operators: PICK-UP, PUT-DOWN, STACK, and UNSTACK. PICK-UP and PUT-DOWN are used for picking blocks up from the table and putting them down on the table. UNSTACK and STACK are analogous, but are used for picking up and putting down blocks on other blocks. In addition to operators, the domain theory specifies the legal inferences that can be made in terms of *inference rules*. In our domain, there is a single inference rule for inferring that the arm is empty (represented by the predicate `(arm-empty)`) when the robot arm is not holding any block, and for retracting `(arm-empty)` when it has a block in hand. In this particular world, a robot can only PICK-UP or UNSTACK a block if it is not already holding another block (i.e. if the predicate `(arm-empty)` is true).

Each operator and inference rule has a *precondition expression* that must be satisfied before the operator can be applied and an *effects list* that describes how the application of the operator changes the world. Variables that appear in both parts can have *types*, constraining the objects that can match them.

For example, the operator STACK in Figure 1 specifies that a block can be stacked if the robot is holding it and the block on which it is to be stacked is clear (i.e., has nothing on top of it). The type declarations in the preconditions specify that both blocks must be instances of the type OBJECT and that they must be different. The effects-list indicates that after the operator is applied the robot is no longer holding the block and the block underneath is no longer clear. In addition, the block that was just put down is now clear and it is on the bottom block. The syntax used for operators and inference rules is covered in more detail in Section 3.

Notice how an operator specifies many effects, and the parameters enable it to apply in a range of situations. For example, STACK can be used to stack any two blocks. When PRODIGY uses operators for planning, though, it specifies values for the parameters, so it can reason about the new state created by applying it. An operator with all its parameters specified, as in `<stack blockA blockB>` which stacks a particular block on another, is called an *instantiated operator*, and is usually written in angle brackets.

```
(OPERATOR PICK-UP
 (params <ob1>)
 (preconds
  ((<ob1> OBJECT))
  (and (clear <ob1>)
       (on-table <ob1>)
       (arm-empty)))
 (effects
  ()
  ((del (on-table <ob1>))
   (del (clear <ob1>))
   (add (holding <ob1>)))))
```

```
(OPERATOR PUT-DOWN
 (params <ob>)
 (preconds
  ((<ob> OBJECT))
  (holding <ob>))
 (effects
  ()
  ((del (holding <ob>))
   (add (clear <ob>))
   (add (on-table <ob>)))))
```

```
(OPERATOR UNSTACK
 (params <ob> <underob>)
 (preconds
  ((<ob> Object)
   (<underob> Object))
  (and (on <ob> <underob>)
       (clear <ob>)
       (arm-empty)))
 (effects
  ()
  ((del (on <ob> <underob>))
   (del (clear <ob>))
   (add (holding <ob>))
   (add (clear <underob>)))))
```

```
(OPERATOR STACK
 (params <ob> <underob>)
 (preconds
  ((<ob> Object)
   (<underob>
    (and OBJECT (diff <ob> <underob>))))
  (and (clear <underob>)
       (holding <ob>)))
 (effects
  ()
  ((del (holding <ob>))
   (del (clear <underob>))
   (add (clear <ob> ))
   (add (on <ob> <underob>)))))
```

```
(INFERENCE-RULE INFER-ARMEMPTY
 (params)
 (preconds
  ()
  (~ (exists ((<ob> OBJECT))
     (holding <ob>))))
 (effects
  ()
  ((add (arm-empty)))))
```

Figure 1: Blocksworld Domain Theory.

```
(setf (current-problem)
      (create-problem
        (name sussman)
        (objects (blockA blockB blockC object))
        (state (and (on-table blockA)
                    (on-table blockB)
                    (on blockC blockA)
                    (clear blockB)
                    (clear blockC)
                    (arm-empty)))
        (goal (and (on blockA blockB)
                   (on blockB blockC)))))
```

Figure 2: Example Blocksworld Problem.

2.2 Specification of a Problem

Once the domain has been specified, *problems* are presented to PRODIGY by describing an initial state and a goal expression to be satisfied. In the example in Figure 2, often known as Sussman's anomaly, the goal is to build a tower of the three blocks blockA, blockB and blockC, with blockA on top and blockC on the bottom. In the initial state blockA and blockB are on the table, and blockC is on top of blockB. Since the syntax for the goal statement is a logical expression, multiple goals can be expressed as a conjunction.

2.3 The Goal Tree

PRODIGY solves a problem by searching for a sequence of operators that transform the initial state into a final state that satisfies the goal expression. It does this by means-ends analysis, which means that it will pick a goal that is not yet true and attempt to find an operator that would make it true. The preconditions of this operator then also become goals, and the process is repeated, until all the goals either have operators attached or are true, yielding a plan.

One structure that helps to follow PRODIGY's problem-solving behaviour is the *goal tree*, which is a tree whose nodes are the goals and operators considered by PRODIGY during problem solving. In this tree, a goal is linked to the operators that PRODIGY considers using to achieve the goal, and an operator is in turn linked to the subgoals corresponding to the operator's preconditions. Figure 3 shows the goal tree that PRODIGY generates in solving our example, Sussman's Anomaly. The two top-level goals appear under the root node at the top of the figure, and must both be solved for the problem to be solved. The operator node for `<stack blocka blockb>` appears as a child of the left goal node, `(on blocka blockb)`, because PRODIGY uses that operator to achieve the goal. Its preconditions are satisfied, in turn, by achieving the two goals `(clear blockb)` and `(holding blocka)` simultaneously.

Notice that the goal node `(clear blockb)` has been greyed out. This is because the goal is already true when PRODIGY tries to achieve it — in fact, for this reason PRODIGY does not bother to create the node. Some of the greyed nodes are goal nodes, meaning that they are true when they need to be achieved, and some are operator nodes, meaning that all the preconditions of the operator are true when they need to be achieved. These nodes thus mark a successful termination of the goal tree.

In general, in order to tell which goals will be true when PRODIGY tries to achieve them, it is necessary to know what other operators PRODIGY has already applied. This information cannot be represented in the goal tree, so we need to consider the problem solving trace to see the whole picture.

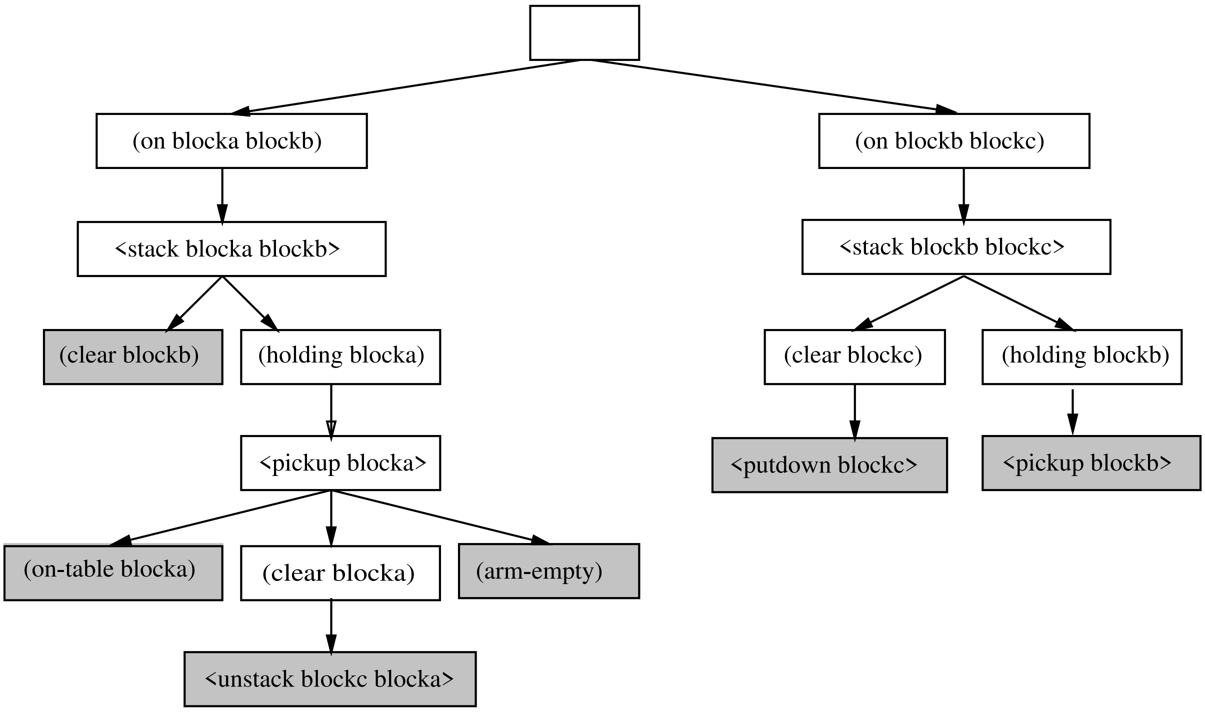


Figure 3: The Goal Tree Generated by PRODIGY4.0 for Solving Sussman's Anomaly.

2.4 The Problem-Solving Trace

A trace of PRODIGY solving Sussman's anomaly is shown in Figure 4. Each decision that PRODIGY makes during problem-solving is shown by a line in the trace, with the exception of decisions about uninstantiated operators. The decisions together form a structure called the *search tree*, which PRODIGY uses to backtrack efficiently. Each line in the trace corresponds to a node in the search tree . The line begins with the depth of the node in the search tree, followed by a label for the node, and a short description.

As problem solving develops, the system expands the goal tree from Figure 3, and so each goal node or instantiated operator node in the search tree can be mapped to a node in the goal tree. The indentation of a line in the trace corresponds to the depth of the equivalent node in the goal tree. In addition to the information in the goal tree, PRODIGY makes decisions about the order in which to apply the operators. These decisions are lines in the trace showing instantiated operators printed in upper case (e.g line corresponding to `n16`), and these lines represent *applied operator* nodes in the search tree.

We now give a detailed description of the trace in Figure 4. The user initiates problem-solving by typing `(run)` at the Lisp prompt. First, the objects that are specific to the problem to be solved are created, in this case, the three blocks. Then PRODIGY creates a special operator, called `*finish*`, whose preconditions are the top-level goals and whose sole effect is to add the goal `(done)`. In addition, PRODIGY applies all the inference rules that can be applied to the initial state.

At node `n5`, PRODIGY begins working on the goal `(on blocka blockb)`, which is the left of the two top level goals in the goal tree shown in Figure 3. The only action available to achieve this goal is stacking blocka on blockb. This then gives rise to a new subgoal, holding blocka, which is a precondition of the STACK operator. PRODIGY then works through the left side of the goal tree, coming to the operator <unstack blockc blocka> at node `n15`. Note that neither PUT-DOWN nor STACK would work, since they lead to a goal loop. Take PUT-DOWN for example. One way to make blocka clear is to put it down.

```
* (run :output-level 3)
Creating objects (BLOCKA BLOCKB BLOCKC) of type OBJECT

  2 n2 (done)
  4 n4 <*finish*>
  5   n5 (on blocka blockb) [1]
  7   n7 <stack blocka blockb>
  8     n8 (holding blocka) [1]
 10     n10 <pick-up blocka>
 11       n11 (clear blocka) [1]
 12       n12 put-down ...goal loop with node 8
 12       n13 stack ...goal loop with node 8
 13       n15 <unstack blockc blocka> [2]
 14       n16 <UNSTACK BLOCKC BLOCKA> [1]
Firing apply/subgoal AVOID-APPLY-FOR-WRONG-GOAL deciding SUB-GOAL
 15   n17 (on blockb blockc)
 17   n19 <stack blockb blockc>
 18     n20 (clear blockc) [1]
 20     n22 <put-down blockc>
 21     n23 <PUT-DOWN BLOCKC> [1]
 22     n24 (holding blockb)
 24     n26 <pick-up blockb>
 25     n27 <PICK-UP BLOCKB>
 26   n28 <STACK BLOCKB BLOCKC> [1]
 27     n29 <PICK-UP BLOCKA>
 27   n30 <STACK BLOCKA BLOCKB>
Achieved top-level goals.

Solution:
<unstack blockc blocka>
<put-down blockc>
<pick-up blockb>
<stack blockb blockc>
<pick-up blocka>
<stack blocka blockb>
```

Figure 4: Example Blocksworld Trace.

In order to do that the block has to be held, but `(holding blocka)` is the goal PRODIGY was trying to achieve in the first place. We call this situation a *goal loop*. PRODIGY detects this failure and tries a different operator. In this case UNSTACK is finally chosen at node `n15` and as its preconditions are satisfied in the initial state, the operator is applicable; in fact PRODIGY decides to apply it immediately, shown by the applied operator node `n16`. Below this point in the trace, PRODIGY's internal state has been updated to reflect that `<unstack blockc blocka>` was applied. For example, `(holding blockc)` is now true. Note that so far in the trace at the end of each line referring to a subgoal or an operator application, there is a `[1]`. This means that at each of those points PRODIGY could have considered a different alternative. In this example, the alternative corresponds to work on the other top-level subgoal, `(on blockb blockc)`. [1]

At node `n17` PRODIGY switches attention to the other top-level goal, `(on blockb blockc)`, even though `(on blocka blockb)` has not been achieved, because `<stack blocka blockb>` has not been applied. This switch of attention is caused by a control rule, called `avoid-apply-for-wrong-goal`. Control rules direct PRODIGY's decision at each point. When the control rule preconditions are satisfied, the rule fires and selects, rejects, or prefers an alternative. The type of alternative depends on the node PRODIGY is at. In this example, the rule guides PRODIGY to prefer working on goal `(on blockb blockc)` instead of applying `<stack blocka blockb>`.

Once again, the system follows the goal tree from Figure 3, choosing the operator `<stack blockb blockc>` at node `n19` to achieve the goal. It works on `(clear blockc)`, first of the preconditions of this operator, and applies the operator `<put-down blockc>` immediately at node `n23`. This changes the state again, so that `<pick-up blockb>` is also applied immediately at node `n27`. Having achieved the preconditions of `<stack blockb blockc>` from node `n19`, it is applied and the second top-level goal `(on blockb blockc)` is achieved.

Now PRODIGY switches attention back to the work remaining for the first top-level goal, the one on the left of the goal tree. Since the goal `(clear blocka)` achieved by the operator application `<unstack blockc blocka>` at node `n16` has not been undone by the intervening work, and the other preconditions are now true, the operator `<pick-up blocka>` chosen at node `n10` is now applicable, and is applied at node `n29`. Finally, `<stack blocka blockb>` from node `n7` is applied at node `n30`, and both the top-level goals are solved. The final plan listed is the sequence of nodes corresponding to operator applications, taken in the order they were applied.

This trace summarizes the simple style of planning that PRODIGY performs under normal circumstances. By default, this is a depth-first search algorithm which chooses operators and instantiations for them in a left-to-right order. This would be very inefficient for most problems, were it not for the application of *control knowledge* to guide the search. Control rules can be specified for every choice point that was shown in the trace. The subject of control knowledge is discussed in more detail in Section 4.

3 Defining a Domain

We present in this section the language that can be used in defining PRODIGY domains. The power of PRODIGY's problem solving engine is owed in part to the flexibility and expressibility of this representation language.

As we saw in the previous section, in order to solve problems in a particular domain, PRODIGY must be first given a specification of that domain, which consists of a type hierarchy for the entities of the domain, a set of operators, inference rules, and search control rules. The PRODIGY system includes a variety of additional capabilities to express domain knowledge that enhance its flexibility and usability. In this section,

[1] PRODIGY allows full interleaving of goals at each point of the search, which confers PRODIGY its nonlinear character. For more on nonlinear problem solving we refer the reader to [Veloso, 1989].

we present both basic and advanced features of PRODIGY's expressibility using examples from a domain presented in detail in Appendix C. It is an extended version of the STRIPS [Fikes and Nilsson, 1971] domain. A robot can move between rooms, and transport boxes from one room to another. Rooms are connected through doors that may be locked and unlocked with keys.

3.1 Type Specification

Before we describe how to write operators for a domain, we need to specify the objects to which they can be applied. In our robot domain, for example, the objects are the robots, the boxes, and the rooms with their doors and keys.

The class or type of every object in a domain must be specified. The type hierarchy in PRODIGY forms a tree structure, i.e., a type can only be a subtype of one other type. Each type in the hierarchy has a super type and several subtypes, except for the root (called `:top-type`) that has only subtypes. A possible type hierarchy for our robot domain is shown in Figure 5.

```
(ptype-of ROBOT :top-type)
(ptype-of ROOM :top-type)
(ptype-of DOOR :top-type)
(ptype-of OBJECT :top-type)
(ptype-of BOX OBJECT)
(ptype-of KEY OBJECT)
```

Figure 5: A Simple Type Hierarchy for a Domain with Robots, Boxes, Rooms and their Doors and Keys.

Any instances of types that are constant in a domain can be specified as such. For example, if a robot `Robby` is the only robot in our domain, we could write:

```
(pinstance-of Robby ROBOT)
```

Some types have an infinite number of instances (such as the integers). Section 3.4.5 describes how to specify these infinite types.

3.2 Operators and Inference Rules

Operators and inference rules share the same syntax, but their semantics are different. Operators should be thought of as representations of actions that can produce changes in the state. Inference rules represent deductions to be derived from the existing state. We use the term operator to refer to both operators and inference rules unless we indicate otherwise.

Each operator has a precondition expression that must be satisfied in the state before the operator can be applied. An operator also has a set of effects that describes how the application of the operator changes the world. Precondition expressions are well-formed formulas in a typed first order predicate logic encompassing negation, conjunction, disjunction, and existential and universal quantification. Each effect is an atomic formula together with a particle that indicates if the formula should be added or deleted from the state.[2] In addition one can also include conditional effects that specify transformations to the world that depend on the current state when the operator is applied, as we describe in Section 3.4.3.

The specification for an operator or inference rule also includes a list of variables that are to be considered *parameters*. Parameters are the variables that you want the user interface to print out in the trace. (Often

[2]Note that inference rules can delete atomic formulas from the state as well as add them. This is different from previous versions of PRODIGY.

an operator may have many variables, and it is unnecessary to print them all out). Universally quantified variables cannot be listed as parameters, since they will have multiple values.

For example, suppose that we want to define an operator PICKUP-OBJ, so a robot can pick up objects that are next to it when its arm is not holding other objects. Figure 6 defines this operator. The figure also includes an example of an inference rule to take advantage of the fact that the predicate connects is symmetrical.

```
(OPERATOR PICKUP-OBJ
 (params <object>)
 (preconds
  ((<robot> ROBOT) (<object> OBJECT))
  (and (arm-empty)
       (next-to <robot> <object>)
       (carriable <object>)))
 (effects
   ((<other-obj> (or OBJECT DOOR))
    (<other-obj2> (or OBJECT DOOR)))
   ((del (arm-empty))
    (del (next-to <object> <other-obj>))
    (del (next-to <other-obj2> <object>))
    (del (next-to <robot> <object>))
    (add (holding <object>)))))
```

```
(INFERENCE-RULE
  CONNECTS-IS-SYMMETRICAL
 (params <door> <room1> <room2>)
 (preconds
  ((<door> Door)
   (<room1> Room)
   (<room2> Room))
  (connects <door> <room1> <room2>))
 (effects ()
  ((add (connects <door> <room2> <room1>)))))
```

Figure 6: Operator and Inference Rule from the Extended-STRIPS Domain.

Each variable in an operator or inference rule must be typed by associating a type specification with the variable. If the variable is used in the precondition of the operator or inference rule, the type specification must be given before the precondition list. If the variable is only used in the effect list, the type specification must be given in the effects list. For example, the operator PICKUP-OBJ shown in Figure 6, has as variables <object>, <robot>, <other-obj>, and <other-obj2>. The type specifications for <robot> and <object> are given in the precondition list because they are used in the precondition expressions. Type specifications for <other-obj> and <other-obj2> are given in the effect list because these variables are only used in the effect list.

Type specifications are used to restrict the range of values that can be used as bindings for a variable. For example, if the variable <box> of an operator is typed as BOX, then <box> can only be instantiated with objects declared to be of type BOX. Types are not the only way to restrict the bindings of a variable, as we show in Section 3.4.2.

3.3 Specification of a Problem

A problem in PRODIGY is given by an initial state and a goal state specification. The description of an initial state (or any state, for that matter) is composed of a list of objects and their corresponding types together with a set of instantiated predicates (i.e. literals) that describes the configuration of those objects. The objects in the state must be instances of the types that are declared in the domain. For example, given the type hierarchy in Figure 5, an initial state can be:

```
(objects
   (room1 room2 room3 ROOM)
   (door12 door23 DOOR)
   (key23 KEY)
```

```
    (box1 BOX))

 (state
   (and
    (connects door12 room1 room2)
    (connects door23 room2 room3)
    (inroom robby room1)
    (inroom key23 room1)
    (inroom box1 room3)))
```

The goal specification is an expression that must be satisfied by any goal state. It may contain variables, and it can be quantified. For example, legal goals in this particular domain may be expressed as follows:

```
(goal (inroom box2 room1))
(goal (~(inroom box2 room1)))
(goal ((<box> BOX)) (inroom <box> room1))
```

The first goal is true in any state where `box2` is in `room1`. The second goal is true in any state where `box2` is not in `room1`. The last goal is true in any state where some box is in `room1`. Notice that when there is a variable in the goal expression, the variable must be typed.

A problem is created by specifying both the initial state and the goal description. Figure 7 illustrates how a problem is defined in PRODIGY4.0.

```
(setf (current-problem)
      (create-problem
        (name test-prob)
        (objects
          (room1 room2 room3 ROOM)
          (door12 door23 DOOR)
          (key23 KEY)
          (box1 BOX))
        (state
         (and
           (connects door12 room1 room2)
           (connects door23 room2 room3)
           (inroom robby room1)
           (inroom key23 room1)
           (inroom box1 room3)))
        (goal (inroom box1 room1))))
```

Figure 7: An Example of Defining a Problem in PRODIGY4.0.

3.4 Beyond the Basics

This section describes in detail some of the most powerful components of PRODIGY'S description language, namely: quantifiers, static properties, and conditional effects. It also describes how to make planning with inference rules more efficient, and talks about the use of infinite types.

3.4.1 Quantification

Expressions in PRODIGY may contain quantifiers, both existential and universal. The type of the variables within the scope of the quantifiers must be specified. For example, the following expression indicates that there is a door in the room where the robot is. Notice that the quantifier can range over several variables.

```
(exists ((<room> ROOM)(<door> DOOR))
   (and ((inroom robot <room>)
         (dr-to-room <room> <door>))))
```

We can use a universal quantifier to express that all the doors are open as follows:

```
(forall (<door> DOOR)
  (dr-open <door>))
```

It is possible to have nested quantifiers in an expression. For example, the following expression can be used to specify that there must be keys for all the doors that are locked.

```
(forall (<door> door)
  (or (unlocked <door>)
      (exists (<key> KEY)
         (is-key <key> <door>))))
```

Since it can be quite long to write down a complex formula containing many existentially quantified variables, we allow a form of shorthand. Any existentially quantified expression that is not within the scope of a negation can be abbrieviated as a conjunction. Thus the expression below can be abbreviated as follows:

```
(exists ((<x> XTYPE)) (P <x>))  goes to:  (((<x> XTYPE)) (P <x>))
```

Thus, the existential quantification of `<x>` is implicit.[3]

In PRODIGY, there is an implicit *universal* quantification of unbound variables in the effects list that allows us to implement the "wildcard" feature that was employed by the original STRIPS system [Fikes and Nilsson, 1971, Fikes *et al.*, 1972]. Thus, one can easily specify multiple deletions. For example, in Figure 6, `<other-obj>` is a wildcard variable for operator PICKUP-OBJ. When the PICKUP-OBJ operator is applied PRODIGY will delete all atomic formulas that match `(next-to <object> <other-obj>)`.

3.4.2 Static Properties

The properties of the objects in a domain that never change are called *static*. This means that their truth value remains the same in any state of the search space. Thus, a static property cannot appear as an effect of an operator. For example, the fact that $2 < 5$, or that a door connects two rooms are both static.

There are several ways to implement static properties in PRODIGY. One way is to represent them as predicates and list them in the initial state. For example,

```
(is-key key12 door12)
```

[3]The scope of an implicitly quantified variable is *inside* the scope of an enclosing universally quantified statement iff all occurrences of that variable are physically within the statement. Generally the use of existential shorthand is not used in such cases to avoid confusion.

represents the unchangeable fact that `key12` opens `door12`.

Another possibility is to represent static properties as predicates that can be deduced by inference rules. For example, we can represent the fact that doors connect rooms in the state as follows:

```
(connects door12 room1 room2)
(connects door23 room2 room3)
```

and we can use the inference rule CONNECTS-IS-SYMMETRICAL shown in Figure 6 to deduce some new static properties. Thus, the literals:

```
(connects door12 room2 room1)
(connects door23 room3 room2)
```

can be added to the state by the inference rule. Section 3.4.4 describes in more detail the use of inference rules for deducing static properties.

Some static properties cannot be implemented efficiently as predicates. Consider for example the property that tests whether two objects are different. This property could be implemented as a predicate, but in a problem with n objects we would have to create n^2 literals (one for each pair of different objects). If the predicate ranges over an infinite number of objects, like "`<`", it would not be reasonable to generate all possible pairs. Rather, it would be nice to have the ability to generate on demand the particular pairs needed for problem solving. In order to implement these static properties efficiently, PRODIGY allows user-defined Lisp *functions* to represent them. In our example, we can define the function `diff` as follows:

```
(defun diff (x y)
  (not (eq x y)))
```

The function returns `t` or `nil` depending on whether `x` and `y` are different objects or not (`nil` indicates that the formula is false.) This function may be used in operators to find out whether two PRODIGY objects are the same or not. When a function like `diff` is used in an operator it restricts the values that variables in the operator can take. The type specification of a variable constrains the range of values that it can take. A function further restricts the range of values to those values that pass the test in the function.

For example, consider the operator that puts down a block next to another. Both of these blocks must be different, since we cannot put a block next to itself. This is specified as follows:

```
(operator PUTDOWN-NEXT-TO
 (params <object> <other-obj> <room>)
 (preconds
  ((<object> OBJECT)
   (<other-obj> (and OBJECT (diff <object> <other-obj>)))
   (<room> ROOM))
  (and (holding <object>)
       (inroom <other-obj> <room>)
       (inroom <object> <room>)
       (next-to robot <other-obj>)))
 (effects
  ()
  ((del (holding <object>))
   (add (next-to <object> <other-obj>))
   (add (next-to robot <object>))
   (add (next-to <other-obj> <object>))
   (add (arm-empty)))))
```

3.4.3 Primary and Conditional Effects

The effect list specifies the changes to be performed in the world when the operator is applied. These changes can be unconditonal, or conditional to a particular state of the world (i.e., context dependent). *Primary* or unconditional effects indicate unconditional changes, i.e., the formulas that are added to or deleted from the state when the operator is applied. *Conditional* effects describe changes to the world also in terms of formulas to be added or deleted to/from the state, but as a function of a particular state configuration.

A conditional effect is composed by a condition expression and a primary effect. The effect should be added or deleted to the current state only if the condition expression is true when the operator is applied. The types of the variables in the condition must be specified in the effects list of the operator. If there are multiple ways to bind the variables in the expression, then there can be multiple formulas that are added or deleted. As an example, consider the following alternative representation for the operator PICKUP-OBJ. If there are any objects that were next to the box being picked up they will not be next to it any longer once the operator is applied. This fact can be expressed as a conditional effect as follows:

```
(OPERATOR PICKUP-OBJ-2
 (params (<object>))
 (preconds
  ((<robot> ROBOT) (<object> OBJECT))
  (and (arm-empty)
       (next-to <robot> <object>)
       (carriable <object>)))
 (effects
  ((<other-obj> OBJECT))
  ((add (holding <object>))
   (del (next-to <robot> <object>))
   (if (next-to <object> <other-obj>)
       ((del (next-to <object> <other-obj>)))))))
```

When the operator PICKUP-OBJ-2 is applied, the robot is holding the object bound to the variable <object>, and for any objects that satisfy the restrictions on <other-obj> the predicate (next-to <object> <other-obj>) is deleted. Notice that because <other-obj> is not defined by the preconditions of the operator, it is treated as a universally quantified variable.

Conditional effects are considered for backward chaining. For example, the above operator will be considered relevant if the goal (~ (next-to <x> <y>)) arises.

3.4.4 Inference Rules

As we described in Section 3.2, inference rules are used in PRODIGY to derive facts from the existing state. Given a state, there may be a large number of facts that can be deduced by firing inference rules. This section shows how to write inference rules while keeping search efficient.

In Section 3.4.2 we show how to use the inference rule CONNECTS-IS-SYMMETRICAL to deduce new static properties. So given the fact (connects door12 room1 room2) in the initial state, we can deduce that (connects door12 room2 room1).

Since this is a static property, it is always true during problem solving. So if the inference rule fires in the initial state, it does so only once for each door. Otherwise, it would fire when the fact that a door connects two rooms is needed during problem solving. In this case, we may need to fire the rule several times for each door if its effect is needed in different parts of the search tree (for example after backtracking). Firing an inference rule automatically in the initial state can be done in PRODIGY by marking the rule as *eager*. Eager rules are used in a forward-chaining manner, and are fired automatically whenever the preconditions are satisfied. Their effects are never used for backward chaining.

Now consider, for example, that the locations where the robot can move are specified by a grid. The robot can move from location (x, y) to location (z, t) only if they are adjacent. We can write an inference rule ADJACENT-LOCATIONS to deduce when two locations are adjacent. However, we would not like to fire the inference rule for each possible pair of adjacent locations (imagine that we have a 100x100 grid!). What we want is to fire that rule only when we need to know during problem solving if two particular locations are next to each other. Firing inference rules only on demand can be done in PRODIGY by marking the rule as *lazy*. Lazy rules are used for backward chaining, and PRODIGY subgoals on their preconditions if they are not true.

CONNECTS-IS-SYMMETRICAL and ADJACENT-LOCATIONS are both rules that deduce static properties of the state. But inference rules can also be used in PRODIGY to deduce non-static facts. Consider, for example, the inference rule INFER-ARMEMPTY for the blocksworld, shown in Figure 1. Whether the arm is empty or not depends on the current state, and therefore `(arm-empty)` is not static. Since we need to subgoal on `(arm-empty)`, the rule needs to be defined as lazy. Lazy inference rules are very similar to operators: PRODIGY may choose a lazy inference rule to achieve a goal, and as a consequence may subgoal on the preconditions of the inference rule. One difference between a lazy inference rule and an operator is that when the operator becomes applicable, PRODIGY may choose to either apply it or to subgoal on other pending goals, whereas when the lazy inference rule becomes applicable, it is applied immediately.

Suppose now that INFER-ARMEMPTY is fired in state S when the arm is not holding any blocks. Then, suppose that PRODIGY decides to apply the operator PICKUP to hold a block, which transforms S into S'. Notice that `(arm-empty)` ceases to be true in S', so it should be removed from the state. In PRODIGY, the consistency of the state is preserved automatically through a truth-maintenance system (TMS). The TMS keeps track of all the inference rules (both eager and lazy) that are fired. When an operator is applied, the effects of inference rules whose preconditions are no longer true are undone. Then, the TMS checks all the eager inference rules and fires them if the preconditions become true.

In conclusion, inference rules may represent many possible deductions from a given state. In order to implement them efficiently, PRODIGY fires them either automatically or on demand when they are specified as eager or as lazy, respectively (the default is lazy). Figure 8 shows an example of each mode. A TMS preserves the consistency of the state after an operator is applied.

```
(INFERENCE-RULE INFER-ARM-EMPTY
 (mode lazy)
 (params)
 (preconds
   ()
   (~ (exists ((<obj> OBJECT))
        (holding <obj>))))
 (effects
   ()
   ((add (arm-empty)))))
```

```
(INFERENCE-RULE
  CONNECTS-IS-SYMMETRICAL
 (mode eager)
 (params <door> <room1> <room2>)
 (preconds
  ((<door> Door)
   (<room1> Room)
   (<room2> Room))
  (connects <door> <room1> <room2>))
 (effects ()
  ((add (connects <door> <room2> <room1>)))))
```

Figure 8: Eager and Lazy Inference Rules.

3.4.5 Infinite Types

If the user wants to specify a type with infinitely many elements, e.g. an integer or a real number, then its class must be declared as an *infinite type*. The instances of an infinite type are not stored in the type hierarchy. Instead, they are generated as needed.

When defining an infinite type in a domain, one must also specify a membership function that allows the system to check for valid instances. As an example, suppose now that we want to specify the weight of an object as a number. Then the Lisp function `numberp` can be used as membership function, and the type can be defined as follows:

```
(infinite-type WEIGHT #'numberp)
```

When a variable in an operator is declared to have an infinite type, in addition to specifying the type of the variable, we must also provide a function that generates bindings for this variable (or, if the variable is bound from the right-hand side, simply checks those bindings). This function is part of the specification for the variable, and it must return a list of values as the bindings generated for the variable. These values may depend on the current state, or on the problem-solver meta-state.

For example, suppose that we want to modify the extended-STRIPS domain so that a robot can only pick up an object whose weight is less than that of the robot. The weights of the objects and the robot may be specified in the initial state as follows:

```
    (is-weight robot 30)
    (is-weight light-block 10)
    (is-weight heavy-block 40)
```

Then the following is a modified operator PICKUP that takes into consideration the weight constraints.

```
(Operator PICKUP
  (params <obj> <weight>)
  (preconds
     ((<obj> OBJECT)
      (<rob-weight> (and WEIGHT (gen-from-pred (is-weight robot <rob-weight>))))
      (<weight> (and WEIGHT (gen-from-pred (is-weight <obj> <weight>))
                            (less-than <weight> <rob-weight>))))
     (arm-empty)
  (effects
    ()
    ((add (holding <obj>))
     (del (arm-empty))))))
```

The function `gen-from-pred` is provided by PRODIGY to generate a list of values as the bindings for a variable by using the information on the current state. In this example, in the case of `<rob-weight>`, `gen-from-pred` returns the list of values $\{x\}$ such that the literal (`is-weight robot` x) is true in the current state. In the case of the variable `<weight>`, the function `less-than` is also used in order to allow only values that are smaller than the weight of the robot. Therefore this operator can be applicable when `<obj>` is `light-block`, but will not be applicable when `<obj>` is `heavy-block`.

Notice that if we use the function `gen-from-pred` to generate bindings for an operator, the values it returns depend on the state at the time when the bindings are generated, i.e., the state when PRODIGY subgoals on the preconditions of this operator. When the operator is actually applied, the state may have changed. If the predicate used to generate the bindings (the argument of `gen-from-pred`) is not static, the operator may be applied incorrectly, because the bindings were generated at subgoaling time, and cannot be changed at applying time. For example, in the extended-STRIPS domain, the weight of the robot, 30, cannot be changed by any operator. But the weight of an object can be changed by applying some operators (for example, by putting a block inside of a box). When we subgoal on the operator PICKUP, the weight of the object is 10, therefore the binding for `<weight>` is 10. Suppose an operator is applied between the time we subgoal on the operator, and the time we actually apply it, and the weight of the object is changed to 35 as an effect of this operator. Then when we apply PICKUP, the binding for `<weight>` is still 10, so the operator PICKUP is applied with `<weight>` bound to 10, although the robot cannot pick the object up because the current weight of the object, 35, is greater than the weight of the robot.

According to this, a non-static predicate should not be used by `gen-from-pred` to generate bindings for a variable. The following is a modified example of the operator PICKUP that works correctly even when the weight of an object can be changed. Function `gen-weight-less-than` generates a list of values as bindings for the variable `<weight>`. These bindings are the integers smaller than the value of `<rob-weight>` but greater than zero. As `(is-weight <obj> <weight>)` is now a precondition, the bindings for the operator will only be the correct ones at application time.

```
(Operator PICKUP
  (params <obj> <weight>)
  (preconds
     ((<obj> OBJECT)
      (<rob-weight> (and WEIGHT (gen-from-pred (is-weight robot <rob-weight>))))
      (<weight> (and WEIGHT (gen-weight-less-than <weight> <rob-weight>))))
     (and (is-weight <obj> <weight>)
          (arm-empty)))
  (effects
    ()
    ((add (holding <obj>))
     (del (arm-empty)))))

(defun gen-weight-less-than (weight rob-weight)
  (let ((possible-weights nil))
    (do ((w rob-weight (1- w)))
        ((zerop w) possible-weights)
     (push w possible-weights))))
```

3.5 PRODIGY4.0's Description Language (PDL4.0)

In Appendix B we list the regular expressions that describe PRODIGY4.0's description language. The rest of this section describes PDL's syntax and semantics from a more intuitive and practical point of view.

3.5.1 PDL Syntax

Here is a brief description of the syntactically legal sentences in PDL:

- Every atomic formula is a sentence.

- If F_1, F_2, ...F_n are sentences, then so is their conjunction, (AND F_1, F_2, ...F_n) and their disjunction, (OR F_1, F_2, ...F_n)
- If F is a sentence then so is its negation, (~ F)
  However, F must be either an atomic formula or an existentially quantified statement.
- A type specification is either a type name, or a conjunction of types, or the set difference of two types. A descriptor for a variable is either a type specification or a type specification followed by some functions that further restrict the variable values.
- If F is a sentence, <v_1>, <v_2>, ..., <v_n> are variables, and D_1, D_2, ..., D_n are descriptors for each variable, then the following is a sentence,

  (EXISTS ((<v_1> D_1) (<v_2> D_2) ...(<v_n> D_n)) SUCH THAT F)

  as well as,

  (FORALL ((<v_1> D_1) (<v_2> D_2) ...(<v_n> D_n)) SUCH THAT F)

3.5.2 PDL Semantics: Matching Expressions in a State

Each node in PRODIGY's search tree is associated with a *state* describing the world at that node. A state consists of a set of literals that are known to be true of the world, given the sequence of operators and inference rules which created that node.

A PDL expression describes a state if the expression can be *matched* in that state. Thus:

- An atomic formula matches iff the formula is present in the state.
- A conjunctive formula matches iff all of its conjuncts match.
- A disjunctive formula matches iff at least one of its disjuncts matches.
- The negation of an expression matches iff the expression does not match.
- An existentially quantified expression consists of a list of variables with their generators and an expression. An existentially quantified expression matches iff there exists an assignment of values to each of the variables that satisfy the generators, such that the expression matches the current state.
- A universally quantified expression consists of a list of variables with their generators and an expression. A universally quantified expression matches iff for all variable assignments that satisfy the generators, the expression matches the state.

3.6 Upgrading Domains From PRODIGY2.0

PRODIGY2.0 was released a few years ago [Minton *et al.*, 1989b] and turned out to be a very popular version of our system. Many users have implemented application domains for the old version. In this section we give some pointers as to how to convert them into PRODIGY4.0.

Types must be added to all the variables and objects in the domain. A first step would be to create the type hierarchy, and then specify in all the operators and inference rules the type of the variables that they use. Then, all the objects used in problems must be declared as instances of one of the types that are defined in the domain.

Generators no longer exist when upgrading to PRODIGY4.0 (except as explained in Section 3.4.5). All the functions that used to be in the precondition expression now become a part of the specification of variables in the operator or inference rule. The code for the functions themselves must be changed too. They do not return bindings or `'no-match- attempted` any more. They should return only `t` or `nil`.

If you would like to see some examples, just compare the old version of the extended-STRIPS domain with the new version in Appendix C.

4 Control Structure

In this section, we describe the control structure of PRODIGY in detail, starting with a review of the basics. PRODIGY explores the search space by building a tree of nodes, and each type of decision made corresponds to a different type of node. The following subsections explain PRODIGY's nodes, decisions, and how backtracking in the search tree occurs.

4.1 The Basics

A problem consists of an initial state and a goal expression (which can be an arbitrary PDL expression). To solve a problem, PRODIGY must find a sequence of operators that produces a state satisfying the goal expression.

PRODIGY uses means-ends analysis. Basically, in a backward chaining mode, given a goal predicate not true in the current world, the planner selects one operator that adds that goal to the world. We say that this operator is *relevant* to the given goal. If the preconditions of the chosen operator are true in the *state*, the operator can be *applied*. *Applying* an operator means *executing* it in the *internal* world of the problem solver, which we refer to by *world* or *state*. If some of the preconditions of the operator are not true in the state, they become *subgoals*, i.e., new goals to be achieved. A goal is *pending* if it is a precondition of a *chosen* operator that is not true in the state. PRODIGY's nonlinear character stems from working with the **set** of pending goals, as opposed to just the top goal in a goal stack. [4]

The search tree is expanded by repeating the following steps: first compute the set of pending goals and the possibly applicable operator, and decide either to apply that operator or choose one of the pending goals to work on. If an operator was chosen as applicable, then apply it. If a goal was chosen, expand it, i.e. choose an operator to achieve that goal, and choose bindings for the variables in the operator. This cycle repeats until all the conjuncts in the goal expression are true, i.e. search terminates after creating a node whose state satisfies the top-level goal expression. We call this a goal state. In other words, PRODIGY will halt after finding the first solution to the problem that it encounters. Notice that there may be many goal states that satisfy the goal expression given in the problem.

At each step in the decision phase, PRODIGY must choose between a set of candidates: nodes, goals or an applicable operator, operators, or bindings. The decision process can be mediated by control rules, as described in Section 5.

Even though a candidate has been chosen at a decision point, PRODIGY can later return (backtrack) to that same decision point and try choosing another candidate. The following sections describe each step in the decision phase in detail.

4.2 What Is a Node?

When PRODIGY is solving a problem, it explores the search space by building a tree of nodes. Each node represents and records a different decision made by the problem solver. To distinguish between the different

[4] This is one of the main differences between PRODIGY4.0 and PRODIGY2.0. For a detailed description of linear and nonlinear planning see [Veloso, 1989].

types of decisions, PRODIGY uses different types of nodes, namely goal nodes, operator (or inference rule) nodes, binding nodes, and applied operator nodes [5]. All the node types have some common features such as parent, children, and alternatives not explored yet, if any. In addition a goal node holds the goal chosen at that point. An operator node is always a child of a goal node, and maintains the operator chosen, which is not instantiated yet. A bindings node is a child of an operator node and contains an instantiation of that operator. Finally, an applied operator node is a child either of a bindings node or of another applied operator node, and records an operator application. For example, in the trace shown in Figure 4, node 24 is a goal node that holds goal `(holding blockb)`. Its child node, node 25, (not shown) is an operator node and represents the choice of PICK-UP to achieve that goal. In turn its child node 26 is a binding node that holds the instantiation of PICK-UP `<pick-up blockb>` that may achieve the goal at node 24. Node 27 records the application of that instantiated operator.

4.3 Choosing a Node

When PRODIGY is choosing a node, the default set of candidates includes all the nodes in the search tree that have not been exhausted. A node is exhausted if PRODIGY has completely expanded the node, i.e., created all possible children for that node. If no control rules are applicable, PRODIGY chooses the last node that has been created, resulting in a depth-first search. However, this default strategy can be modified by adding explicit control rules to guide the process of choosing a node. One can also select the search strategy as part of the problem space (see Section 8.1 for an explanation of problem spaces). Breadth-first search is another strategy provided.

The usual order in which nodes are expanded, i.e. the order in which decisions are made, is goal, operator, bindings, and then either one or more operator application or goal again. However, at any point PRODIGY may decide to explore a different path (we call this a *change of context*) and therefore break the usual order. Depending on the type of the chosen node, a different action will be performed. Even when a change of context occurs, the order of the nodes as read down any search path always conforms to the above description.

4.4 Choosing a Goal

The first step in the search cycle is to determine which goal to work on. The candidate goals are the unmatched preconditions of the instantiated operators that have been chosen during the search so far [6]. Once a goal is chosen, a goal node is created. All goals must eventually be achieved for the solution to be reached, but the problem solver only focuses on one goal at a time. At any point during the search however it may change its focus and decide to work on a different goal, allowing therefore the nonlinear character. For example, in the problem presented in Section 2, PRODIGY starts working on the goal `(on blockA blockB)`, but at node 17, before that goal is achieved, the planner switches its attention to a different goal, `(on blockB blockC)`, therefore interleaving the plans that achieve each goal. Should nonlinear planning have been disallowed, PRODIGY would have failed without finding a solution for this problem.

4.5 Choosing an Operator

After choosing a current goal, the planner chooses an operator to achieve that goal by looking at the effects, including the conditional effects, of the operators. The default set of candidates consists of all operators

[5] PRODIGY2.0 used a single type of node.

[6] At the beginning of search the goals come from the initial goal statement. The process of matching the preconditions of the parent operator to produce subgoals is described in Section 4.6.

that are relevant to the goal. In fact, at this point PRODIGY considers both operators and inference rules. In this section we refer to both by operators.

In the absence of control knowledge, operators are considered in the order stated in the domain specification. Once an operator is chosen, an operator node is created.

An operator may have more than one effect that unifies with the goal. Unifying a relevant effect with the goal produces bindings for some of the variables in the operator (we call this a *partial set of bindings* for the operator), since the variables that occur in the effect of the operator will be bound. Each relevant effect produces a different set of partial bindings. Given a set of partial bindings, the next step is to expand this set of bindings by matching the preconditions, as described below.

4.6 Choosing a Set of Bindings for the Operator

The process of choosing the bindings for an operator, referred to as 'instantiation', is one of the most involved steps in the decision phase. The bindings indicate values for the variables that are existentially quantified in the operator's preconditions. [7]

As indicated above, the process of determining the relevant effects of the operator generates (possibly several sets of) partial bindings. For each set of partial bindings, PRODIGY calls the matcher and attempts to match the operator's preconditions. The bindings for all the variables in the precondition list of an operator are generated at this time. However the bindings for variables in the effect list of an operator are generated only when the operator is applied. (An exception is when we are subgoaling on a conditional effect: then the bindings for the variables in the conditional effect are also generated when we subgoal on the operator.) PRODIGY4.0's matching algorithm is described in [Wang, 1992].

Once a set of bindings for the operator is chosen, a bindings node is created. This process also creates subgoals corresponding to the preconditions unmatched under those bindings. These preconditions become pending goals. Here PRODIGY checks to make sure that there is no *goal cycle*. A goal cycle occurs if one of the new subgoals is subsumed (i.e., matched) by a goal already present in the current search path. A goal cycle signals an infinite regress, and therefore if a goal cycle is detected, the bindings node is terminated and declared a failure.

If the bindings were chosen for an inference rule, and the rule is applicable, the problem solver will apply it immediately. If the rule is not applicable, its unmatched preconditions will be new subgoals, as in the operator case.

Let us consider an example from the blocksworld domain defined in Section 2. The UNSTACK operator in figure 9 is taken from that domain. At node 11 in the example problem described in Figure 4 the goal is `(clear blockA)` and in the state blockC is on blockA, and both blockA and blockB are on the table. UNSTACK is one of the relevant operators that is considered, [8] and it generates three alternatives, `<unstack blockA blockA>`, `<unstack blockB blockA>` and `<unstack blockC blockA>`, as shown below:

[7] Variables within the scope of a universal quantifier are not included in the bindings, since they may have more than one value.

[8] PUTDOWN and STACK are also relevant, but they lead to a goal-loop since `(holding blockA)` is one of their preconditions.

```
(OPERATOR UNSTACK
  (params <ob> <underob>)
  (preconds
   ((<ob> Object)
    (<underob> Object))
   (and (on <ob> <underob>)
        (clear <ob>)
        (arm-empty)))
  (effects
   ()
   ((del (on <ob> <underob>))
    (del (clear <ob>))
    (del (arm-empty))
    (add (holding <ob>))
    (add (clear <underob>)))))
```

Figure 9: Blocksworld UNSTACK Operator.

BINDINGS	SUBGOALS
1. (<ob> blockA)(<underob> blockA)	(ON blockA blockA) (CLEAR blockA) (ARM-EMPTY)
2. (<ob> blockB)(<underob> blockA)	(ON blockB blockA) (ARM-EMPTY)
3. (<ob> blockC)(<underob> blockA)	(ON blockC blockA) (ARM-EMPTY)

The alternatives are generated as follows: <underob> is bound to blockA during the process of determining that the operator is relevant. As the operator type information specifies that <ob> is of type OBJECT, the matcher generates the three alternative bindings. Attempting to match the preconditions determines the subgoals for each of these candidates.

4.7 Applying an Operator

Once an operator has been completely instantiated, if all of its preconditions are satisfied, the operator can be applied. Note, however, that just because an instantiated operator can be immediately applied does not guarantee that it is always best for solving the global problem. (For example, consider the case where the goal is to be holding blockA but the robot is holding blockB. When the subgoal (arm-empty) arises, it is a bad idea to STACK blockB on blockA, even though that instantiation of STACK may be immediately applicable.)

At the point when all the preconditions are satisfied, a decision has to be made between applying the operator or continue subgoaling. If the decision is to subgoal, a new goal to work on has to be chosen, as described in Section 4.4. If the decision is to apply an applicable operator, an applied-operator node is created, and the operator is applied. The operator considered to be applicable is the last chosen operator not applied yet in the current search path. Note that we can apply several operators in sequence by repeatedly choosing them in successive cycles.

Applying an operator means performing its effects into the current state. Therefore the set of pending goals needs to be updated after these changes in the state. The problem solver also checks if any eager inference rules may be applied and applies them if so. If some inference rules fire, the state is updated and the problem solver checks again for applicable eager inference rules. This process is repeated until no more changes occur, since these inference rules can be fired in a chain.

Once an operator is applied, PRODIGY checks to make sure that there is no *state cycle*. A state cycle occurs if the new state is equivalent to a state higher up in the tree (at one of the node's ancestors). If a state cycle is detected, the child node is terminated and declared a failure. [9]

4.8 Backtracking

PRODIGY's search space can frequently be very large. The problem solver successively chooses a goal, operator and bindings. Any of these decisions can be wrong, in which case backtracking will be necessary. PRODIGY backtracks by returning to a node that it has already visited and choosing an alternative not explored yet. Backtracking can occur due to the default depth-first selection of nodes, or due to a user-defined node-selection rule. (In the latter case the term 'backtracking' may not be completely appropriate, but we'll use it anyway.)

This process is illustrated in our original example from Section 2. At node 12 PRODIGY makes an incorrect selection of operator to achieve `(clear blockA)` that leads to failure (a goal loop) at node 12. In fact PRODIGY returns to node 11 one more time before it finds the right operator, UNSTACK.

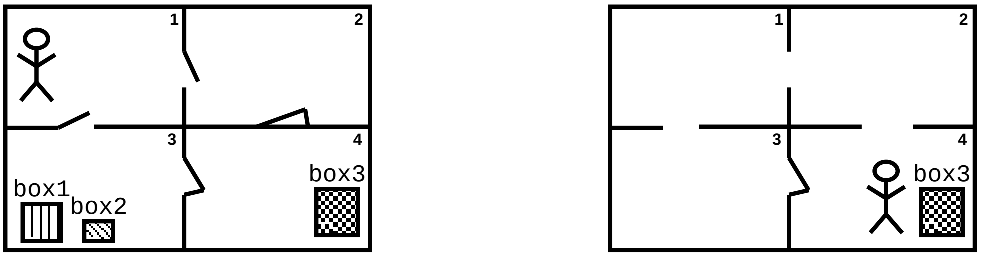
4.9 An Example in the Extended-STRIPS Domain

To further illustrate PRODIGY's control structure and its nonlinear character we present now an example from the extended-STRIPS domain. In this domain there is a set of rooms connected through doors. A robot can move around between the rooms carrying or pushing objects along. Doors can be locked or unlocked. Keys to the doors lay in rooms and can be picked up by the robot. The set of operators include going to be next to objects, going through doorways, pushing objects to rooms, picking up objects and carrying them around, and opening, closing, locking and unlocking doors. Appendix C contains the definition of this domain.

Consider Figure 10 where in (a) we show the initial state and in (b) the goal statement of an example problem from this domain. Rooms are numbered at the corner of their picture. The problem solver must find a plan to reach a state where door34, connecting room3 and room4, is closed, and the agent hero is next to box3.

Figure 11 shows PRODIGY4.0's problem solving trace for this problem. Node 29 is worth remarking. At that point in the search, PRODIGY has two alternatives. The first one is to apply the operator `<GOTO-OBJ BOX3 ROOM4>` immediately, as it becomes applicable when the robot enters room4 (at node 28), and therefore achieve the first goal `(next-to robot box3)`. The other alternative is to subgoal on the other goal `(dr-closed door34)` which became a goal when door34 was opened at node 27. PRODIGY chooses to work on the first alternative, applying `<GOTO-OBJ BOX3 ROOM4>`. Note that this is the default strategy. However it leads to a failure, as after achieving `(next-to robot box3)` the robot has to return back to close door34 at node 35 where the problem solver detects a state loop. PRODIGY recognizes that it was in the same state before, and backtracks to node 29 (it is the first node it encounters where there was some alternative not tried yet). At that point it chooses the correct ordering, and postpones applying the operator after achieving the goal `(dr-closed door34)`.

[9] If there exists a solution that involves a state cycle, then there must also exist a shorter solution that does not involve a state cycle.




```
(inroom robot room1) (inroom box1 room3)
(inroom box2 room3)  (inroom box3 room4)
(dr-open door12)     (dr-open door13)
(dr-closed door34)   (dr-closed door24)
```

(a) Initial State

```
(and (dr-closed door34)
     (next-to robot box3))
```

(b) Goal Statement

Figure 10: Problem Situation in the Extended-Strips Domain. The goal statement is a partial specification of the final desired state: the location of other objects and the status of other doors remain unspecified.

Note that if PRODIGY had been searching in its *linear* mode, it could not have found a solution for this problem due to the occurrence of the state loop. [10]

5 Control Rules

As PRODIGY attempts to solve a problem, it must make decisions about which operator to use and which subgoal to pursue. These decisions can be influenced by control rules for the following purposes:

1. **To increase the efficiency of the problem solver's search.** Control rules guide the problem solver down the correct path at the decision points where they apply so that solutions are found faster. (Not all decision points are typically guided by control rules.)

2. **To improve the quality of the solutions that are found.** There is usually more than one solution to a problem, but only the first one that is found will be returned. By directing the problem solver's attention along a particular path, control rules can express preferences for solutions that are qualitatively better (e.g., more reliable, less costly to execute, etc.).

PRODIGY's reliance on explicit control rules, which can be learned for specific domains, distinguishes it from most domain-independent problem solvers. Instead of using a least-commitment search strategy, for example, PRODIGY expects that any important decisions will be guided by the presence of appropriate control knowledge. If no control rules are relevant to a decision, then PRODIGY makes an arbitrary choice. If the wrong choice is made and costly backtracking is necessary, an attempt may be made to learn the control knowledge that must be missing (this is done by PRODIGY's EBL mechanism, not discussed here). PRODIGY's 'casual commitment' strategy assumes that for any decision with significant ramifications, control rules will be present; if they are not, all the alternatives are equally good.

PRODIGY4.0's search algorithm described in the previous section, involves several choice points, to wit:

- Choosing the node to expand next from the set of open nodes;

- Choosing a goal to work on from the set of pending goals;

[10] A solution could have been found if state loops had been allowed. However it would have been a non-optimal plan, since the robot would have gone first next to box3, then to door34 to close it, and finally next to box3 again.

```
* (run :depth-bound 50 :output-level 3)
  2 n2 (done)
  4 n4 <*finish*>
  5   n5 (next-to robot box3)
Firing select op ONLY-GOTO-OBJ selecting GOTO-OBJ
Firing select binding rule GOTO-OBJ-ROOM
  7   n7 <goto-obj box3 room4>
  8     n8 (inroom robot room4)
Firing reject operator ONLY-PUSH-BOX-THRU-DOOR1 PUSH-THRU-DR
Firing reject operator ONLY-CARRY-BOX-THRU-DOOR CARRY-THRU-DR
Firing reject binding rule GO-THRU-LOCKED-DR
 10     n10 <go-thru-dr door34 room3 room4>
 11       n11 (dr-open door34) [2]
 13       n13 <open-door door34>
 14         n14 (next-to robot door34) [1]
Firing select op ONLY-GOTO-DR selecting GOTO-DR
 16         n16 <goto-dr door34 room4> [1] ...goal loop with node 8
 16         n17 <goto-dr door34 room3>
 17           n18 (inroom robot room3)
Firing reject operator ONLY-PUSH-BOX-THRU-DOOR1 PUSH-THRU-DR
Firing reject operator ONLY-CARRY-BOX-THRU-DOOR CARRY-THRU-DR
 19           n20 <go-thru-dr door13 room1 room3> [1]
 20             n21 (next-to robot door13)
Firing select op ONLY-GOTO-DR selecting GOTO-DR
Firing select binding rule GOTO-DR-ROOM
 22             n23 <goto-dr door13 room1>
 23             n24 <GOTO-DR DOOR13 ROOM1>
 24           n25 <GO-THRU-DR DOOR13 ROOM1 ROOM3>
 25         n26 <GOTO-DR DOOR34 ROOM3>
 26       n27 <OPEN-DOOR DOOR34>
 27     n28 <GO-THRU-DR DOOR34 ROOM3 ROOM4> [1]
 28   n29 <GOTO-OBJ BOX3 ROOM4> [1]
 29   n30 (dr-closed door34)
 31   n32 <close-door door34>
 32     n33 (next-to robot door34)
Firing select op ONLY-GOTO-DR selecting GOTO-DR
Firing select binding rule GOTO-DR-ROOM
 34     n35 <goto-dr door34 room4> ...applying leads to state loop.

 27     n28 <GO-THRU-DR DOOR34 ROOM3 ROOM4> [1]
 28   n37 (dr-closed door34)
 30   n39 <close-door door34>
 31   n40 <CLOSE-DOOR DOOR34>
 31   n41 <GOTO-OBJ BOX3 ROOM4>
Achieved top-level goals.

Solution:
<goto-dr door13 room1>
<go-thru-dr door13 room1 room3>
<goto-dr door34 room3>
<open-door door34>
<go-thru-dr door34 room3 room4>
<close-door door34>
<goto-obj box3 room4>
```

Figure 11: Example Trace in the Extended-Strips Domain.

- Choosing an operator from the set of relevant operators to achieve a particular goal;
- Choosing a set of bindings in order to instantiate the chosen operator;
- Deciding to apply an applicable operator or to continue subgoaling.

Decisions at all these choices are taken based on user-given or learned *control rules* to guide the casual-commitment search. Control rules can *select*, *reject*, or *prefer* certain alternatives to reduce the exponential explosion in the size of the search space. Each control rule has left-hand side conditions testing applicability and a right-hand side indicating whether to select, prefer, or reject a particular candidate. In this section we will describe the different types of control rules in PRODIGY4.0 and we also present some examples for each type. Further examples can be found in Appendix C and in the domains provided with PRODIGY4.0.

5.1 Control Rule Syntax

Each control rule has a name, left-hand side conditions testing applicability, and a right-hand side indicating whether to SELECT, REJECT, or PREFER a particular candidate. The left hand side of the control rule is written in a subset of PDL (see Section 3.5) that only allows the uses of `and`, `or`, and `not`. The predicates in the left hand side of control rules are called *meta-predicates*. Meta-predicates are used to reason about the meta-state of the planner (e.g., the goal that the planner is currently working on, the current operator being considered, etc.). Almost any combination of `and`, `not` and `or`, and is valid. However `not` cannot be used with meta-predicates that serve as generators. Meta-predicates and generators are further described in Section 5.3.

Appendix B includes the syntax for control rules. When writing your own control rules it is helpful to look at examples from other domains, such as those shown in this section and Appendix C.

5.2 The Decision Procedure via Control Rules

In general, to make a control decision, PRODIGY4.0 first applies the *selection* rules to select a set of candidates (nodes, goals, or operators). Usually this set is smaller than the default set generated when there are no select rules; sometimes a single choice is selected. Next, *rejection* rules further filter this set by eliminating candidate decisions that would lead to problem solving failure. Control decisions for bindings work in a different way, as it will be explained in Section 5.2.4. Finally, *preference* rules are used to choose the best alternative from the set of candidates. Preference rules can optimize problem-solver performance (i.e. reduce search), or optimize solution quality. The remaining alternatives may later be explored if the preferred one leads the problem solver to a failure (rejected candidates are never reexamined).

5.2.1 Control Rules for Choosing a Node

the default search strategy in PRODIGY4.0 is depth first. this default can be changed to breadth first search by setting the flag `:search-default` to `:breadth-first` (see section 6.4). node selection, rejection and preference rules can be used to override the defaults, although they are seldom used in PRODIGY4.0.

the following is an example of a select node control rule that can be used for breadth first search instead of setting the flag `:search-default`. The meta-predicate `oldest-candidate-node` tests whether `<node>` is the oldest node created (see Section 5.3).

```
(Control-Rule breadth-first-search
      (IF (oldest-candidate-node <node>))
      (THEN select node <node>))
```

5.2.2 Control Rules for Choosing a Goal

Goal selection, rejection, and preference rules can be used to guide goal decisions in PRODIGY4.0. At each such decision point, PRODIGY first applies the goal selection rules to select a set of candidate goals. If none of these rules fires, then the set of all the pending goals is selected as the set of the candidate goals. The rejection rules are used to further reject some candidates from the set of candidate goals.

It is important to remember that when PRODIGY rejects some goals, it is rejecting these goals for the particular node it is at (Section 4.3 discusses more about nodes). That is, when PRODIGY expands this node, it will only subgoal on the goals from the final set of candidate goals, and not on those that are rejected. The remaining goals will have to be achieved further down in the search tree, since all of the pending goals must eventually be achieved somewhere along the solution path.

The following rule is a prefer goal control rule in the blocksworld domain. In a problem in this domain where the goal is to build a stack of blocks, PRODIGY should try to subgoal on the bottom part of the stack first according to this prefer rule. The right-hand side of the rule means *prefer* to establish one of the on relations (on <y> <z>) over the other on relation.

```
(Control-rule prefer-right-goal
    (IF (and (candidate-goal (on <x> <y>))
             (candidate-goal (on <y> <z>))
             (true-in-state (clear <x>))))
    (THEN prefer goal (on <y> <z>) (on <x> <y>)))
```

5.2.3 Control Rules for Choosing an Operator

Operator selection, rejection, and preference rules can all be used in PRODIGY4.0. Operator selection rules are fired first to generate a set of candidate operators. If no such rule fires, then all the operators relevant to the given goal form the set of candidate operators. Secondly, operator rejection rules are used to further reject some candidates from the set of candidate operators. Finally, preference rules are applied to choose which operator to use first. The following rule is an operator selection rule in the blocksworld domain.

```
(Control-Rule select-op-unstack-for-clear
    (IF (and (current-goal (clear <x>))
             (true-in-state (on <y> <x>))))
    (THEN select operator UNSTACK))
```

5.2.4 Control Rules for Choosing Bindings

Bindings selection, rejection, and preference rules can all be used in PRODIGY4.0. The detailed process of matching these control rules is described in [Wang, 1992]. Basically, when generating appropriate bindings for an operator, PRODIGY combines type constraints, functions, and static information in the operator, as well as the bindings selection and rejection rules, and then it generates alternatives according to these constraints. Then binding preference rules are used to order the alternatives.

The following is a select bindings control rule for operator UNSTACK in the blocksworld domain. In this control rule, <x> and <y> are variables that are bound to objects in the domain when the rule fires. <ob> and <underob> are the name of variables in the operator UNSTACK. So for example, if we are instantiating operator UNSTACK, when the goal we are trying to achieve is (clear B), and (on A B) is true in the state, then variable <ob> is bound to A, and <underob> is bound to B.

```
(Control-Rule select-bindings-unstack-clear
    (IF (and (current-goal (clear <y>))
             (current-ops (UNSTACK))
             (true-in-state (on <x> <y>))))
    (THEN select bindings ((<ob> . <x>) (<underob> . <y>))))
```

5.2.5 Control Rules for Deciding Between Applying or Subgoaling

When PRODIGY is expanding a node where there are some applicable operators as well as pending goals, the default behavior is to apply the applicable operator. However, control rules can be used to override this default so that PRODIGY will continue subgoaling instead.

The following is such an example from the blocksworld domain. The goal is to build a tower of blocks, and when the bottom of the tower is not built yet, by firing this rule PRODIGY will not apply operators to build the top part of the tower.

```
(Control-Rule avoid-apply-for-wrong-goal
    (IF (and (on-goal-stack (on <x> <y>))
             (candidate-goal (on <y> <z>))
             (applicable-operator (pick-up <x>))))
    (THEN sub-goal))
```

5.3 Meta-Predicates

The left-hand side of the control rule is written in a subset of PDL, the same language as the preconditions of our operators and inference rules, though different predicates are used. Meta-level predicates such as CURRENT-GOAL and CANDIDATE-BINDINGS are used in control rules, whereas the predicates used in operators and inference rules are domain-level predicates, such as AT, ON-TABLE and HOLDING. A meta-predicate can be used as a generator of values for some of the variables it contains. For example, CANDIDATE-NODE returns bindings for <node> if <node> is unbound.

A stock set of meta-level predicates is provided with PRODIGY, but the user can also write additional ones as described in Section 5.4. Commonly used meta-predicates which are provided with PRODIGY4.0 are shown below.

- (CANDIDATE-NODE <node>): Tests whether <node> is among the set of open nodes in the tree. Can be used as a generator.
- (NEWEST-CANDIDATE-NODE <node>): Tests whether <node> is the most recently created candidate node. Can be used as a generator.
- (OLDEST-CANDIDATE-NODE <node>): Tests whether <node> is the oldest candidate node. Can be used as a generator.
- (CANDIDATE-GOAL <goal>): Test whether <goal> is a candidate goal for this node. Used when deciding on the current goal. Can be used as a generator for <goal>, which can be partially instantiated, e.g. (CANDIDATE-GOAL (clear <x>)) is legal and will generate values for <x>.
- (EXPANDED-GOAL <goal>): Tests whether <goal> is already expanded along the current path. Can be used to generate values for the variables in <goal>.

- (ON-GOAL-STACK <goal>): Tests whether the goal is on the goal stack - either expanded or a candidate. Can be used to generate values for the variables in <goal>.

- (CURRENT-GOAL <goal>): Tests whether <goal> is the current goal. Can be used as a generator. Can be used to generate values for the variables in <goal>.

- (CANDIDATE-OPERATOR <op>): Tests whether <op> is a member of the relevant operators being considered at the current node. Can be used as a generator.

- (CURRENT-OPERATOR <op>): Tests whether <op> is the operator chosen at the current node. Can be used as a generator.

- (CURRENT-OPS <ops>): Similar to CURRENT-OPERATOR, except that <ops> is a list of operators.

- (CANDIDATE-BINDINGS <bindings>): Tests whether <bindings> is a member of the candidate set of bindings for the current node, goal and operator. Can be used as a generator for <bindings>. <bindings> can be an association list of (operator-parameter . variable). Partial bindings are not allowed.

- (TRUE-IN-STATE <expr>): Tests whether <expr> is true in the state. <expr> can only be a literal, or negated literal, not any PDL expression. Can be used to generate values for the variables in <expr>, but cannot be used as a generator for nodes or expressions.

- (KNOWN <expr>): synonymous with TRUE-IN-STATE.

- (APPLICABLE-OPERATOR <inst-op>): Tests whether <inst-op> is an applicable operator. Can be used as a generator if <inst-op> is a variable or a list of the form (Operator-name <arg1> <arg2> ...)

- (TYPE-OF-OBJECT <obj> <type>): Tests whether <obj> is of type <type>.

5.4 Writing Your Own Meta-Predicates

A user-defined meta-predicate is created by defining a function in the USER package. The function can take nodes, goals, operators and/or bindings-lists as arguments, depending on the type of control rule it will be used in. There are no restrictions on the function, but the user should be familiar with the relevant aspects of PRODIGY4.0's internals. User-defined meta-predicates are usually put in file **functions.lisp** in the same directory as the domain. Make sure you load **functions.lisp** *before loading the domain*.

Here is an example of a user-defined meta-predicate taken from the multirobot domain. x1 and y1 are the x and y coordinates for location 1, respectively, and x2 and y2 are the coordinates for location 2. These two locations are the same if x1 and x2, as well as y1 and y2, are identical.

```
(defun same-loc (x1 y1 x2 y2)
  (and (= x1 x2) (= y1 y2)))
```

A meta-predicate may also be used as a generator of values for a variable (or several variables). For example, the following meta-predicate can generate all the instances of a type. If `obj` is bound, it just checks that `obj` is of type `type` by calling the predefined function `type-of-object`. If `obj` is unbound, it generates a list of bindings for `obj`. The function `all-the-instances` returns a list with all the instances of a given type.

```
(defun type-of-object-gen (obj type)
  (cond ((p4::strong-is-var-p type)
         (error "~A has to be bound.~%" type))
        ((p4::strong-is-var-p obj)
         ;;generate bindings: notice how binding list is built
         (mapcar #'(lambda (instance)
                         (list (cons obj instance)))
                 (all-the-instances type)))
        (t (type-of-object obj type))))
```

For example, if blockA and blockB are all the instances of type BLOCK in a problem, `(type-of--obj-gen <block> BLOCK)` returns `(((<block> . blockA))((<block> . blockB)))`. Note that the second argument, `type` has to be bound or the function will produce an error. [11]

6 Using PRODIGY

6.1 Introduction

Currently PRODIGY4.0 has a very simple interface. Domains and problems are created with a general purpose text editor and loaded into Lisp with functions. This section discusses some PRODIGY interface functions, shows how to control system behavior, displays a problem solution trace and describes the trace information.

6.2 Loading PRODIGY

There are two different ways to load PRODIGY. The first one is by loading the file **loader.lisp** from the **system** directory. The second alternative is by loading **prodigy.system** from that same directory and then typing `(load-prodigy)`. The first alternative is preferred because it is faster. The second one will try to compile the files as needed. Figure 12 shows how PRODIGY is loaded using **loader.lisp**. The variable `*WORLD-PATH*` contains the directory where the domains are. If you want to use your own domains, you should set `*WORLD-PATH*` to your domain directory.

6.3 Planner Commands

The following commands are Lisp functions that allow you to run PRODIGY and manipulate the problem solving environment. Each subsection describes briefly a function and its arguments. These functions can be used in the Lisp listener or included in Lisp routines.

6.3.1 `(domain '<domain-name>)`

Loads the file **domain.lisp** from the `<domain-name>` directory. The domain name is concatenated with the global variable `*WORLD-PATH*` to find the domain directory path. If no domain is given, the system will print out the list of domains found in `*WORLD-PATH*`. Once the file is loaded the function `(load-domain)` is called. `domain` has several keyword arguments:

[11] Meta-predicates have to return either true, false, or a list of bindings. A list of bindings contains one element for each possible way to instantiate the meta-predicate. And each of these elements contains a value for each unbound variable.

```
* (load "/afs/cs.cmu.edu/project/prodigy/version4.0/system/loader")

; Loading stuff from #<Stream for file
  "/afs/cs.cmu.edu/project/prodigy/version4.0/system/.ibm-rt/load-domain.fasl">.
; Loading stuff from #<Stream for file
  "/afs/cs.cmu.edu/project/prodigy/version4.0/system/.ibm-rt/types.fasl">.
; Loading stuff from #<Stream for file
  "/afs/cs.cmu.edu/project/prodigy/version4.0/system/.ibm-rt/assertions.fasl">.
...
; Loading stuff from #<Stream for file
  "/afs/cs.cmu.edu/project/prodigy/version4.0/system/.ibm-rt/hier.fasl">.

;;; Prodigy is loaded.
T
* *world-path*

"/afs/cs/project/prodigy/version4.0/domains/"
* (setf *world-path* "/usr/aperez/domains/")

"/usr/aperez/domains/"
```

Figure 12: Loading PRODIGY.

`:load-domain` (default `t`).
If this is nil, `(load-domain)` will not be called after loading the **domain.lisp** file.

`:compile` (default `nil`).
The matcher creates access functions to speed up the matching process. Compiling these functions increases planning speed even more, but may slow the process of loading the domain. If this argument is `nil`, the functions used in the matcher will not be compiled, the domain load will be faster, but the planner may run slower. This may be useful to solve lots of small problems from different domains. The best thing is to test it with your particular application.

`:function` (default `t`).
If this is true the **functions.lisp** file is loaded, if available.

These are some of the domains available in PRODIGY4.0:

blocksworld	-	the standard blocksworld domain
hanoi	-	Towers of Hanoi
logistics	-	transportation domain with package route planning
processp	-	a process planning domain
multirobot	-	multiagent planning domain
rocket	-	The Chinese Rocket problem; illustrates nonlinear planning
schedworld	-	a job shop scheduling domain
shopping	-	buying a sweater in Oakland, Pittsburgh
stripsworld	-	standard STRIPS domain
extended-strips	-	extension of the STRIPS domain
subway	-	includes the entire Pittsburgh subway!!

All these domains can be found in the **domains** directory of PRODIGY4.0's top directory.

6.3.2 `(load-domain)`

This function sets up a domain in order to solve problems. It installs operators in the problem space, builds functions for the matcher net, and does other useful things. You don't need to call `(load-domain)` if you loaded the domain with `(domain)`, but otherwise you do. `load-domain` returns the number of CPU seconds it takes to run `load-domain`. It takes one keyword argument:[12]

`:compile` (defaults to `t`)
same as for `domain` above.

6.3.3 `(problem '<problem-name>)`

This function loads a specific problem. It looks for the file `<*world-path*>/<domain-name>/probs/-<problem-name>.lisp`, after converting the name to lower case. If found, it loads it as a Lisp file. Note that problem files must be loaded *after* the domain has been set up, either by calling `domain` or `load-domain`.

6.3.4 `(run)`

This function calls the planner to solve a problem once a domain and a problem have been loaded. The values of its keyword arguments can be given for the current problem, or be the default values. Setting these keywords using `run` will set the switches in the current problem space. [13] `run` returns a list of values depending on the reason why the problem solver stopped. The `car` of the list returned represents an interrupt handler message. See Section 8 for more details. The keyword arguments are as follows:

`:current-problem` (value: object of type problem, defaults to `(current-problem)`)
specifies the problem to be run. Normally, `(current-problem)` will return the problem corresponding to the last problem file loaded, but it can be changed (with `setf`) to switch between two or more problems in memory.

`:search-default` (value: `depth-first` or `breadth-first`, defaults to the value of the corresponding problem space flag)
This can be used to switch between depth-first and breadth-first search strategies for problem solving.

`:depth-bound` (value: a number, defaults to the value of the corresponding problem space flag)
This controls the maximum depth of the search performed. If it is not specified it, the value is taken from the problem space, which is initialized to 30. The default value is too low for many problems. Some problems in the standard set need a depth of roughly 50 to be solved.

---

[12] There is another keyword argument, `:problem-space`, that defaults to the value of `*CURRENT-PROBLEM-SPACE*`. It can be used to switch between two or more problem spaces in memory, but is not implemented yet.

[13] Note the distinction between `run`'s keyword arguments (described in this section) and the problem space flags (Section 6.4).

`:max-nodes` (value: a number or `nil`, defaults to `nil`)
If this is a number, then it bounds the maximum number of nodes in the search tree that will be generated before stopping. `nil` means that there is no bound.

`:time-bound` (value: a number or `nil`, defaults to `nil`)
If this is a number, then it limits the maximum run time in seconds that will be allowed before the problem solving is stopped. `nil` means that there is no time bound.

`:output-level` (value: 0, 1, 2 or 3, defaults to 2)
This controls how much trace information is printed out during a run. If not specified as a keyword, the value is read from the current problem space. This value can be set by calling the function `(output-level <value>)`. The values are interpreted as follows:

0	-	Print nothing during problem solving.
1	-	Only print the resulting plan.
2	-	Print information about the nodes as they are expanded, and the resulting plan.
3	-	Print everything in 2 along with which control rules are fired.

`:compute-abstractions` (value: `t` or `nil`, defaults to `t`)
If this is true, when planning with abstraction, the abstraction hierarchy is computed automatically (see Section 9).

6.3.5 `(output-level <value>)`

This function sets the amount of information that will be printed out during problem solving. See the description of the `:output-level` flag of `(run)` above for more details.

6.4 Planner Flags

In addition to the above functions and options, there are some "problem space variables" that you can set. They will alter the way PRODIGY4.0 works, and become the default values when none are specified. [14] We have tried to provide reasonable defaults for these flags so that you do not need to bother with them at first. `run`'s keywords allow you to override these values for a particular problem. To alter the flags the function `pspace-prop` is provided as an accessor/modifier. For example, to find out if a search depth bound is explicitly set, type `(pspace-prop :depth-bound)`. To set a depth bound of 50, you would type `(setf (pspace-prop :depth-bound) 50)`. Using `pset` is sometimes easier: `(pset :depth-bound 50)` works just as well. The following flags are currently available in PRODIGY4.0:

`:linear` (values: `t` or `nil`, default is `nil`)
If this flag is not `nil`, PRODIGY4.0 will behave as a linear problem solver (no goal interleaving). This might make it faster in some cases, and might make it fail in others, depending largely on the problem being solved.

[14] These variables are stored as properties of the problem space so when the capability of switching between a number of problem spaces is fully implemented, you still will be able to have an individual setting for the flags. Note that a problem space is comprised of a domain definition and problem(s). See Section 8.1 for more details.

:depth-bound (values: nil or an integer, default is nil)
If this flag is a number, the search will be terminated if the search tree exceeds that depth. If it is nil, then the depth bound will be read from the problem space or lastly set to 30.

:search-default (values: :depth-first or :breadth-first, default is :depth-first)
Default search type for the problem space if not specified in a call to run.

:excise-loops (values: t or nil, default is t)
If this is not nil, dependency-directed backtracking will be used to try to cut down the search space. See Section 8 for more detail about dependency-directed backtracking.

:min-conspiracy-number (values: t or nil, default is t)
If this is not nil, when there is more than one alternative set of bindings to use for an operator, one will be chosen that leads to the smallest number of unsolved preconditions. [15]

:use-abs-level (values: t or nil, default is t)
If this is not nil, the abstraction hierarchy will be used as a preference ordering on goals even when separate abstraction levels are not being used. The hierarchies are always generated when you load the domain - this takes negligible time compared with compiling the matcher functions.

:random-behaviour (values: t or nil, default is nil)
If this is not nil, then when no more ordering control rules are left to choose between candidates at a decision point, one will be chosen randomly. Otherwise they will be chosen in left-right order.

:multiple-sols (values: t or nil, default is nil)
If this is not nil, once a solution is found, the planner will ask the user if search should continue looking for more solutions.

:print-alts (values: t or nil, default is nil)
If this is not nil, instead of printing out in the trace a number in square brackets to indicate how many alternatives are left, the alternatives themselves will be listed.

:print-old-way (values: t or nil, default is nil)
If this is not nil, instead of printing the trace as an indented tree (see Figure 13 for an example), each node is simply printed on a separate line as it is created. This provides fuller information, but is harder to take in at a glance. May be useful for debugging - that depends on personal taste.

6.5 The Trace Facility

The trace facility prints out a trace of PRODIGY solving a problem. The trace is useful for viewing the progress of problem solving for debugging purposes. The amount of information displayed is dependent on the output level selected. This section explains the trace when the output level is 3. Figure 13 shows a snapshot of the trace for the blocksworld problem presented in Section 2.

[15] This is a weakened version of "conspiracy search" [Elkan, 1989]. A stronger one, which might appear in the future, would be specified as a search type, replacing :depth-first.

```
* (domain 'paperblocksworld)
Remove existing problem space: #<p-space:  paper-blocksworld>?(y/n) y
Properties of 441 symbols removed.
Reading Meta predicate: on-goal-stack
Reading Meta predicate: candidate-goal
Reading Meta predicate: applicable-operator
Reading Meta predicate: current-goal
Reading Meta predicate: current-ops
Reading Meta predicate: true-in-state
Running load-domain.
0.1599998
* (problem 'suss)
T
* (run :output-level 3)
Creating objects (BLOCKA BLOCKB BLOCKC) of type OBJECT
  2 n2 (done)
  4 n4 <*finish*>
  5   n5 (on blocka blockb) [1]
  7   n7 <stack blocka blockb>
  8     n8 (holding blocka) [1]
 10     n10 <pick-up blocka>
 11       n11 (clear blocka) [1]
 12       n12 put-down ...goal loop with node 8
 12       n13 stack ...goal loop with node 8
 13       n15 <unstack blockc blocka> [2]
 14       n16 <UNSTACK BLOCKC BLOCKA> [1]
Firing apply/subgoal AVOID-APPLY-FOR-WRONG-GOAL deciding SUB-GOAL
 15   n17 (on blockb blockc)
 17   n19 <stack blockb blockc>
 18     n20 (clear blockc) [1]
 20     n22 <put-down blockc>
 21     n23 <PUT-DOWN BLOCKC> [1]
 22     n24 (holding blockb)
 24     n26 <pick-up blockb>
 25     n27 <PICK-UP BLOCKB>
 26   n28 <STACK BLOCKB BLOCKC> [1]
 27     n29 <PICK-UP BLOCKA>
 27   n30 <STACK BLOCKA BLOCKB>
Achieved top-level goals.

Solution:
<unstack blockc blocka>
<put-down blockc>
<pick-up blockb>
<stack blockb blockc>
<pick-up blocka>
<stack blocka blockb>

(((:STOP . :ACHIEVE)
  . #<APPLIED-OP-NODE 30 #<STACK [<OB> BLOCKA] [<UNDEROB> BLOCKB]>>)
 #<UNSTACK [<OB> BLOCKC] [<UNDEROB> BLOCKA]>
 #<PUT-DOWN [<OB> BLOCKC]> #<PICK-UP [<OB1> BLOCKB]>
 #<STACK [<OB> BLOCKB] [<UNDEROB> BLOCKC]> #<PICK-UP [<OB1> BLOCKA]>
 #<STACK [<OB> BLOCKA] [<UNDEROB> BLOCKB]>)
```

Figure 13: Example Blocksworld Trace.

Each line of the trace begins with a number that specifies the node's depth in the search tree and then the node identifier, beginning with `n`. The spacing between them reflects the depth of subgoaling from a user-specified goal to the current goal. The next piece of information is the goal or operator currently being worked on. Round brackets denote a goal — for example, `(on blocka blockb)`. Angle brackets denote an instantiated operator and its arguments — for example, `<pick-up blocka>`. If the instantiated operator is capitalized, this means that PRODIGY is applying it at the current node, otherwise it means that it has been chosen to achieve the goal mentioned in the line above it in the trace.

Normally, nodes for operators are not printed. However, if PRODIGY decides for some reason not to follow up on an chosen operator, it prints it out along with the reason. For example, in node 13 in the trace in Figure 13, PRODIGY is considering the operator STACK to achieve the goal `(clear blockA)`. However, every possible instantiation of STACK that could achieve this goal leads to a goal loop — that's why the operator node is printed along with the message `goal loop with node 8`.

Finally, the number in square brackets at the end of a line indicates how many alternatives there were to the choice that was made.

When the output level is 3, PRODIGY displays information about a control rule firing, including in some cases which alternatives where selected or rejected by the rule. In our example, a control rule fires at node `n16` to guide the decision between applying an operator and continue subgoaling.

When a solution exists for a problem, the solution steps are printed out. Finally the problem solver returns a list with the reason for the end of the search, the last node expanded, and a list of the operators that lead to the solution.

7 Debugging Techniques

This section contains suggestions on useful techniques for debugging a domain and problems runs. First we advise on good techniques to debug and develop a domain and then we suggest some available debugging techniques.

7.1 Debugging a Domain

After a domain is defined, the user should test each one of the operators by itself. To do this, create an initial state, where all the preconditons of the operator are true, and create different problems where you set as goals the different effects of the operator. Test both negative and positive effects, and also conditional effects if that is the case. After the problem run, check that the final state is the desired one, i.e. that all the effects, regular and conditional, changed the state as expected. As an example, suppose you want to write a domain for moving around by bus. There are buses, bus stops, and people, who ride, get on, and get off the buses. In Figure 14 (a) we show the operator to get on a bus. It says that if a person and a bus are at the bus stop, then the person gets on the bus. Notice that in this simple example we are not considering the destinations of the bus and of the people (as an exercise, you can try to extend this domain to handle destinations).

In Figure 14 (b) we show a wrong version of the operator to get off the bus. Let us see how to debug this operator. We can create a problem where a person is on a bus, and to test this operator we will set as a goal that the person is not on the bus anymore. In Figure 15 we show this problem, that can be solved as the effect `(del (on-bus <person> <bus>))` of operator GETOFF-BUS unifies with the negated goal with bindings `((<person> John) (<bus> bus-67F))`. The operator is applied succesfully as you can see in Figure 16. However the user may now notice by looking at the state that John is not at any place anymore, by calling the function `(show-state)` at the end of the problem solving episode, i.e. after the solution is

```
(OPERATOR GETON-BUS
   (params <person> <bus> <bstop>)
   (preconds
    ((<person> PERSON)
     (<bus> BUS)
     (<bstop> BUS-STOP))
    (and (at-person <person> <bstop>)
         (at-bus <bus> <bstop>)))
   (effects
    ()
    ((del (at-person <person> <bstop>))
     (add (on-bus <person> <bus>)))))

             (a)
```

```
(OPERATOR GETOFF-BUS
   (params <person> <bus>)
   (preconds
    ((<person> PERSON)
     (<bus> BUS))
    (on-bus <person> <bus>))
   (effects
    ()
    ((del (on-bus <person> <bus>)))))

             (b)
```

Figure 14: (a) Operator to Get On the Bus, (b) Operator to Get Off the Bus.

returned (bottom of Figure 16). This makes her realize that a location for the person has to be specified. The operator GETOFF-BUS should then be rewritten as shown in Figure 17.

```
(setf (current-problem)
      (create-problem
       (name pgh1)
       (objects
        (objects-are John Mary PERSON)
        (objects-are bus-67F bus-61A BUS)
        (objects-are Forbes Beeler Wilkins Murray BUS-STOP))
       (state
        (and (on-bus John bus-67F)
             (at-person Mary Forbes)
             (at-bus bus-67F Beeler)))
       (goal (~ (on-bus John bus-67F)))))
```

Figure 15: Problem 1: Testing the Operator GETOFF-BUS.

```
* (run)
Creating objects (JOHN MARY) of type PERSON
Creating objects (BUS-67F BUS-61A) of type BUS
Creating objects (FORBES BEELER WILKINS MURRAY) of type BUS-STOP

  2 n2 (done)
  4 n4 <*finish*>
  5   n5 not (on-bus john bus-67f)
  7   n7 <getoff-bus john bus-67f>
  7   n8 <GETOFF-BUS JOHN BUS-67F>
Achieved top-level goals.

Solution:
        <getoff-bus john bus-67f>

* (show-state)
(#<AT-BUS BUS-67F BEELER> #<AT-PERSON MARY FORBES>)
*
```

Figure 16: Trace for Solving Problem 1 with the Domain as Specified in Figure 14, and Inquire About the State of the World.

```
(OPERATOR GETOFF-BUS
   (params <person> <bus> <bstop>)
   (preconds
    ((<person> PERSON)
     (<bus> BUS)
     (<bstop> BUS-STOP))
   (and
    (on-bus <person> <bus>)
    (at-bus <bus> <bstop>)))
   (effects
    ()
    ((del (on-bus <person> <bus>))
     (add (at-person <person> <bstop>)))))
```

Figure 17: Operator to Get Off the Bus - Correct Version.

7.2 Other Debugging Techniques

There is not a nice analysis facility for PRODIGY4.0 yet, but we will present here some hints that can help you when debugging a problem solving episode. The trace facility is the most primitive technique, but it is useful for determining whether the system is on track or has become hopelessly lost. The easiest way to determine this is simply to keep an eye on the level of indentation, which tells you roughly how far down PRODIGY has subgoaled (see Section 6.5 for more on the meaning of indentation). You can also check periodically the goal PRODIGY is working on to see if it is trying to do something reasonable. If at any point you think that the planner is lost, you should stop the planner and try to analyze the search nodes you think are not correct.

The different output levels allow you to get more information on the trace as needed. In particular level 3 will display the control rules that fired with the alternatives that were selected, rejected, or preferred in each case. You can see all the alternatives at each node by setting `(pset :print-alts t)` before problem solving. This may give you an idea of why some decision is different from what you expect; it may be that the expected alternative was not even considered. For example, if the relevant effects of an operator are not correctly specified, the operator may not be considered relevant for a particular goal, when it should be.

To print the contents of a search node, you can do `(pshow 'node` *n*`)` where *n* is the node number. `pshow` can be used with other arguments to display information about other types of objects. Figure 18 shows the information available on node 5 of the example trace in Figure 16. Of course the information displayed depends on the type of node. In particular in this example the field `introducing-operators` indicates which operator this goal is a precondition of. [16]

The trace also indicates when the problem solver decided to abandon a search path, backtrack to the last node with some alternatives left, and try one of them. For example, in the trace in Figure 11, the current search path at node 35 fails because it leads to a state loop. PRODIGY has to backtrack up to node 28, the last node where there was some other alternative. PRODIGY first decided at that point to apply `<goto-obj box3 room4>`, but there was one other alternative (shown by the `[1]` at the end of the line). PRODIGY goes again to node 28 and tries the other alternative, to subgoal on `(dr-closed door34)`, expanding node 37. Backtracking is noted in the trace by both the break in the node number sequence, and by the decrease of the search tree depth (the first number in each of the trace lines).

[16] The top-level goals are considered preconditions of the `<*finish*>` operator.

```
* (pshow 'node 5)

#<GOAL-NODE 5 #<ON-BUS JOHN BUS-67F>> is a structure of type GOAL-NODE.
NAME: 5.
ABS-LEVEL: 0.
ABS-PARENT: NIL.
ABS-CHILDREN: NIL.
OPEN-NODE-LINK: NIL.
PARENT: #<BINDING-NODE 4 #<*FINISH*>>.
CHILDREN: (#<OPERATOR-NODE 6 #<OP: GETOFF-BUS>>).
PLIST: (:TERMINATION-REASON :EXHAUSTED :UNDER-DELETION NIL).
GOAL: #<ON-BUS JOHN BUS-67F>.
GOALS-LEFT: NIL.
APPLICABLE-OPS-LEFT: NIL.
INTRODUCING-OPERATORS: (#<BINDING-NODE 4 #<*FINISH*>>).
OPS-LEFT: NIL.
```

Figure 18: Description of Node 5 from the Trace in Figure 16.

8 Advanced Features

8.1 Problem Spaces

Problem spaces are a very general performance model in the context of problem solving. In PRODIGY a problem space consists of a domain theory, control knowledge for the domain, and a set of problems. The domain theory further consists of a set of operators and inference rules, and a type hierarchy for the objects in the domain. When a domain is defined, a problem space representing that domain is automatically defined. When PRODIGY4.0 is solving a problem in a problem space, all the information relevant to planning, such as the depth-bound, the output level, the current state of the problem, etc, is stored in a structure of type problem space.

PRODIGY can have more than one problem space around at a time and switch between problem spaces during planning. Thus it allows planning to take place in the middle of another plan. For instance, if an agent, while planning in a problem space, is missing a piece of information, it can plan in a different problem space to get that information, execute that plan, and continue planning in the previous problem space.

8.2 Interrupt Handling

Sometimes it is useful for the user to be able to call Lisp functions while PRODIGY4.0 is solving a problem. This would allow, for example, a graphical representation of the state to be updated while planning takes place, or would allow a controlling agent to interrupt the planner if the state were to change because of external events. The *signal mechanism* provides a way for the user to have functions called as often as once every time a new node is generated in the search space.

Figure 19 shows how the signal mechanism is used to implement the time-bound argument to `run`, and can be used as a template. The time bound indicates the maximum run-time allowed before the planner is stopped (see Section 6.3). A function called `time-check` is defined that takes one argument, a *signal*, which is defined below, and returns the list `(:stop :out-of-time)` if PRODIGY4.0 should halt when the function is called. `time-check` is declared to PRODIGY by `define-prod-handler`; the argument `:always` means that `time-check` should be called every time a new node is generated.

```
;;; This function stops Prodigy when time has run out.
(defun time-check (signal)
  (declare (ignore signal))
  (when (>= (get-internal-run-time) *stop-time*)
    ;; Print out a nice message and clear up (not shown)
    '(:stop :out-of-time)))

(define-prod-handler :always #'time-check)
```

Figure 19: A Handler Function to Stop PRODIGY4.0 when the Time Bound is Reached.

8.2.1 How it Works

How does the function defined in Figure 19 halt PRODIGY4.0? On every cycle of problem solving, PRODIGY4.0 matches a list of *signals* with a list of *signal handlers*. The signals (for example `always`) are small tagged pieces of data that may have been generated during the last cycle. The signal handlers (such as `time-check`) are functions that take a signal as an argument. A signal handler is associated with a signal, and whenever that signal is generated, PRODIGY4.0 will call the signal handler function with the signal as its argument. You can create a signal handler, as we did in Figure 19, by writing a function that takes a signal and then *registering* the function for a signal name. This is done with the function `define-prod-handler`. There may be more than one signal handlers registered for a signal. In the example, the function `time-check` is registered for the signal `always`. This means that it will be called at least once every cycle.

If a signal handler ever returns a list that begins with `:stop`, then PRODIGY4.0 stops, and includes in its return data the full list produced by the signal handler. Handlers can have useful side-effects even if they don't cause PRODIGY4.0 to stop.

8.2.2 Generating Signals

You may have several functions you would like to be called to handle different situations that arise while PRODIGY4.0 is planning. This can be done by generating your own signals and registering handlers for them. Suppose, for example, that a Lisp process is checking your calendar for events, and you want PRODIGY to stop when it is time for lunch. One way to do this would be to have the calendar checker generate a signal for every calendar event, of the type `calendar-event`, and to register a handler for these events, that tests whether the event is for lunchtime and if so stops PRODIGY. This is done in Figure 20. The `prod-signal` function generates a signal of a certain type, to which PRODIGY4.0 will respond within one problem solving cycle. Its first argument is the name, or type, of the signal — in this case `:calendar-event` — and its optional second argument is the signal data, in this case containing details about lunch.

Of course, you don't have to use `prod-signal` in order to have handler functions called. The reason that our first example (Figure 19) works is that PRODIGY4.0 behaves as if the call `(prod-signal :always)` was made on every cycle.

8.2.3 Summary of Functions for Signal Creation and Handling

`(prod-signal <SIGNAL-NAME> <SIGNAL-DATA>)` Registers a signal with the given name and data.

`(define-prod-handler <SIGNAL-NAME> <HANDLER>)` Registers a handler for a given signal name. The handler must be a function of one argument, which will be called once for every signal of the given

```
;;; this line appears in the calendar code:
(prod-signal :calendar-event '(lunch with Brian at the 0))

;;; The handler function
(defun stop-for-lunch (signal)
  (if (equal (car (second signal)) 'lunch)
      (list :stop signal)))

;;; This registers the handler function for the correct events.
(define-prod-handler :calendar-event #'stop-for-lunch)
```

Figure 20: Installing a Handler Function to Stop PRODIGY for Lunch.

name. Handlers are wiped clean by calls to `(load-domain)`. [17]

`(remove-prod-handler <SIGNAL-NAME> &optional <FUNCTION>)` If the optional `<FUNCTION>` is specified, removes that function from the list of handlers for the signal name. Otherwise, removes every handler for the signal name.

`(clear-prod-handlers)` Removes every handler for every signal name. This function should be used with caution, since some of PRODIGY4.0's normal behavior is implemented using the signal mechanism. However, since the handlers involved are re-defined on every call to `(run)`, it is safe to call this function before calling `(run)`.

8.3 Dependency-Directed Backtracking

As the search space explored by PRODIGY4.0 can be very large, a simple dependency-directed backtracking mechanism is implemented to prune parts of the search space. The main effect of dependency-directed backtracking in PRODIGY4.0 is to remove binding nodes from consideration when a necessary subgoal introduced by the node is later found to be unachievable. See [Blythe and Veloso, 1992] for a more thorough description and an example of dependency-directed backtracking, and other heuristics used by PRODIGY4.0 to reduce its search space.

9 Abstraction in PRODIGY

> This section discusses an advanced feature of PRODIGY4.0, and should only be read if you are already familiar with the basics of the system. The section can safely be skipped on a first reading of the manual.

As you have seen, PRODIGY is a powerful and general problem solving tool. All too frequently however, PRODIGY can get swamped in the details while solving a fairly complicated problem. Ideally, the planner should concentrate on the harder problems first, ignoring the details, and then add the needed detail to a general plan. This could save a lot of time, as well as probably lead to more intuitive plans from a user's point of view. This section describes PRODIGY's *abstraction* mechanism, which is designed to enable PRODIGY to do just that. Much of the section borrows heavily from [Knoblock, 1991].

[17] Handlers are local to problem spaces, so `def-prod-handler` must be called in every problem space where you want the handler to be defined.

There are many issues involved in using abstraction successfully in a planner, and we will not deal with them here, except to point out some potential problems which can be avoided by the user. For more information, look at [Nilsson, 1980, Rich and Knight, 1991] for an introduction, and [Knoblock, 1991] and [Blythe and Veloso, 1992] for more technical details about abstraction in PRODIGY.

PRODIGY can use an *abstraction hierarchy* over the literals in a domain, and it can generate one automatically given a particular domain and problem. It is relatively simple to get PRODIGY to work with its own abstraction hierarchies. If, in your domain, you know which are the right things to focus on first, you might want to try providing PRODIGY with a hierarchy, but you should also compare this with the ones generated automatically.

The next part of this chapter provides an introduction to abstraction and abstraction hierarchies in PRODIGY, using an example from the Tower of Hanoi domain. We then show how to use the abstraction module in PRODIGY4.0 with a trace that uses the same example.

9.1 Tower of Hanoi

The Tower of Hanoi puzzle requires moving a pile of various-sized disks from one peg to another with the use of an intermediate peg. The constraints are that only one disk at a time can be moved, a disk can only be moved if it is the top disk in a pile, and a larger disk cannot be placed on a smaller one. Figure 21 shows the initial and goal states of a three-disk Tower of Hanoi problem.

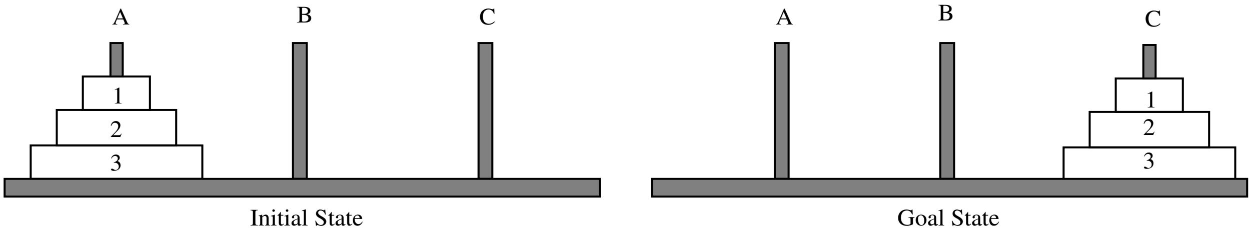


Figure 21: Initial and Goal States in the Tower of Hanoi.

In the simplified domain specification used for the Tower of Hanoi, each disk is a different type and is moved by a separate operator. Figure 22 shows the operator for moving disk 2 from one peg to another.

```
(pinstance-of disk2 DISK2)

(OPERATOR Move-disk2
 (params <from> <to>)
 (preconds ((<from> Peg)
            (<to> Peg))
  (and (on disk2 <from>)
       (~ (on disk1 <from>))
       (~ (on disk1 <to>))))
 (effects
  ()
  ((del (on disk2 <from>))
   (add (on disk2 <to>)))))
```

Figure 22: Operator for Moving Disk2 in the Tower of Hanoi.

9.2 Abstraction in Planning

Abstraction is a technique used to focus the problem solver on the more important aspects of the problem first, so it will only deal with other aspects once those are solved. In PRODIGY, as in many other systems, we do this by forming a *reduced model* of the problem space, solving first a problem in the reduced model, and then *refining* the plan in the reduced model into a plan in the ground space. The ground space contains the complete specification of the problem space. A reduced model is formed by removing literals from the operators. For example, in the Tower of Hanoi domain, one might remove every instance of `(on disk1 <peg>)` from the domain to simplify the 3-peg problem to the 2-peg problem.

The process of forming a reduced model can be repeated a number of times to form an *abstraction hierarchy*, in which more literals are removed the higher one climbs the hierarchy. PRODIGY can form the hierarchy itself — that is, it can decide which predicates to leave out from the model at any stage — or it can work with a hierarchy defined by the user.

We describe planning with abstraction in detail for the case where there are only two spaces, namely the ground space and the abstract space. A full hierarchy can be treated simply as a sequence of these cases. The first step is to map the original problem, which is defined in the ground space, into the abstract space. This is done by dropping from both the initial state and goal description every literal that is dropped in the abstraction.

The second step is to solve the planning problem in the abstract space, by the usual planning procedure. Now, however, any reference to the dropped literals in the operators is ignored. This speeds up planning by reducing the number of goals to consider, but may yield a plan that is incorrect at the ground space, for instance a plan which tries to move `disk2` even though `disk1` is on top of it.

Finally, the abstract plan is *refined* to produce a plan in the ground space, if possible. PRODIGY uses the abstract plan as a rough guide for searching in the ground space, but it allows the new subgoals that appear when it adds back all the dropped predicates to be addressed in any order, to maintain the completeness of PRODIGY4.0's nonlinear search. More details can be found in [Blythe and Veloso, 1992].

In some cases, which are precisely defined in [Knoblock, 1991], hierarchical problem solving of this kind can reduce the size of the search space from exponential to linear in the size of the solution. In other cases it can slow it down. For this reason, it is very important to choose a good abstraction hierarchy for abstract planning. When PRODIGY computes a hierarchy for a problem, it tries to guarantee that plans in the abstract space will be refineable, although it does not always succeed.

9.3 Working with Abstraction in PRODIGY4.0

The functions for running PRODIGY without using abstraction, as described in Section 6, are also used for running the system with abstraction. Two more functions are used, namely to run PRODIGY with an abstraction hierarchy and to load an abstraction hierarchy.

The function `(abs-level)` describes the level of the abstraction hierarchy at which PRODIGY will begin to form a plan. This is usually set to either `0` or `nil`. A setting of `0`, which is the default, indicates the ground space, where no abstractions are considered. A setting of `nil` indicates that the system should choose the highest possible abstraction level to begin planning. Whenever PRODIGY begins planning with an abstraction level greater than zero, it will refine a successful plan rather than just return it.

PRODIGY uses *literal type signatures* to represent the literals at an abstraction level. These consist of a predicate name and a type for each argument of the predicate, for example `(holding block)`. If a type hierarchy exists, all the types must be of the lowest possible generality, i.e. they must correspond to leaves in the type hierarchy.

By default, PRODIGY generates its own abstraction hierarchies to use when planning with abstraction, but the function `load-hierarchy` can be used to specify a different abstraction hierarchy. This function takes a list of lists of literal type signatures, with the intention that the first list defines the literals dropped at the first level of abstraction, the second list those dropped at the second level of abstraction, and so on. In addition, if you wish to use your own abstraction hierarchy, you must supply each call to the function `run` with the keyword-value pair (`:compute-abstractions nil`) to prevent PRODIGY from computing a hierarchy and replacing yours with it.

These function `print-hierarchy` can be used to inspect the hierarchies generated. It prints out the predicate type signatures associated with each level in descending order.

9.4 An Example

The example shown here corresponds the Tower of Hanoi problem with 3 disks. Without abstraction or control rules, this problem is not solved within 5,000 nodes. When PRODIGY is running with abstraction, the node labels in the trace specify the abstraction level before the node number (0 corresponds to the ground space). In this trace the initial segment of each refinement has been omitted, since it is identical to the one for more abstract plan. The line `...  ;; refining <parent node>...` has been added to show the join.

```
<cl> (setf (abs-level) nil)
NIL
<cl> (run :depth-bound 50 :compute-abstractions t)
Creating objects (PEG1 PEG2 PEG3) of type PEG

  2 n2-2 (done)
  4 n2-4 <*finish*>
  5   n2-5 (on disk3 peg2)
  6   n2-6 move-disk1 ...no choices for bindings
  6   n2-7 move-disk2 ...no choices for bindings
  7   n2-9 <move-disk3 peg1 peg2> [1]
  7   n2-10 <MOVE-DISK3 PEG1 PEG2>
Achieved top-level goals. (Abstraction level 2)

... ;; refining n2-9...
  6   n1-9 <move-disk3 peg1 peg2>
  7     n1-11 not (on disk2 peg1) [1]
  8     n1-12 move-disk1 ...no choices for bindings
  9     n1-14 <move-disk2 peg1 peg3> [1]
 10     n1-15 <MOVE-DISK2 PEG1 PEG3> [1]
 11   n1-10 <MOVE-DISK3 PEG1 PEG2> [1]
 12   n1-16 (on disk2 peg2)
 13   n1-17 move-disk1 ...no choices for bindings
 14   n1-19 <move-disk2 peg3 peg2> [1]
 14   n1-20 <MOVE-DISK2 PEG3 PEG2>
Achieved top-level goals. (Abstraction level 1)

... ;; refining n1-14...
  9     n0-14 <move-disk2 peg1 peg3>
 10       n0-21 not (on disk1 peg1) [1]
 12       n0-23 <move-disk1 peg1 peg3> [1]
 13       n0-24 <MOVE-DISK1 PEG1 PEG3> [1]
 14       n0-25 not (on disk1 peg3) [1]
 16       n0-27 <move-disk1 peg3 peg2> [1]
 17       n0-28 <MOVE-DISK1 PEG3 PEG2> [1]
 18     n0-15 <MOVE-DISK2 PEG1 PEG3> [1]
 19     n0-29 not (on disk1 peg2)
 21     n0-31 <move-disk1 peg2 peg3> [1]
```

```
 22     n0-32 <MOVE-DISK1 PEG2 PEG3>
 23   n0-10 <MOVE-DISK3 PEG1 PEG2> [1]
 24   n0-16 (on disk2 peg2) [1]

    ;; refining n1-19...
 26   n0-19 <move-disk2 peg3 peg2>
 27     n0-33 not (on disk1 peg3) [1]
 29     n0-35 <move-disk1 peg3 peg2> [1]
 30     n0-36 <MOVE-DISK1 PEG3 PEG2> [1]
 31     n0-37 not (on disk1 peg2)
 33     n0-39 <move-disk1 peg2 peg3> [1] ...applying leads to state loop.
 33     n0-41 <move-disk1 peg2 peg1>
 34     n0-42 <MOVE-DISK1 PEG2 PEG1>
 35   n0-20 <MOVE-DISK2 PEG3 PEG2> [1]
 36   n0-43 (on disk1 peg2)
 38   n0-45 <move-disk1 peg1 peg2> [1]
 38   n0-46 <MOVE-DISK1 PEG1 PEG2>
Achieved top-level goals.

Solution:
       <move-disk1 peg1 peg3>
       <move-disk1 peg3 peg2>
       <move-disk2 peg1 peg3>
       <move-disk1 peg2 peg3>
       <move-disk3 peg1 peg2>
       <move-disk1 peg3 peg2>
       <move-disk1 peg2 peg1>
       <move-disk2 peg3 peg2>
       <move-disk1 peg1 peg2>
<cl> (print-hierarchy)
 3:
   [DONE]
 2:
   [ON DISK3 PEG]
 1:
   [ON DISK2 PEG]
 0:
   [ON DISK1 PEG]

NIL
<cl> (load-hierarchy '(((on disk1 peg) (on disk2 peg))
                       ((on disk3 peg)(p4::done))))

;;; returns the abstraction hierarchy. Only two abstraction levels.

<cl> (run :depth-bound 50 :compute-abstractions nil)

;;; Now it takes 274 nodes to complete.
```

10 The Prodigy System

This section describes the organization of the files, the components of a domain, and the implementation dependencies of the system.

10.1 The Organization of the Files

All the files containing PRODIGY and the domain are depicted in Figure 23. The top-level PRODIGY directory contains two subdirectories, **domains** and **system**. If you are using PRODIGY at CMU, you will probably find the top level directory at `/afs/cs/project/prodigy/version4.0`. It will contain a few other files and directories, which aren't necessary for the system. Otherwise, the top level directory should be the value of the variable `*PRODIGY-ROOT-DIRECTORY*`.

The directory **system** contains several directories for the different modules of the system: planner, abstraction, ... [18] The system is defined in a file called **prodigy.system** and the directory **planner** contains all the files that compose the planner system. They are loaded either by loading the file **loader.lisp**, or alternatively by loading **prodigy.system** and then typing `(load-prodigy)`. All the domains are under the directory **domains**. The variable `*WORLD-PATH*` contains the name of this directory. The Blocksworld domain, for example, is one of its subdirectories. The subdirectory **probs** contains a file for each problem in the domain.

The files that form the planner in PRODIGY are described in appendix C. The contents of the files related to a domain are described below. Of course, you don't need to put the PRODIGY system under the same directory as the domains. You can have the domains wherever you like, maybe in your own personal directory. There is a way to tell the system where the domain you want to use is. We talk about it below.

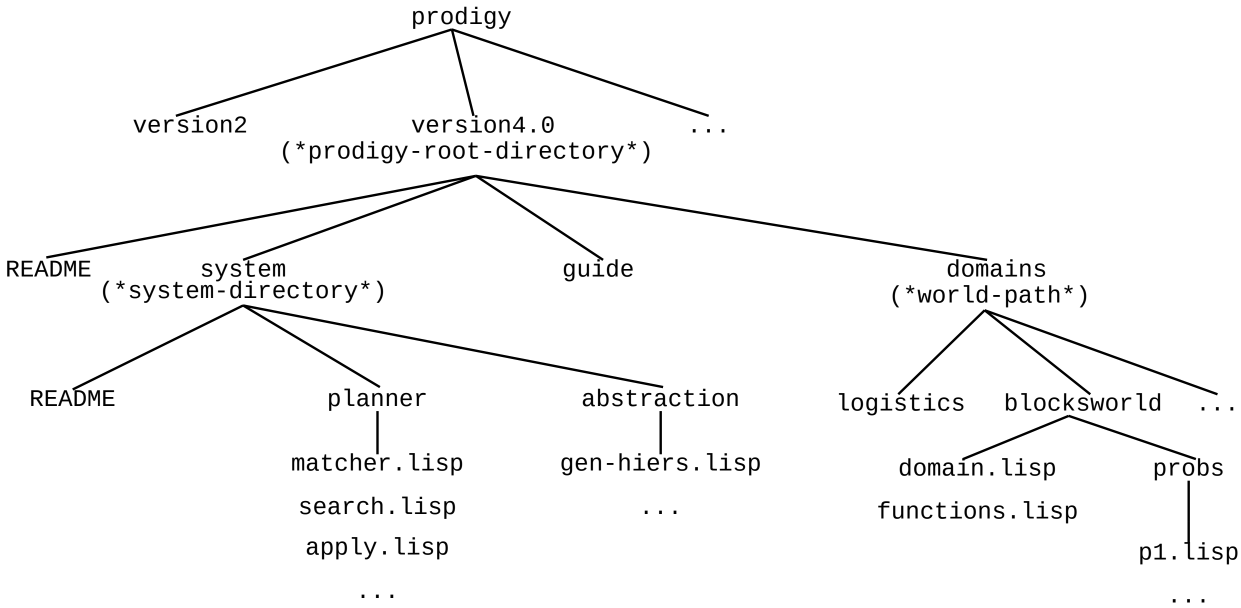


Figure 23: The Files That Contain PRODIGY and the Domains.

10.2 The Components of a Domain

As we saw before, the files that describe the domain are in the directory that corresponds to that domain. All the operators, inference rules and control rules of the domain are included in the file **domain.lisp** as follows (an example of such a file is included in Appendix C):

[18] Currently the directories available are **planner** and **abstraction**. Further modules, such as different learning methods, case-based reasoning, and knowledge acquisition, will be added as directories as the modules are added to the system.

```
(create-problem-space 'DOMAIN-NAME :current t)

(ptype-of TYPE1 :top-type)
...
(ptype-of TYPEm :top-type)

(Operator OPERATOR1 ...)
...
(Operator OPERATORn ...)

(Inference-Rule INFERENCE-RULE1 ...)
...
(Inference-Rule INFERENCE-RULEp ...)

(Control-Rule CTRL-RULE1 ...)
...
(Control-Rule CTRL-RULEq ...)
```

The file **functions.lisp** should contain all the functions you defined as follows:

```
(defun function1 ...)
...
(defun functionf ...)
```

The easiest way to load a domain in PRODIGY4.0 is using the function `(domain '<domain-name>)`. It looks for a directory with the name you specified as a subdirectory of `*WORLD-PATH*`. However one can also load the domain files just as normal Lisp files and call `(load-domain)` afterwards. Note that this function should be called after any changes to a domain.

In summary, `(domain '<domain-name>)` will load the following files:

- **domain.lisp**, that should contain all the operators, inference rules and control rules of your domain.
- **functions.lisp**, if it exists, that should contain all the functions and user-defined meta-predicates that your operators, inference rules, and control rules use.

Besides all the above files, there is a subdirectory called **probs** (see Figure 23), that should contain all the problems of your domain. Each problem should be in a file containing:

```
(setf (current-problem)
      (create-problem
       (name example)
       (objects
        (instance1 instance2 type1)
        ...)
       (state
        (and assertion1 ... assertionn)
        )
       (goal (and goal-assertion1 ...)
        )))
```

Section 3.3 gives an example of the description of a problem. If you type `(problem 'example)` to Lisp, it will look for the file **example.lisp** in the **probs** directory, and load it. You can also just load the file yourself using the Common Lisp `load` function. It doesn't matter. Note that problem files must be loaded *after* the domain has been set up by calling either `domain` or `load-domain`.

10.3 The Implementation

PRODIGY is written in Common Lisp. PRODIGY is not a big space hog, but Common Lisp tends to be, so you should have plenty of megabytes on your machine. Currently, the system is run in CMU Common Lisp on the IBM-RT and in Allegro Common Lisp on the PMaxes, Sun workstations, and the Macintosh. We have tried to keep the code as system independent as possible.

You should be able to run PRODIGY in another version of Common Lisp on a different machine with only a few changes. The **system** directory contains two files for loading PRODIGY: **prodigy.system** which uses the defsystem package and **loader.lisp** which does not. The directory structure will vary from machine to machine, so you will need to change the variables that refer to directories, namely `*PRODIGY-ROOT-DIRECTORY*` (the top-level directory), `*SYSTEM-DIRECTORY*` (the planner directory) and `*WORLD-PATH*` (the directory containing the domains).

The PRODIGY code is available for distribution. Write to the following address or send electronic mail to `prodigy@cs.cmu.edu` to inquire about release forms and conditions:

```
The PRODIGY Project
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA  15213-3890
```

The PRODIGY system is advanced experimental software, furnished on an 'as is' basis. Carnegie Mellon and the authors make no warranties of any kind regarding the software or its documentation, and shall not be liable for direct or consequential damages resulting from the use of the software.

11 Answers To Some Frequently Asked Questions

Q. *How do I start writing a domain in* PRODIGY*?*
A. Below are seven suggested steps for creating a domain in PRODIGY:

1. Outline the problems you are trying to solve and solutions to the problems.

2. Create the primitives for the problem domain. These primitives are the physical objects in the system (i.e., box, object, etc.) or the properties of these objects (i.e., location, position, etc.).

3. Formulate several example problems and a step by step description of their solution in terms of the primitives above. It might help to think about how you would teach a human apprentice this expertise.

4. Create the rules for each possible operation or deduction. These rules are the system's methods for changing state (stack, unstack, etc.) or making inferences (clear, etc.).

5. Test each individual rule on a trivial problem. This will insure that your individual rules work as you expected.

6. Run more complex problems in the domain. This will test the interaction between the rules.

7. Add control knowledge to the system. Try to determine what makes one path preferred over another and encode this information in the control rules.

Q. *When should I use functions?*
A. Functions are used to denote facts about the world that never change. They are especially useful for information that can be computed more easily than the information can be stored (e.g. it is easier to check that one number is smaller than other number by using a function than to store in the state a LESS-THAN literal for every pair of numbers). (See Section 3.4.2.)

Q. *What do functions return?*
A. Functions return `t`, `nil`. In addition, functions used as generators for infinite type variables should return a list of values. (See Sections 3.4.2 and 3.4.5.)

Q. *What does control knowledge allow me to do?*
A. Control knowledge allows me to 1) improve the efficiency of the problem solver and 2) improve the quality of the solutions produced. (See Section 5.)

Q. *How do I know where and when to add control knowledge to the system?*
A. To determine where to add control knowledge can be a difficult problem. Some hints are listed below to help with this task:

1. Look at a trace of the system solving a problem and try to detect a position where a bad goal, operator or binding selection was made. Determine why that selection was made and write a general rule that will make the correct selection.

2. Analyze the system's final solution. If there is a better solution then try and determine how to improve the solution using control knowledge. (See Section 5.)

Q. *What if I add control knowledge and nothing changes?*
A. First make sure you have saved your control rule changes in the file **domain.lisp**. Then *make sure you have reloaded the domain*, either by calling the `domain` function, or loading the **domain** file and then typing (`load-domain`) to Lisp. This step is often left out by novices. Then you can see which control rules fired by setting the output-level to 3 (call `(output-level 3)`). And you may trace the meta-predicates to see why the control rules did not fire. Sometimes just checking the spelling of the predicates in the rule will solve your problems: you may spend some time trying to figure out why `(current-goal (inroom robot <room>))` did not match when the goal was `(in-room robot rm1)`.

Q. *What if the system's problem-solving behavior does not appear to make sense?*
A. Find the point in the trace where the system's behavior diverged from what was expected. Try to trace the problem to a particular operator (Section 7). Carefully check that operator's preconditions and effects.

Q. *To whom do I send mail about bugs, problems, praises, etc.?*
A. Mail correspondences to `prodigy@cs.cmu.edu`. For mail about bug reports please include the following information:

1. The system version number.
2. Enough information to reproduce the problem. This includes:
   (a) the domain file.
   (b) the functions file.
   (c) a detailed description of the problem including the error message.
   (d) a procedure for reproducing the bug (the problem PRODIGY was solving, the flags that were active, ...).
3. How to get in touch with you (electronic mail address and phone number).

12 Acknowledgements

We would like to thank all the other members of the PRODIGY project along the years for many helpful and insightful discussions, namely Dan Kuokka, Ellen Riloff, Henrik Nordin, Hiroshi Tsuji, Daniel Borrajo, Michael Miller, Santiago Rementería, Nobuyoshi Wada, Yan-Bin Jia, José Brustoloni, David Rager, and Masahiko Iwamoto. Thanks also to David Long for his help with LaTeX in the edition of this manual.

References

[Blythe and Veloso, 1992] J. Blythe and M. Veloso. An analysis of search techniques for a totally-ordered nonlinear planner. In *Proceedings of the First International Conference on AI Planning Systems*, College Park, MD, June 1992.

[Carbonell and Gil, 1990] J. G. Carbonell and Y. Gil. Learning by experimentation: The operator refinement method. In *Machine Learning: An Artificial Intelligence Approach, Volume III*. Morgan Kaufmann, San Mateo, CA, 1990.

[Carbonell and Veloso, 1988] J.G. Carbonell and M.M. Veloso. Integrating derivational analogy into a general problem-solving architecture. In *Proceedings of the First Workshop on Case-Based Reasoning*, Tampa, FL, May 1988. Morgan Kaufmann.

[Carbonell *et al.*, 1990a] J. G. Carbonell, Y. Gil, R. Joseph, C. A. Knoblock, S. Minton, and M. M. Veloso. Designing an integrated architecture: The PRODIGY view. In *Proceedings of the Twelfth Annual Conference of the Cognitive Science Society*, 1990.

[Carbonell *et al.*, 1990b] J. G. Carbonell, C. A. Knoblock, and S. Minton. Prodigy: An integrated architecture for planning and learning. In Kurt VanLehn, editor, *Architectures for Intelligence*. Erlbaum, Hillsdale, NJ, 1990. Also Technical Report CMU-CS-89-189.

[Elkan, 1989] C. Elkan. Conspiracy numbers and caching for searching and/or trees and theorem proving. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, MI, 1989.

[Etzioni, 1990] O. Etzioni. *A Structural Theory of Explanation-Based Learning*. PhD thesis, Carnegie Mellon University, School of Computer Science, 1990. Also appeared as Technical Report CMU-CS-90-185.

[Fikes and Nilsson, 1971] R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2, 1971.

[Fikes *et al.*, 1972] R. Fikes, P. Hart, and N. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3(4), 1972.

[Joseph, 1989] R.L. Joseph. Graphical knowledge acquisition. In *Proceedings of the 4th Knowledge Acquisition For Knowledge-Based Systems Workshop*, Banff, Canada, 1989.

[Knoblock, 1990] C.A. Knoblock. Learning effective abstraction hierarchies. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, Boston, MA, 1990.

[Knoblock, 1991] C.A. Knoblock. *Automatically Generating Abstractions for Problem Solving*. PhD thesis, Carnegie Mellon University, School of Computer Science, 1991. Also appeared as Technical Report CMU-CS-91-120.

[Minton *et al.*, 1989a] S. Minton, J. G. Carbonell, C. A. Knoblock, D. R. Kuokka, O. Etzioni, and Y. Gil. Explanation-based learning: A problem-solving perspective. *Artificial Intelligence*, 40:63–118, 1989. Available as technical report CMU-CS-89-103.

[Minton *et al.*, 1989b] Steven Minton, Craig A. Knoblock, Daniel R. Kuokka, Yolanda Gil, Robert L. Joseph, and Jaime G. Carbonell. PRODIGY2.0: The manual and tutorial. Technical Report CMU-CS-89-146, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1989.

[Minton, 1988] S. Minton. *Learning Effective Search Control Knowledge: An Explanation-based Approach*. PhD thesis, Carnegie Mellon University, School of Computer Science, 1988. Also appeared as Technical Report CMU-CS-88-133.

[Mitchell *et al.*, 1990] T. M. Mitchell, J. Allen, P. Chalasani, J. Cheng, O. Etzioni, M. Ringuette, and J. C. Schlimmer. Theo: A framework for self-improving systems. In Kurt VanLehn, editor, *Architectures for Intelligence*. Erlbaum, Hillsdale, NJ, 1990.

[Nilsson, 1980] N.J. Nilsson. *Principles of Artificial Intelligence*. Tioga Publishing Co., 1980.

[Pérez and Etzioni, 1992] M. Alicia Pérez and Oren Etzioni. DYNAMIC: A new role for training problems in EBL. In D. Sleeman and P. Edwards, editors, *Machine Learning: Proceedings of the Ninth International Conference (ML92)*. Morgan Kaufmann, San Mateo, CA., 1992. Extended version appeared as Technical Report CMU-CS-92-124.

[Rich and Knight, 1991] E. Rich and K. Knight. *Artificial Intelligence*. Mc Graw Hill, New York, second edition, 1991.

[Rosenbloom *et al.*, 1990] P. S. Rosenbloom, A. Newell, and J. E. Laird. Towards the knowledge level in SOAR: The role of the architecture in the use of knowledge. In Kurt VanLehn, editor, *Architectures for Intelligence*. Erlbaum, Hillsdale, NJ, 1990.

[Thompson and Langley, 1989] Thompson and P. Langley. An architecture for a learning autonomous agent. In *Proceedings of European Workshop on Representation and Learning in an autonomous agent*, 1989.

[Veloso and Carbonell, 1989] M.M. Veloso and J.G. Carbonell. Learning analogies by analogy - the closed loop of memory organization and problem solving. In *Proceedings of the Second Workshop on Case-Based Reasoning*, Pensacola, FL, May 1989. Morgan Kaufmann.

[Veloso, 1989] M. Veloso. Nonlinear problem solving using intelligent casual-commitment. Technical Report CMU-CS-89-210, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1989.

[Wang, 1992] Xuemei Wang. Constraint-based efficient matching in PRODIGY. Technical Report CMU-CS-92-128, School of Computer Science, Carnegie Mellon University, April 1992.

A PRODIGY as an Integrated Architecture

Artificial intelligence has progressed to the point where multiple cognitive capabilities are being integrated into computational architectures, such as SOAR [Rosenbloom *et al.*, 1990], PRODIGY [Carbonell *et al.*, 1990b], THEO [Mitchell *et al.*, 1990], and ICARUS [Thompson and Langley, 1989]. Our goal is to build a general purpose learning and reasoning system that given basic axiomatic knowledge of a domain is capable of becoming an expert problem solver. Our machine learning approach starts with a general problem-solving engine based on a possibly incomplete domain theory. The problem solver improves its performance through experience by refining the initial domain knowledge and learning knowledge to control the search process. Figure 24 presents a diagram of the PRODIGY architecture including the problem solver and the different learning components. This manual describes the basic problem solver. For further references on PRODIGY we refer the reader to [Minton *et al.*, 1989a, Carbonell *et al.*, 1990a, Carbonell *et al.*, 1990b]. In this section we give an overview of the learning components: [19]

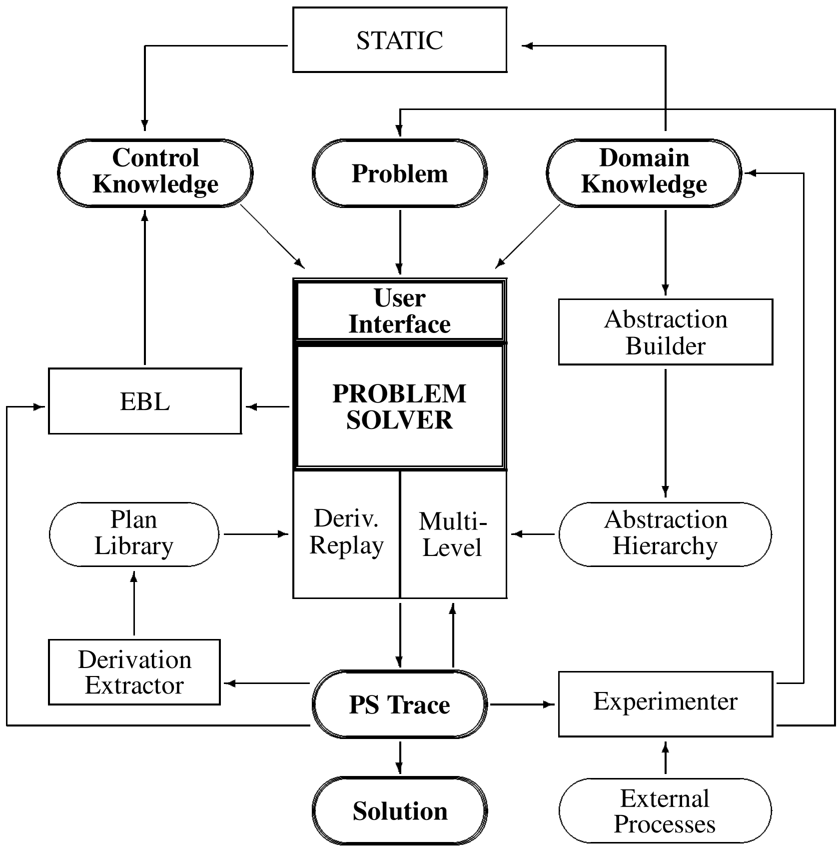


Figure 24: The PRODIGY Architecture.

[19]The modules described here have been implemented and tested in previous versions of PRODIGY. Only the abstraction generation and planning module is currently available in PRODIGY4.0.

- **ALPINE**: An abstraction hierarchy generation module [Knoblock, 1990, Knoblock, 1991]. The axiomatized domain knowledge is divided into multiple abstraction levels based on an analysis of the domain. Then during problem solving, PRODIGY first finds a solution to guide the search for solutions in more detailed problem spaces. The method for generating abstraction hierarchies was developed by Knoblock, but the method for planning with abstractions is different from the one used in previous versions of PRODIGY. See Section 9 for a more detailed description of this module.

- **APPRENTICE**: A graphical user interface that can participate in an apprentice-like dialogue, and can both acquire domain knowledge or accept advice during problem solving [Joseph, 1989].

- **EBL**: An explanation-based learning facility [Minton, 1988] for acquiring control rules from a problem-solving trace. Explanations are constructed from an axiomatized theory describing both the domain and relevant aspects of the problem solver's architecture. Then the resulting descriptions are expressed in control rule form, and control rules whose utility in search reduction outweighs their application overhead are retained.

- **STATIC**: A method for learning control rules by analyzing PRODIGY's domain descriptions prior to problem solving. The STATIC program [Etzioni, 1990] produces control rules without utilizing any training examples. STATIC matches EBL's performance on some domains and exhibits one to two orders of magnitude faster learning rate. However, not all problem spaces permit purely static learning, requiring EBL to learn control rules dynamically.

- **DYNAMIC**: A method for learning control rules by analyzing both the problem space description and a problem-solving trace [Pérez and Etzioni, 1992]. DYNAMIC integrates the use of examples to pinpoint learning opportunities with static analysis of the problem space to generate control rules.

- **ANALOGY**: A derivational analogy engine [Carbonell and Veloso, 1988, Veloso and Carbonell, 1989] that uses similar previously solved problems to solve new problems. The problem solver records the justifications for each decision during its search process. These justifications are then used to guide the reconstruction of the solution for subsequent problem solving situations where equivalent justifications hold true. Analogy, EBL, and static analysis are independent mechanisms to acquire domain-specific control knowledge.

- **EXPERIMENT**: A learning-by-experimentation module for refining domain knowledge that is incompletely or incorrectly specified [Carbonell and Gil, 1990]. Experimentation is triggered when plan execution monitoring detects a divergence between internal expectations and external observations. The main focus of experimentation is to refine the factual domain knowledge, rather than the domain knowledge.

B PRODIGY4.0's Description Language (PDL4.0)

rule	=	*operator* | *inference-rule* | *control-rule*
operator	=	(**OPERATOR** *rule-name*
		(**params** *params-list*)
		(**preconds** (*var-spec**) *pdl-expression*)
		(**effects** *var-descriptor** *effects*))
inference-rule	=	(**INFERENCE-RULE** *rule-name*
		mode
		(**params** *params-list*)
		(**preconds** (*var-spec**) *pdl-expression*)
		(**effects** *var-descriptor** *effects*))
params-list	=	*var-name**
var-spec	=	(*var-name var-descriptor*)
var-descriptor	=	(**AND** *var-descriptor*$^+$ *constraint**) | (**OR** *var-descriptor*$^+$ *constraint**) |
		($\sim$ *var-descriptor*$^+$ *constraint**) | *type-name*
constraint	=	*var-descriptor* | **lisp-function**
pdl-expression	=	(**AND** *pdl-expression*$^+$) | (**OR** *pdl-expression*$^+$) | ($\sim$ *pdl-expression*$^+$) |
		(**FORALL** *var-descriptor pdl-expression*$^+$) |
		(**EXISTS** *var-descriptor pdl-expression*$^+$) | *predicate*
effects	=	(*effect*$^+$)
effect	=	(**ADD** *predicate*) | (**DEL** *predicate*) | *conditional-effect*
conditional-effect	=	(**IF** *pdl-expression effects*)
mode	=	**(mode eager)** | **(mode lazy)** | ϵ
predicate	=	**literal**
rule-name	=	**id-name**
type-name	=	**id-name**
instance	=	**id-name**
var-name	=	**<id-name>**
control-rule	=	(**CONTROL-RULE** *rule-name*
		(**if** *restricted-pdl-exp*)
		(**then** *rhs-specification*))
restricted-pdl-exp	=	see Section 5.1
rhs-specification	=	*select-spec* | *reject-spec* | *prefer-spec* | *SUBGOAL* | *APPLY*
select-spec	=	**SELECT** *candidate-type value*
reject-spec	=	**REJECT** *candidate-type value*
prefer-spec	=	**PREFER** *candidate-type preferred-value preferred-over*
candidate-type	=	**NODE** | **GOAL** | **OPERATOR** | **BINDINGS**
start-state	=	(**AND** *literal*$^+$)
goal-expression	=	*var-spec pdl-expression* | *pdl-expression*

C The Extended-STRIPS domain

```
;;; ******************************************
;;; File: domain.lisp
;;; Adapted from the Prodigy2.0 domain,
;;; 1/25/92 aperez

;;; Notes about this version:
;;; -no locations (i.e. no "at" predicate,
;;;  no numbers)
;;; -if something is picked up by the robot,
;;;  it is not next-to anything
;;;  (i.e. (holding x) --> (~(next-to x y))
;;;  (see  PICKUP-OBJ)
;;; -the robot cannot be holding two things
;;;  at a time:see PICKUP-OBJ
;;; -next-to not transitive: see PUTDOWN
;;; -next-to symmetric: see PUTDOWN-NEXT-TO,
;;;  PUSH-BOX
;;; -only boxes are pushable.

(create-problem-space 'extended-strips
                      :current t)

(ptype-of OBJECT :top-type)
(ptype-of BOX    OBJECT)
(ptype-of KEY    OBJECT)
(ptype-of ROOM   :top-type)
(ptype-of DOOR   :top-type)
(ptype-of LOCX   :top-type)
(ptype-of LOCY   :top-type)
(ptype-of Robot  :top-type)
(ptype-of Status :top-type)

(pinstance-of Robot ROBOT)
(pinstance-of Open  STATUS)
(pinstance-of Closed STATUS)


;;; ******************************************
;;;
;;;           O P E R A T O R S
;;;
;;; ******************************************

;;; Picking up and putting down objects

(operator PICKUP-OBJ
 (params <object>)
 (preconds
  ((<object> OBJECT))
  (and (arm-empty)
       (next-to robot <object>)
       (carriable <object>)))
 (effects
  ((<other-obj> (or OBJECT DOOR))
   (<other-obj2> (or OBJECT DOOR)))
  ((del (arm-empty))
   (del (next-to <object> <other-obj>))
   (del (next-to <other-obj2> <object>))
   (add (holding <object>)))))


(operator PUTDOWN
 (params <object>)
 (preconds
  ((<object> OBJECT))
  (holding <object>))
 (effects
  ()
  ((del (holding <object>))
   (add (next-to robot <object>))
   (add (arm-empty)))))


(operator PUTDOWN-NEXT-TO
 (params <object> <other-obj> <room>)
 (preconds
  ((<object> OBJECT)
   (<other-obj> (and OBJECT
                     (diff <object> <other-obj>)))
   (<room> ROOM))
  (and (holding <object>)
       (inroom <other-obj> <room>)
       (inroom <object> <room>)
       (next-to robot <other-obj>)))
 (effects
  ()
  ((del (holding <object>))
   (add (next-to <object> <other-obj>))
   (add (next-to robot <object>))
   (add (next-to <other-obj> <object>))
   (add (arm-empty)))))


;;; ******************************************
;;; Moving to doors (pushing or just going)
;;; and through doors (the robot by itself,
;;; pushing something, or carrying something)

(operator PUSH-TO-DR
 (params <box> <door> <room>)
 (preconds
  ((<door> DOOR)
   (<room> ROOM)
   (<box> BOX))
  (and (dr-to-rm <door> <room>)
       (inroom <box> <room>)
       (next-to robot <box>)
       (pushable <box>)))
 (effects
  ((<other>  (and (or OBJECT DOOR)
                  (diff <other> <box>)))
   (<other2> (or OBJECT DOOR))
   (<other3> (and (or OBJECT DOOR)
                  (diff <other3> robot))))
  ((del (next-to robot <other>))
   (del (next-to <box> <other2>))
   (del (next-to <other3> <box>))
   (add (next-to <box> <door>))
   (add (next-to robot <door>)))))


(operator PUSH-THRU-DR
 (params <box> <door> <roomx> <roomy>)
 (preconds
  ((<box> BOX)
   (<door> DOOR)
   (<roomx> ROOM)
   (<roomy> ROOM))
  (and (dr-to-rm <door> <roomx>)
       (dr-open <door>)
       (next-to <box> <door>)
       (next-to robot <box>)
```

```
       (pushable <box>)
       (connects <door> <roomx> <roomy>)
       (inroom <box> <roomx>)))
 (effects
  ((<other>  (and (or OBJECT DOOR)
                  (diff <other> <box>)))
   (<other2> (or OBJECT DOOR))
   (<other3> (and (or OBJECT DOOR)
                  (diff <other3> robot))))
  ((del (next-to robot <other>))
   (del (next-to <box> <other2>))
   (del (next-to <other3> <box>))
   (del (inroom robot <roomx>))
   (del (inroom <box> <roomx>))
   (add (inroom robot <roomy>))
   (add (inroom <box> <roomy>)))))


(operator GO-THRU-DR
 (params <door> <roomx> <roomy>)
 (preconds
  ((<door> DOOR)
   (<roomx> ROOM)
   (<roomy> ROOM))
  (and (arm-empty)
       (dr-to-rm <door> <roomx>)
       (dr-open <door>)
       (next-to robot <door>)
       (connects <door> <roomx> <roomy>)
       (inroom robot <roomx>)))
 (effects
  ((<other> OBJECT))
  ((del (next-to robot <other>))
   (del (inroom robot <roomx>))
   (add (inroom robot <roomy>)))))


(operator CARRY-THRU-DR
 (params <object> <door> <roomx> <roomy>)
 (preconds
  ((<object> OBJECT)
   (<door> DOOR)
   (<roomx> ROOM)
   (<roomy> ROOM))
  (and (dr-to-rm <door> <roomx>)
       (dr-open <door>)
       (holding <object>)
       (connects <door> <roomx> <roomy>)
       (inroom <object> <roomy>)
       (inroom robot <roomy>)
       (next-to robot <door>)))
 (effects
  ((<other> (and (or OBJECT DOOR)
                 (diff <other> <object>)
                 (diff <other> <door>)))
   (<other2> (or OBJECT DOOR))
   (<other3> (or OBJECT DOOR)))
  ((del (next-to robot <other>))
   (del (inroom robot <roomy>))
   (del (inroom <object> <roomy>))
   (add (inroom robot <roomx>))
   (add (inroom <object> <roomx>)))))


(operator GOTO-DR
 (params <door> <room>)
 (preconds
  ((<door> DOOR)
   (<room> ROOM))
  (and (dr-to-rm <door> <room>)
       (inroom robot <room>)))
 (effects
  ((<other> (or OBJECT DOOR)))
  ((del (next-to robot <other>))
   (add (next-to robot <door>)))))

;;; ******************************************
;;; Going next to boxes and pushing them

(operator PUSH-BOX
 (params <box> <object> <room>)
 (preconds
  ((<box> BOX)
   (<object> OBJECT)
   (<room> ROOM))
  (and (inroom <object> <room>)
       (inroom <box> <room>)
       (pushable <box>)
       (next-to robot <box>)))
 (effects
  ((<other>  (and (or OBJECT DOOR)
                  (diff <other> <box>)))
   (<other2> (or OBJECT DOOR))
   (<other3> (and (or OBJECT DOOR)
                  (diff <other3> robot))))
  ((del (next-to robot <other>))
   (del (next-to <box> <other2>))
   (del (next-to <other3> <box>))
   (add (next-to robot <object>))
   (add (next-to <box> <object>))
   (add (next-to <object> <box>)))))


(operator GOTO-OBJ
 (params <object> <room>)
 (preconds
  ((<object>  OBJECT)
   (<room> ROOM))
  (and (inroom <object> <room>)
       (inroom robot <room>)))
 (effects
  ((<other>  (or OBJECT DOOR)))
  ((del (next-to robot <other>))
   (add (next-to robot <object>)))))

;;; ******************************************
;;; actions with doors

(Operator OPEN-DOOR
 (params <door>)
 (preconds
  ((<door> Door))
  (and (next-to robot <door>)
       (unlocked <door>)
       (dr-closed <door>)))
 (effects
  ()
  ((del (dr-closed <door>))
   (add (dr-open <door>)))))


(operator CLOSE-DOOR
 (params <door>)
 (preconds
```

```
  ((<door> DOOR))
  (and (next-to robot <door>)
       (dr-open <door>)))
 (effects
  ()
  ((del (dr-open <door>))
   (add (dr-closed <door>)))))


(operator LOCK
 (params <door> <key> <room>)
 (preconds
  ((<door> DOOR)
   (<key> KEY)
   (<room> ROOM))
  (and (is-key <door> <key>)
       (holding <key>)
       (dr-to-rm <door> <room>)
       (inroom <key> <room>)
       (next-to robot <door>)
       (dr-closed <door>)
       (unlocked <door>)))
 (effects
  ()
  ((del (unlocked <door>))
   (add (locked <door>)))))


(operator UNLOCK
 (params <door> <key> <room>)
 (preconds
  ((<door> DOOR)
   (<key> KEY)
   (<room> ROOM))
  (and (is-key <door> <key>)
       (holding <key>)
       (dr-to-rm <door> <room>)
       (inroom <key> <room>)
       (inroom robot <room>)
       (next-to robot <door>)
       (locked <door>)))
 (effects
  ()
  ((del (locked <door>))
   (add (unlocked <door>)))))

;;; ******************************************
;;;
;;;       I N F E R E N C E   R U L E S
;;;
;;; ******************************************

(inference-rule KEYS-ARE-CARRIABLE
  (mode eager)
  (params <key>)
  (preconds
   ((<key> KEY))
   ())
  (effects
   ()
   ((add (carriable <key>)))))


(inference-rule CONNECTS-IS-SYMMETRICAL
 (mode eager)
 (params <door> <room1> <room2>)
 (preconds
```

```
  ((<door> Door)
   (<room1> Room)
   (<room2> Room))
  (connects <door> <room1> <room2>))
 (effects
   ()
   ((add (connects <door> <room2> <room1>)))))


(inference-rule INFER-ARM-EMPTY
 (mode lazy)
 (params)
 (preconds
   ()
   (~ (exists ((<obj> OBJECT))
        (holding <obj>))))
 (effects
   ()
   ((add (arm-empty)))))


;;; ******************************************
;;;
;;;       C O N T R O L   R U L E S
;;;
;;; ******************************************

;;; Goal decision rules

(control-rule LOCK-LAST
  (if (and (candidate-goal (inroom <x> <room>))
           (candidate-goal (locked <door>))
           (dr-to-rm <door> <room>)))
  (then prefer goal (inroom <x> <room>)
                    (locked <door>)))

;;; ******************************************
;;; Binding decision rules

(control-rule PUTDOWN-FOR-ARM-EMPTY
  (if (and (current-goal (arm-empty))
           (current-operator PUTDOWN)
           (known (holding <obj>))))
  (then select bindings ((<object> . <obj>))))

(control-rule GOTO-OBJ-ROOM
  (if (and (current-goal  (next-to robot <obj>))
           (type-of-object <obj> OBJECT)
           (current-operator GOTO-OBJ)
           (known (inroom <obj> <rm>))))
  (then select bindings
        ((<object> . <obj>)(<room> . <rm>))))

(control-rule GOTO-DR-ROOM
  (if (and (current-goal  (next-to robot <dr>))
           (type-of-object <dr> DOOR)
           (current-operator GOTO-DR)
           (known (inroom robot <rm>))
           (known (dr-to-rm <dr> <rm>))))
  (then select bindings
        ((<door> . <dr>)(<room> . <rm>))))

;; I need a generator for <k1> because <k1> in
;; (~(known)) has to be bound:

(control-rule GO-THRU-LOCKED-DR
  (if (and (current-goal (inroom robot <room>))
```

```
           (current-operator GO-THRU-DR)
           (known (dr-to-rm <dr> <room>))
           (known (locked <dr>))
           (unbound-neg-known
              (is-key <dr> <k1>) (<k1>  KEY))))
  (then reject bindings ((<door> . <dr>))))

;;; *****************************************
;;; Operator decision rules

(control-rule PUSH-IF-NOT-CARRIABLE
  (if (and (current-goal (next-to <x> <y>))
           (candidate-operator PUTDOWN-NEXT-TO)
           (~ (known (carriable <x>))
           (~ (known (carriable <y>))))))
  (then reject operator PUTDOWN-NEXT-TO))

;;;rules from Prodigy2.0 version

(control-rule ONLY-GOTO-DR
  (if (and (current-goal  (next-to robot <x>))
           (type-of-object <x> DOOR)))
  (then select operator GOTO-DR))

(control-rule ONLY-GOTO-OBJ
  (if (and (current-goal  (next-to robot <x>))
           (type-of-object <x> OBJECT)))
  (then select operator GOTO-OBJ))

(control-rule ONLY-CARRY-BOX-THRU-DOOR
  (if (and (current-goal  (inroom robot <x>))
           (candidate-operator CARRY-THRU-DR)))
  (then reject operator CARRY-THRU-DR))

(control-rule ONLY-PUSH-BOX-THRU-DOOR1
  (if (and (current-goal  (inroom robot <x>))
           (candidate-operator PUSH-THRU-DR)))
  (then reject operator PUSH-THRU-DR))

(control-rule ONLY-PUSH-BOX-THRU-DOOR2
  (if (and (current-goal  (next-to <y> <x>))
           (candidate-operator PUSH-THRU-DR)))
  (then reject operator PUSH-THRU-DR))

(control-rule ONLY-PUTDOWN-NEXT-TO-1
  (if (and (current-goal  (arm-empty))
           (candidate-operator PUTDOWN-NEXT-TO)))
  (then reject operator PUTDOWN-NEXT-TO))

(control-rule ONLY-PUTDOWN-NEXT-TO-2
  (if (and (current-goal  (inroom <xx> <yy>))
           (candidate-operator PUTDOWN-NEXT-TO)))
  (then reject operator PUTDOWN-NEXT-TO))
```

```
;;; *****************************************
;;; File: functions.lisp
;;; 1/25/92 aperez

;;; *****************************************
;;;
;;;          F U N C T I O N S
;;;
;;; *****************************************


(defun diff (x y)
  (not (equal x y)))
```

Index

Contents