Supporting Combined Human and Machine Planning: The Prodigy 4.0 User Interface Version 2.0*

Michael T. Cox mcox@cs.cmu.edu

Manuela M. Veloso mmv@cs.cmu.edu

September, 1997
CMU-CS-97-174

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213-3891

Abstract

Realistic and complex planning situations require a mixed-initiative planning framework in which human and automated planners interact to mutually construct a desired plan. Ideally, this joint cooperation has the potential of achieving better plans than either the human or the machine can create alone. Human planners often take a case-based approach to planning, relying on their past experience and planning by retrieving and adapting past planning cases. Planning by analogical reasoning in which generative and case-based planning are combined, as in Prodigy/Analogy, provides a suitable framework to study this mixed-initiative integration. However, having a human user engaged in this planning loop creates a variety of new research questions. The challenges we found creating a mixed-initiative planning system fall into three categories: planning paradigms differ in human and machine planning; visualization of the plan and planning process is a complex, but necessary task; and human users range across a spectrum of experience, both with respect to the planning domain and the underlying planning technology. This paper presents our approach to these three problems when designing an interface to incorporate a human into the process of planning by analogical reasoning with Prodigy/Analogy. The interface allows the user to follow both generative and case-based planning, it supports visualization of both plan and the planning rationale, and it addresses the variance in the experience of the user by allowing the user to control the presentation of information.

---

*This research is sponsored as part of the DARPA/RL Knowledge Based Planning and Scheduling Initiative under grant number F30602-95-1-0018. A short version of this document appeared as Cox, M. T., & Veloso, M. M. (1997). Supporting combined human and machine planning: An interface for planning by analogical reasoning. In D. B. Leake & E. Plaza (Eds.), *Case-Based Reasoning Research and Development: Second International Conference on Case-Based Reasoning* (pp. 531-540). Berlin: Springer-Verlag.

Introduction

In mixed-initiative planning, automated and human planners need to interact in order to mutually construct a plan that satisfies a set of goals in a specific situation. Ideally, joint cooperation has the potential of achieving better plans than either the human or the machine can create alone (Ferguson, Allen, & Miller, 1996; Oates & Cohen, 1994). However, given the significant disparity between human and automated planning mechanisms, achievement of this potential is a difficult goal. The challenges of creating a successful mixed-initiative planning system fall into at least three categories: planning paradigms differ in human and machine planning; plan visualization is a complex, although necessary, task; and human users range across a spectrum of experience, both with respect to the planning domain and the planning technology. This paper describes the directions we are pursuing to address these problems when designing a mixed-initiative interface for the PRODIGY planning system.

One of the most significant problems facing the integration of human and automated planning is the cognitive metaphor shared between the participants. In general, AI planning assumes a model of actions in the world and generates new plans by searching the space of possible actions. Alternatively, the case-based metaphor of planning as a memory task (Hammond, 1989) can be more accessible to the human user. Indeed, few humans plan as if they have never faced a problem like the current one before (Riesbeck & Schank, 1989). Instead, the solution is a matter of remembering concrete past experience to form a new plan similar to an old one previously performed (Kolodner, 1993; Seifert et al., 1994). Yet, when gaps exist in experience, reasoning from first principles is equally as natural and necessary for deriving a successful plan. The Prodigy/Analogy planning system represents a hybrid metaphor that reflects planning in a more complete manner than is the case with traditional case-based planners. Within the same planning system, both generative and case-based algorithms have an equal role.

A related issue is the task of visualizing the plan itself. Although a plan can be conceptualized as a simple sequence of actions, the relationship of each action to the goals of the planner and the state of the world has complex structure. Plan steps often interact with each other depending on the conditions necessary for performing the action and the results of having carried out given actions. Instead of sequential action, a goal tree representation aids a planner when trying to discover or make concrete the structure of plans. Moreover, the *reasons* for why certain actions are chosen (that is, the planning justifications or rationale) are seldom represented, although beneficial for a user when trying to reason in complex domains either from a past case or from first principles. Therefore as presented in this paper, we have designed a mixed-initiative planning interface that supports the planning process through plan and rationale visualization. The inclusion of both facets of planning is aimed at supporting the metacognitive as well as the cognitive aspects of reasoning (Cox, 1996).

Finally, users of mixed-initiative planning systems differ in experience along at least two dimensions. Users differ in experience with respect to their knowledge and skills of the planning domain (e.g., military deployment planning), and they also differ with respect to their knowledge and skills of the underlying planning technology in the mixed-initiative system. Therefore, a tension exists when presenting information to the user. In some cases a user may wish to override or have access to information concerning the operation of the automated planner, while such information and decisions may bewilder other users. From the beginning, the PRODIGY philosophy has been one of a *glass-box* approach, allowing the user to exert control over all aspects of decision making in the planner (Carbonell, Knoblock & Minton, 1991). Within a mixed-initiative framework, however, a *black-box* philosophy that shields some of the technological and implementational details from the user (especially the naïve user) can be appropriate as well (Cox & Veloso, 1997). A major goal of the interface reported here is to develop a *selective-control mechanism* to support the range of expertise from novice to expert in both dimensions. That is, the user should be able to examine and control any desired decision, while the remainder is abstracted or hidden from the user. This paper reports on the implementation of the interface that begins to support such a philosophy.[1]

The mixed-initiative interface contains a number of features that solve or mitigate the above problems. The user is able to switch between generative and analogical planning manually, or the interleaving of both modes can be left to automation. The interface provides mechanisms to save in the case library planning cases created generatively or analogically, to retrieve old cases that match current demands (either automatically or manually) and to choose various case interleaving strategies for adaptation and replay. The evolving plan is graphically presented to the user as a goal-tree structure and justifications for automated choices is displayed upon demand. Finally, the user

[1] The interface is publicly available via http://www.cs.cmu.edu/~prodigy

can maintain a custom level of information display and user-control depending upon experience and interests.

Section 2 describes the PRODIGY control algorithms, both in generative and analogical mode. Section 3 then introduces the mixed-initiative interface of PRODIGY, describing how both the plan and the planning decisions are displayed to the user. Section 4 briefly describes the current efforts to make the interface more responsive to the user's level of expertise. Finally, Section 5 concludes with a short discussion.[2]

PRODIGY: A Hybrid of Generative and Case-Based Planning

PRODIGY is a fully-automated planner that combines generative state-space planning and case-based planning. In a generative planning mode (Prodigy4.0), the system uses search through a space of operator choices. When under case-based planning mode (Prodigy/Analogy), it retrieves from a case library past plans that are most similar to a given new problem. These plans are reused (replayed) to create a solution for the current goals.

The Generative Planning Algorithm

The Prodigy4.0 system (Carbonell et al., 1992) employs a state-space nonlinear planner and follows a means-ends analysis backward-chaining search procedure that reasons about both multiple goals and multiple alternative operators from its domain theory appropriate for achieving such goals. A domain theory is composed of a hierarchy of object classes and a suite of operators and inference rules that change the state of the objects. A planning problem is represented by an initial state (objects and propositions about the objects) and a set of goal expressions to achieve. Planning decisions consist of choosing a goal from a set of pending goals, choosing an operator to achieve a particular goal, choosing a binding for a given operator, and deciding whether to commit to a possible plan ordering and to get a new planning state or to continue subgoaling for unachieved goals. Different choices give rise to different ways of exploring the search space. These choices are guided by either control rules, by past problem-solving episodes (cases), or by domain-independent heuristics.

[2]Appendix A includes a User Overview of the Interface.

As shown in Figure 1, Prodigy4.0 follows a sequence of decision choices, selecting a goal, an operator, and an instantiation for the operator to achieve the goal. Prodigy4.0 has an additional decision point, namely where it decides whether to "apply" an operator to the current state or continue "subgoaling" on a pending goal. "Subgoaling" can be best understood as regressing one goal, or backward chaining, using means-ends analysis. It includes the choices of a goal to plan for and an operator to achieve this goal. "Applying" an operator to the state means a commitment (not necessarily definite since backtracking is possible) in the ordering of the final plan. On the other hand, updating the state through this possible commitment allows Prodigy4.0 to use its state to more informed and efficient future decisions. Hence, the planning algorithm is a combination of state-space search corresponding to a simulation of plan execution of the plan (the head plan; Fink & Veloso, 1996) and backward-chaining responsible for goal-directed reasoning (the tail plan).

1. Initialize.
2. Terminate if the goal statement has been satisfied.
3. Determine which goals are pending, i.e. still need to be achieved.
4. Determine if there are any selected operators that have their preconditions satisfied in the current state, and hence could be applied to the state as the next step in the plan.
5. Choose to subgoal on a goal or to apply an operator: (backtrack point) * To subgoal, go to step 6 * To apply, go to step 7
6. Select one of the pending goals (no backtrack point), an instantiated operator that can achieve it (backtrack point); go to step 3.
7. Change the state according to an applicable operator (backtrack point); go to step 2.

Figure 1. A top-level view of Prodigy4.0's planning algorithm.

The reasons that these choices are made by the automated planner (or, if under user control, the user) are preserved in a plan derivation trace to improve planning efficiency in the future. When the user saves a planning episode, these decision justifications (i.e., the traces) are saved along with the solution in the case library. The case is indexed by the problem goals and a subset of the initial state responsible for the achievement of the goals.

The Derivational Analogy (CBR) Planning Algorithm

Under a case-based planning mode, Prodigy/Analogy (Veloso, 1994) creates plans, interprets and stores planning episodes, and retrieves and reuses multiple past plans that are found similar to new problems. Stored plans are annotated with plan rationale so that, when the plans are retrieved in the future, new decisions can be guided and validated by the past rationale, hence avoiding inefficient search as can be the case in generative mode. The derivational-analogy strategy is to derive new solutions based on the *decision-making process* used in the past, rather than by adapting old *solutions* created in the past (Carbonell, 1986).

Figure 2 outlines the derivational analogy algorithm used by the system. When a new problem is proposed, Prodigy/Analogy retrieves from the case library one or more problem solving episodes that may partially cover the new problem solving situation. The system uses a similarity metric that weighs goal-relevant features (Veloso & Carbonell, 1993). Essentially, it selects a set of past cases that solved subsets of the new goal statement. The initial state is partially matched in the features that were relevant to solving these goals in the past. Each retrieved case provides guidance to a set of interacting goals from the new goal statement. At replay time, a guiding case is always considered as a source of guidance, until all the goals it covers are achieved.

1. Initialize.
2. Select a case to follow: follow the justifications and the selected strategy to merge the guidance from multiple past cases.
3. Get the relevant operators from the past cases.
4. Prune alternative failures from the current search path if the reasons for past failure hold.
5. Check syntactic operator-applicability by testing whether its left-hand side matches in the current state.
6. Check semantic applicability by determining whether the past reasons for their use still hold.
7. If choice not valid, choose a suitable action:
    - Suspend the guiding case if extra planning work needed to make choice valid:
        - Retrieve additional case
        - Or replan using domain theory.
    - Advance guiding case for past planning work that is not necessary.
    - Change the focus of attention by selecting another guiding case.

Figure 2. A top-level view of the case-based planning algorithm.

The general replay mechanism involves a complete interpretation of the justification structures annotated in the past cases in the context of the new problem to be solved. Equivalent choices are made when the transformed justifications hold. When that is not the situation, Prodigy/Analogy plans for the new goals using its domain operators adding new steps to the solution or skipping unnecessary steps from the past cases.

Prodigy/Analogy constructs a new solution from a *set* of guiding cases, as opposed to a single past case. Complex problems may be solved by resolving minor interactions among simpler past cases. However, following several cases poses an additional decision making step of choosing which case to pursue. Prodigy/Analogy includes several strategies to merge the guidance from the set of similar cases (Veloso, 1997).

Visualization of the Plan and the Planning Process

One of the main goals underlying the design of the graphical user interface for PRODIGY is to provide a clear animation of the planning algorithm (Carbonell et al., 1992). Given the planning algorithm outlined in the previous section, we discuss several features in the user interface that enable the visualization of the running of algorithm.

The user interface (UI) is implemented in Tcl/Tk, a scripting language which includes a set of Motif widgets (see Blythe, Veloso, & de Souza, 1997 for additional implementational details). The user interface runs in a separate process from the planner, and the two communicate through Tcl/Tk's IPC mechanism. The planner currently runs in a Common Lisp process that listens for commands from two sources: the terminal running lisp and a socket connected to the user interface. This allows problems to be loaded or planning to be initiated from either the terminal or the interface. The interface makes use of Dag, a publicly available preprocessor for drawing directed graphs (Gansner, North, & Vo, 1988) that represent characteristics of the domain and plan. For example, when the user chooses to view a partial ordering of the plan, the UI sends a message to the planner to print the graphical representation to a file and runs Dag on that file. The output from Dag is a file containing placements for nodes and spline commands for edges that the UI reads to produce a window containing the graph.

The original interface to PRODIGY is a simple command line interface from Lisp (see Figure 3). The user loads a planning domain and a problem to solve, retrieves and loads past cases, and then calls the planner by invoking the run command (optional program parameters can be controlled with command-line switches not shown). The output is composed of print statements at selected decision points in the search tree with indentation indicating search depth. The plan is printed as a list of instantiated operators, and then a brief statement reports success, running time, the number of search nodes expanded, and the number of solutions obtained.

Figure 4 shows the integrated system consisting of the original Lisp interface contained in an

```
USER(66): (domain 'rocket)
; Loading /afs/cs/project/prodigy-1/analogy/domains/rocket/domain.lisp.
Subgoal behaviour: SUBGOAL-ALWAYS-BEFORE-APPLY
;   Loading /afs/cs/project/prodigy-1/analogy/replay-crs.lisp.
Reading Meta predicate: analogy-get-guidance-goal
Reading Meta predicate: analogy-get-guidance-operator
Reading Meta predicate: analogy-get-guidance-bindings
Reading Meta predicate: analogy-decide-subgoal
Reading Meta predicate: analogy-decide-apply
0.0
USER(67): (problem 'prob2objs-2)
; Loading /afs/cs/project/prodigy-1/analogy/domains/rocket/probs/prob2objs-2.lisp.
T
USER(68): (use-cases 2 0.6 "case-prob1-blancmange" "case-prob1-hammer" "case-prob1-robot")
NIL
USER(69): (init-guiding)
List of available cases :
 0: "CASE-PROB1-ROBOT.15036"
 1: "CASE-PROB1-BLANCMANGE.15038"
 Current *merge-mode* is SABA
NIL
USER(70): (run)
Creating objects (LOCA LOCB) of type LOCATION
Creating objects (OBJ1 OBJ2) of type OBJECT
Creating object R1 of type ROCKET

  2 n2 (done)
  4 n4 <*finish*>
  5   n5 (at obj2 locb)
  7   n7 <unload-rocket obj2 locb r1>
  8     n8 (inside obj2 r1)
 10     n10 <load-rocket obj2 r1 loca>
 11   n11 (at obj1 locb)
 13   n13 <unload-rocket obj1 locb r1>
 14     n14 (inside obj1 r1)
 16     n16 <load-rocket obj1 r1 loca>
 17     n17 <LOAD-ROCKET OBJ1 R1 LOCA> ("CASE-PROB1-BLANCMANGE.15038" 10)
 18     n18 (at r1 locb)
 20     n20 <move-rocket r1>
 21     n21 <LOAD-ROCKET OBJ2 R1 LOCA> ("CASE-PROB1-ROBOT.15036" 10)
 22     n22 <MOVE-ROCKET R1> ("CASE-PROB1-BLANCMANGE.15038" 14)
 23   n23 <UNLOAD-ROCKET OBJ1 LOCB R1> ("CASE-PROB1-BLANCMANGE.15038" 7)
 23   n24 <UNLOAD-ROCKET OBJ2 LOCB R1> ("CASE-PROB1-ROBOT.15036" 7)
Achieved the following top-level goals:
((AT OBJ1 LOCB) (AT OBJ2 LOCB))
Solution:
     <load-rocket obj1 r1 loca> ("CASE-PROB1-BLANCMANGE.15038" 10)
     <load-rocket obj2 r1 loca> ("CASE-PROB1-ROBOT.15036" 10)
     <move-rocket r1> ("CASE-PROB1-BLANCMANGE.15038" 14)
     <unload-rocket obj1 locb r1> ("CASE-PROB1-BLANCMANGE.15038" 7)
     <unload-rocket obj2 locb r1> ("CASE-PROB1-ROBOT.15036" 7)

#<PRODIGY result: T, 0.7 secs, 24 nodes, 1 sol>
```

Figure 3. Original command line interface from Lisp

To obtain the above output the following parameters were set: Output level set to 2; *talk-case-p* set to nil; *verbose* set to nil; and the output of call to use-cases removed for sake of brevity.

inferior Lisp buffer within emacs, the main User Interface window and displays, and the Load Domain and Problem window. As with the direct command-line interface in Lisp, the user can load a domain and problem with the "Load" button on the left of the main UI window (it enables the Load Domain and Problem window shown in the figure), control program parameters through the "Control Variables" pull-down menu, and execute the planner with the "Run" button, or incrementally step through the execution using the "Step" button. During planning, the user can "Break" the execution to pause and then "Restart" to continue or "Abort" to halt the execution after a break. The system is running a simple problem in the "One-Way Rocket" domain[3] under Generative Planning Mode (Prodigy4.0). The tail plan is graphically illustrated in the Goal-Tree Display, and the head plan is now printed as a list of applied instantiated operators (action steps) in the Plan Display window to the right.

Figure 5 shows the graphical user interface for PRODIGY (in CBR mode). The main control and display window is at the top of the figure, while two cases for replay are shown below. During execution of either generative or case-based planning, the current state of the plan is maintained in two major graphical windows (Tk canvases). The program generates a view of the tail plan in the Goal Tree Display on the left, and the head plan is printed as a list of committed planning steps in the Instantiated Operator Display on the right. Note that while in CBR mode, the "Run" button is relabeled the "Replay" button.

The segment of the goal tree displayed in Figure 5 shows that the user is finished with planning in the current problem in the one-way rocket domain if the two top goals are solved (i.e., objects 1 and 2 are at location locb). These goals can be solved if operator unload-rocket is executed twice while rocket R1 containing both objects is at locb (the goal destination). The two previous cases below the main window show a merge of separate parts of the plan. Stars mark reused plan steps, the arrows mark current locations in the old plans, and blank spaces indicate skipped steps. The Instantiated Operator Display enumerates the steps of the final merged plan. Although simplified due to space limitations here, the UI provides full access to PRODIGY to help the user visualize arbitrarily complex planning examples.

Selecting from the "Planning Mode" pull-down menu, the user can alternate control to and

[3]See Appendix B for a listing of the operators and type hierarchy for that domain.

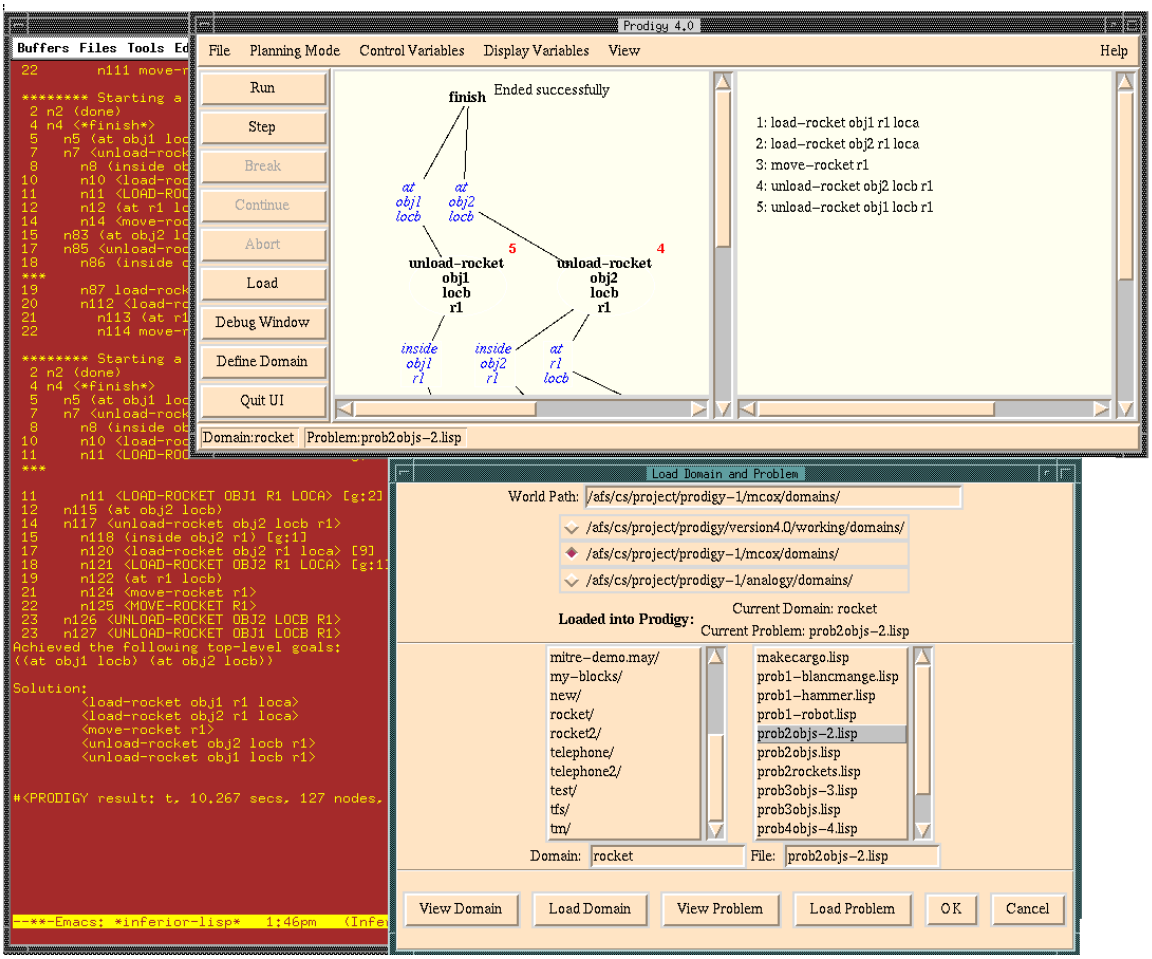


Figure 4. Overall User Interface under generative mode

from a generative and case-based planning mode.[4] In response, a number of mode-specific buttons are arranged along the bottom of the window. The Prodigy/Analogy buttons allow for retrieval (either automated or manual), storage, and merge selection (together with a button that presents to the user a demonstration of multiple case merge and replay from the rocket domain). The user can visualize the merging procedure as it interleaves multiple cases, marks the steps that are used after successful validation, and skips the ones that are no longer necessary or are invalid. Guiding cases are displayed as shown in the two additional windows at the bottom of Figure 5.

[4]Alternatively, the user can switch to a conditional-planning version that can use analogy along with a generative mechanism (Blythe, 1996; Blythe & Veloso, 1997).

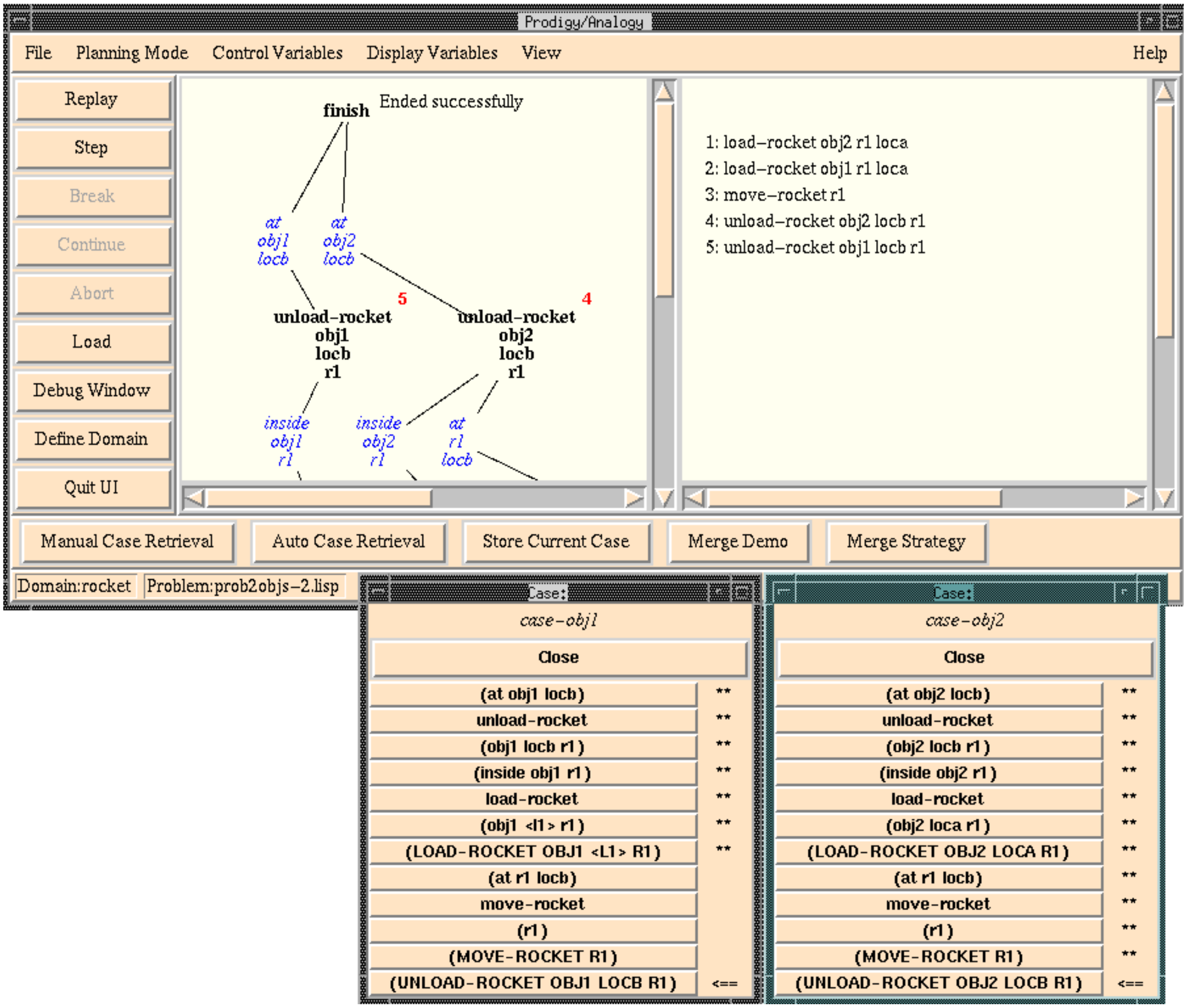


Figure 5. Case replay, merge, and adaptation

Figure 6 shows the interface window for manual retrieval. The user is able to over-ride the cases Prodigy/Analogy believes to be most similar to the new problem and to select one or more cases in the library for a particular domain. Cases are clicked for selection (or chosen with the "Select Case" button) and they are loaded in to the system via a "Load Selected Cases" button. The loading procedure establishes variable substitution according to the new goals and initial state and maps these values to the operator variables in the old cases.

The UI attaches actions to the displayed graph nodes in the main window so that if the user clicks on them it will pass another message to the planner to display appropriate information. From

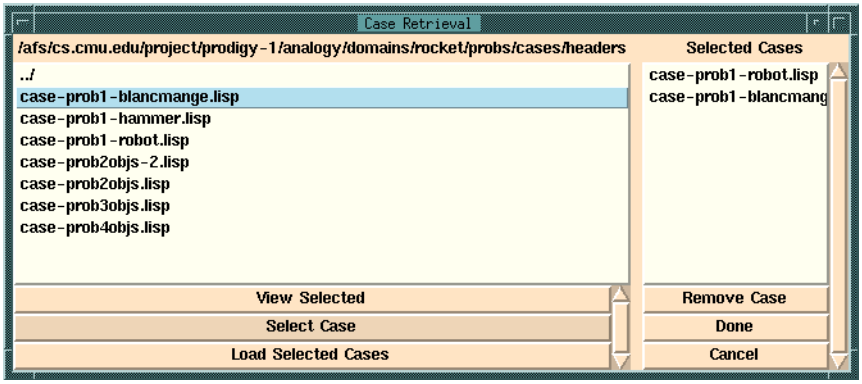


Figure 6. Manual case retrieval

the Goal Tree Display, the left mouse button causes the system to display the search tree node that relates justifications for the choice of that operator or goal (see Figure 7). Among other information, the sub-windows show that the choice of goal and operator bindings were made based on past experience linked to `case-obj2`. The middle (or right) mouse button causes the system to display the operator definition from the domain theory. In addition, the applied-operator search nodes corresponding to the plan steps are available with a left mouse click in the Plan Display. From these opened windows the user can obtain the parent and children of a search tree node or information in the property lists of a given node.

The nodes also reveal other types of information about the rationale of the planner (whether human or automated). For example Figure 8 shows why some planning decisions fail.[5] Node number 25 fails dues to an attempt to re-establish a goal condition, while node 22 fails due to attempting to achieve a goal that is not possible (the rocket destination is not legal). Both of these failure justifications are saved to a case so that when following similar decisions in the future, the search path can be pruned *a priori*. In addition to these types of justifications, the cases represent decisions based on user selection, the application of control rules (see node 12 in Figure 7 of Appendix C), failure due to state loops, and success due to goal achievement.

Finally, the interface allows the user to "drill down" into information within any search node. All information displayed can itself be inspected with a left mouse click to obtain further details.

[5]The failures shown are under generative mode and without control rules. Thus, variable bindings and goals are selected arbitrarily. Under case guidance this is less likely to occur given a good matching past case.

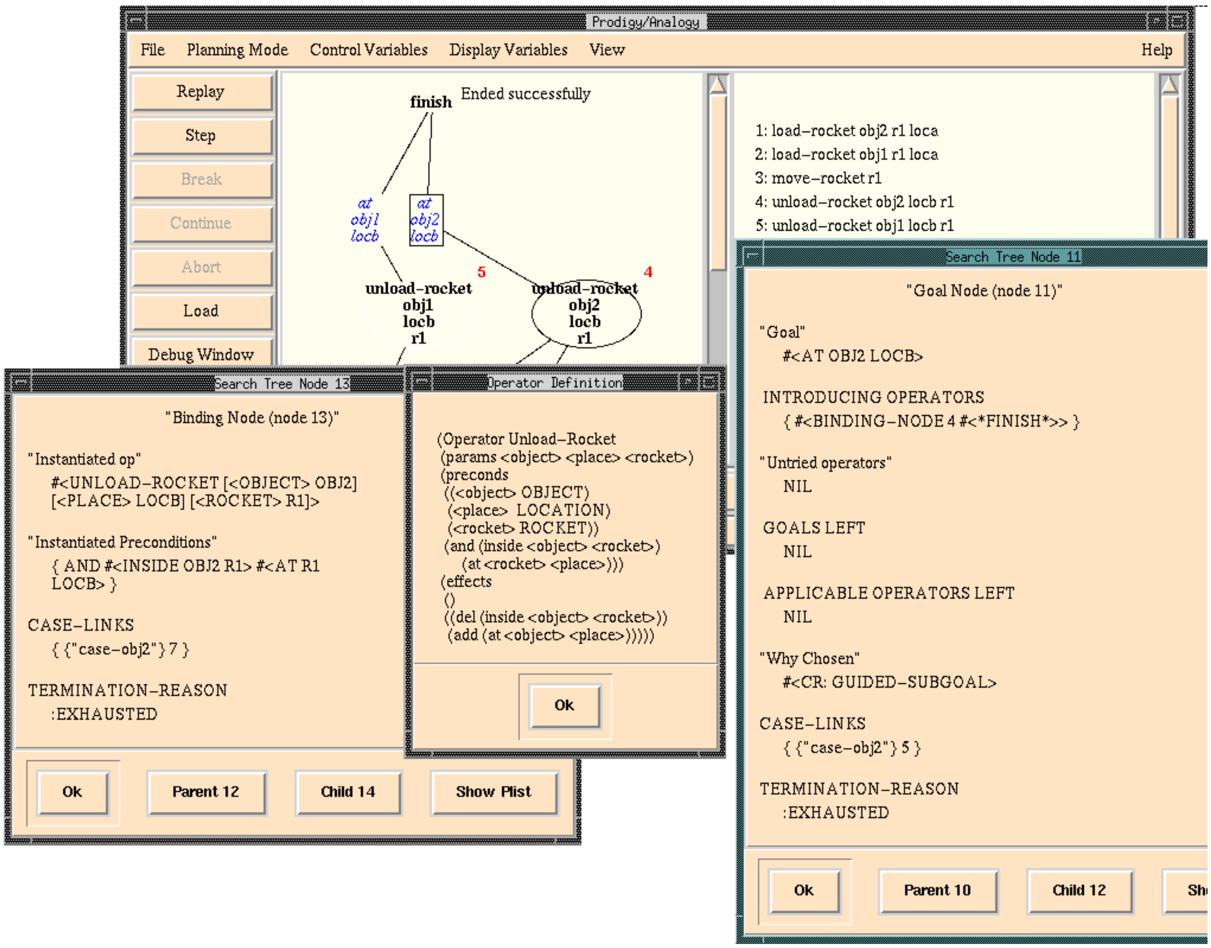


Figure 7. Inspecting decision nodes from the goal tree

An example of this recursive feature is shown in Figure 9. Here the user is examining goal node number 11. The rationale for why it was chosen is shown to be a control rule. Inspecting this rule, the user sees that it is a control rule that enforces a Subgoal Always Before Apply (SABA) policy, rather than a control rule from the domain. Additionally, the figure shows that the goal was originally introduced by instantiated operator Unload-Rocket with the given variable bindings (binding node 7). Its preconditions (shown as a list of two Prodigy states) include the requirement that the rocket R1 be at the location locb. Figure 9 also shows an examination of the object R1. An interesting feature is that the system allows the user to list all literals in the world state that include a reference to the object (see the R1 object window).

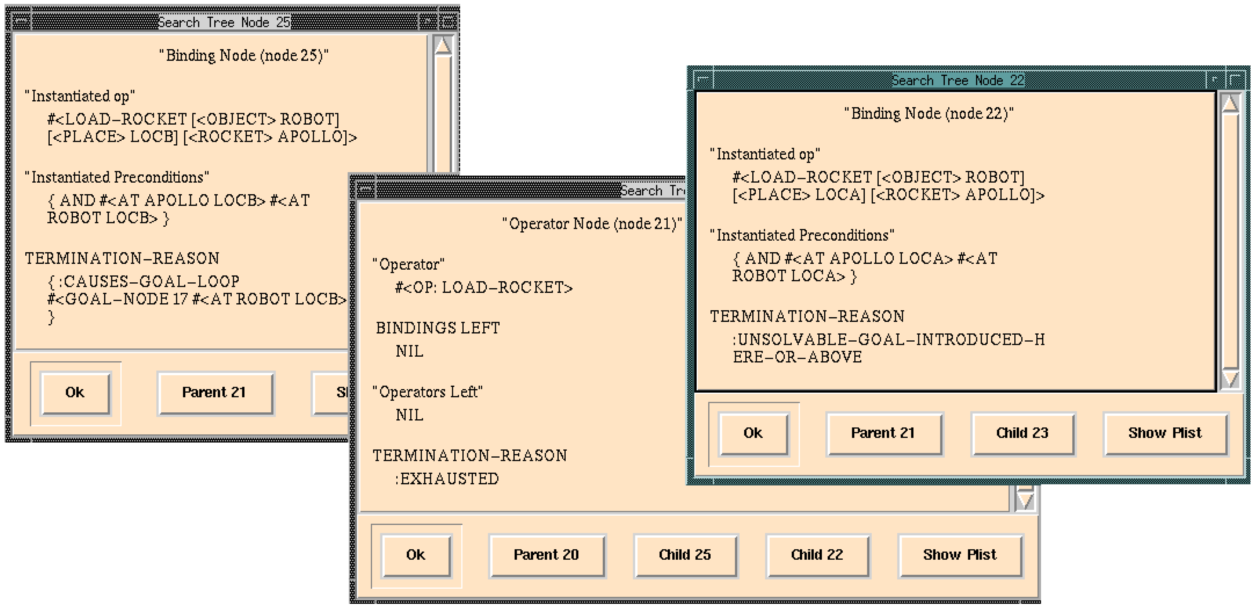


Figure 8. Reasons for failed decisions.

Expertise in Planning Versus Expertise in Technology

At the current time, the interface to PRODIGY has been optimized for users who are familiar with the underlying planning system (e.g., the AI research community in planning). As such it provides a detailed level of access to the internal mechanisms, data representations, and plan structure. For example, the entire decision cycle described in Section 2 is open for inspection through the search-tree node windows (see Appendix C for an illustration of one cycle). However, given that the development of the interface is still a work in progress, we have made substantial efforts to facilitate use by the more naïve user. A context sensitive help system is partially in place, the ease of execution of the planner has improved given menu selections to replace command-line switches, flags, and function calls, a substantial domain-specification subsystem exists, and the graphical display of the plan enhances plan visualization. Also, although not shown here (see Blythe, Veloso, & de Souza, 1997, instead), the user can easily preempt automated planning decisions, making them manually instead. Finally, the system supports a concrete experiential basis for planning (i.e., planning is by cases, rather than strictly by first principles).

To further support the planner who is both a novice in the planning domain and knows little

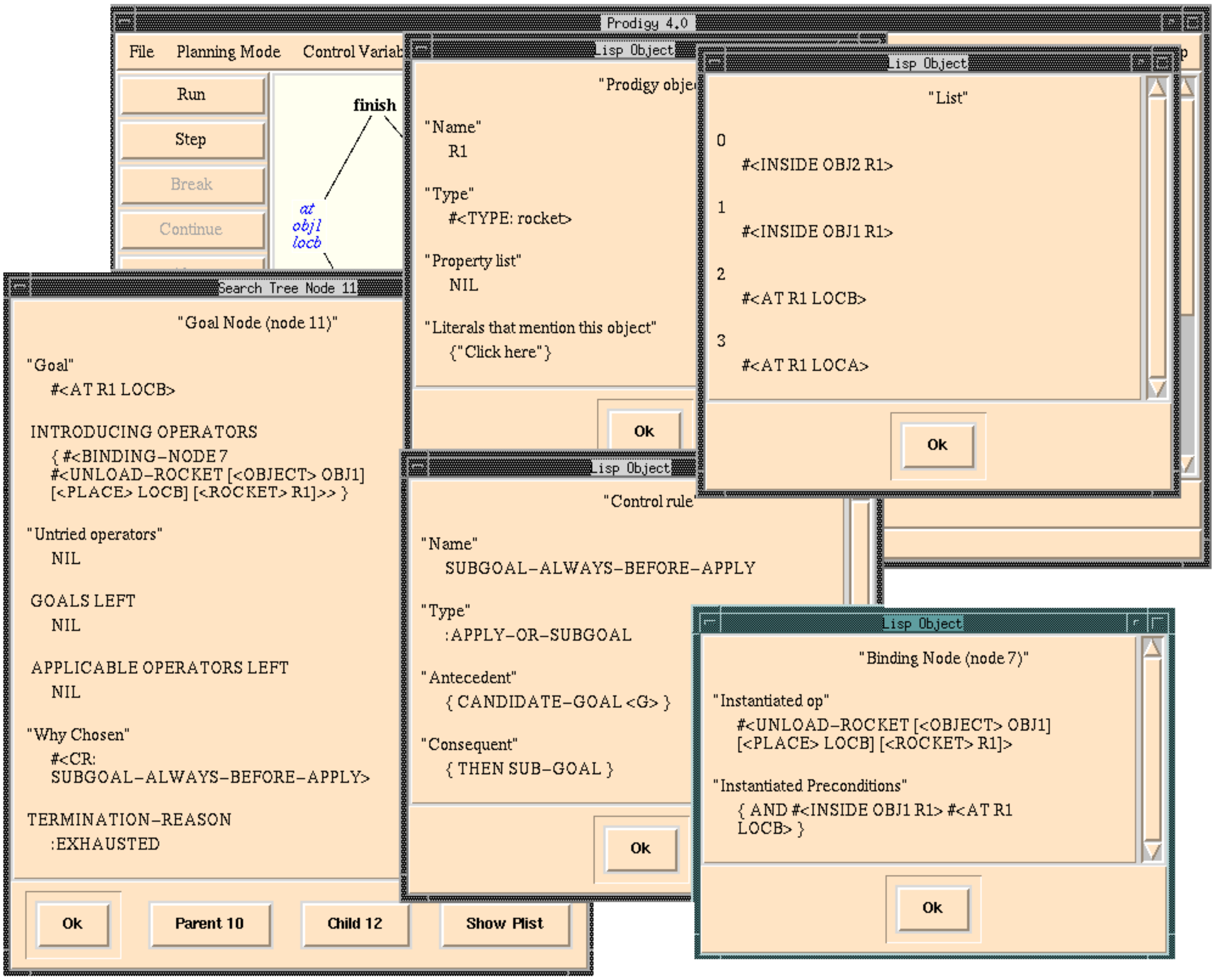


Figure 9. Drilling down into search node information

about planning formalisms, we have integrated Prodigy/Analogy with the ForMAT military-force deployment planner (see Veloso, Mulvehill, & Cox, 1997).[6] Using stored cases that match current planning problems, Prodigy/Analogy provides adaptation advice to the ForMAT user when modifying old deployment plans. In this mode, the user is remote across the internet and knows nothing about the manner in which Prodigy/Analogy works.

Finally, we are also working on a new *novice mode* that presents the output to the user in a more natural format. For example, instead of representing the predicates in prefix form in the goal

[6] Appendix C shows planning decisions from a problem taken from this domain.

tree display, infix is output so that is reads more like natural language (See Figure 10). Likewise in this mode, we are substituting the technical data-structure terms in the justification windows with more intuitive language that describes their function.

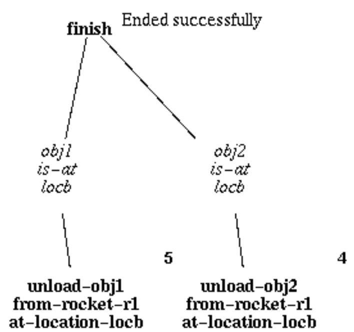


Figure 10. Goal tree segment generated in infix representation

Conclusion

This paper presents a case-based, mixed-initiative and extensible user interface built using Tcl/Tk that we developed for the PRODIGY architecture. Planning is a complex process and developing user planning interfaces is an important contribution for making implemented systems available to researchers, students, and practitioners. The user is able to switch between generative and analogical planning manually, or the interleaving of both modes can be left to automation. The interface provides mechanisms to save planning cases created generatively or analogically, to retrieve old cases that match current demands (either automatically or manually) and to choose various case interleaving strategies for adaptation and replay. The evolving plan is graphically presented to the user as a goal-tree structure, and justifications for automated choices is displayed upon demand.

The goals of the interface have been to facilitate the interaction between human and machine during the planning process by integrating both generative and case-based planning in the same framework, by providing plan visualization and plan rationale information, and by working toward a more flexible architecture that allows the user to settle to a personal level of equilibrium between too little information and too much. Because the user should not be arbitrarily subjected to the full technological details of the underlying planning technology, our ultimate aim is to develop a user view that abstracts the details of the underlying planner according to need. Our belief is that the immediate focus should be upon what the user sees (the interface), what the user wants (the goals), and what the user does (the task). Additional details should remain transparent to the user unless requested.

Acknowledgments

This research is sponsored as part of the DARPA/RL Knowledge Based Planning and Scheduling Initiative under grant number F30602-95-1-0018. The authors thank Jim Blythe for help with programming the "Drill Down" capabilities of the UI, with the Operator-Assertion Dependency graph generator, and many other features. We also thank Jason Reisman for programming the Tk code that invokes the manual case-retrieval function for the user interface.

Appendix A: User Overview of the PRODIGY User Interface (Version 2.0)

NOTE for those users who have prior experience with using PRODIGY 4.0 from a Lisp command line interface: the domain and problem commands can be invoked from the Load command button or pull-down menu item, and the run command can be invoked with the Run command button.

Introduction

The PRODIGY User Interface (UI) Version 2.0 is designed for two purposes and for two classes of users. First, the interface makes it easier for the PRODIGY researcher to access internal data structures (i.e., the values of search tree nodes) and to view the head plan and tail plan graphically. Secondly, the interface makes its easier for the PRODIGY user to control and execute the underlying planning system and to define domains (i.e., the abstraction hierarchy, the planning operators, and the planning problems). For both users, the UI supports the visualization of the planning process.

The PRODIGY planning system can behave in a number of planning modes: generative planning, analogical (case-based) planning, conditional planning, and hierarchical planning. To use the UI, the user simply chooses a planning mode (the default is generative planning), loads a domain and a problem, and then runs the planner (note that if in analogical planning mode, the user must retrieve a past case before execution). The following document will explain these processes and how the UI supports them in general detail.

Description of the UI Main Window Layout

The main window of the interface is divided into a number of major components. Along the top of the main window is the menu bar from which the user controls the functioning of the system. Pull-down menus are accessed from each menu bar entry by clicking on the desired location with the left mouse button. The menu bar items include File operations, Planning Mode selections, Control Variable changes, Display Variable changes, and View selections. Each category is further described in some detail below.

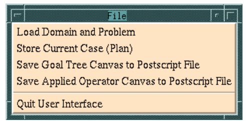
Along the left side of the main window is a series of "command buttons." These buttons represent the most commonly used commands and are placed for quick access. Some of the functions that they activate can also be selected from pull-down menus as well. The Run button executes the planner using the current planning mode. The Step button executes the planner incrementally, running one or more steps and then pausing. The Break button interrupts execution after it has begun. At either a step pause or at a break, the Continue button non-incrementally resumes the planner's execution at the current point; whereas, the Abort button clears the planner and places it in a reinitialized state.

Along the very bottom of the main window is a information row. Two sunken boxes display the names of the domain and problem that is currently loaded into the system. The remainder of the row will display descriptions of various window entries as the mouse passes across them.

Mode-specific buttons can also appear along the bottom of the main menu (immediately above the information row) depending on the planning mode PRODIGY is placed. Note, however, that the default planning mode has no such row of buttons. Instead, the row will appear and the main window will adjust its size as the user changes modes using the Planning Mode pull-down menu.

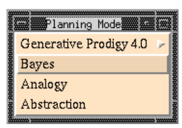
Two sub-windows with scroll bars exist in the center of the main window. The left most window is the Goal Tree Display. The right-most window is the Plan Display window. The first window displays the tree of goal/sub-goal relations rooted in the top-level planning goals provided as input to PRODIGY. Blue-colored nodes in the tree are goal states and the black-colored nodes are operator nodes. Therefore, an operator is connected above to its primary effect; whereas, it is connected below to its preconditions. Red numbers next to operator nodes refer to the order of the corresponding plan step in the Plan display. Each node in the Goal Tree Display and each step in the Plan Display is mouse sensitive. Clicking with the mouse provides access to further information and internal data structures. This access is explained below in the section below entitled "Viewing Plan and Domain Objects."

File Operations



The files operations are simple commands for retrieving problems from and saving solutions to disk and for saving either Display window to a postscript file. As convention will have it, the user can also quit from this pull-down menu.

Planning Modes



The UI supports a number of planning modes. They are selected from the Planning Mode pull-down menu.

Generative Planning - Prodigy 4.0

Prodigy 4.0 is a generative planner that creates plans “from scratch” by traversing the search space. The user loads a domain and then a problem by selecting the “Load” button from either the “File” menu or from the “Load” command button on the left panel of the main UI window. Once loaded, the planner can be executed by selecting the “Run” command button. Alternatively, the user can “Step” through the planning process by choosing the “Step” command button and specifying a step number. Once a plan is created, the user can save it for later replay in analogical mode. A save is performed from the File pull-down menu.

Generative planning can be either linear or nonlinear. Nonlinear planning can be performed with either delayed or eager state changes. During planning when all of the preconditions for a given operator are true in the initial state, Prodigy has the choice of either applying the operator immediately to change the current state or continuing to plan with the current state by selecting another pending goal to achieve. The first policy is called SAVTA (Subgoal After eVery Try to Apply) and represents an eager approach. The second policy is called SABA (Subgoal Always Before Application) and represents a delayed or lazy approach. For additional details, see the section entitled "The Generative Planning Algorithm" above.

The default generative running mode is nonlinear with eager state changes (SAVTA).

Analogical Planning - Prodigy/Analogy

Under an analogical or case-based planning mode, Prodigy/Analogy plans by replaying past planning experiences (i.e., ones saved after planning under generative mode). First, the user loads a domain and then a problem. This action is required even if a problem and domain was previous loaded in another planning mode. Secondly, the user retrieves a past case or set of cases with which to guide subsequent planning. Retrieval can be performed either automatically or manually using analogy-specific buttons along the bottom of the main window. Finally once a case is retrieved, the user selects "Replay" or "Step" to execute the planning.

Sometimes no single case is sufficient for solving a new problem and multiple cases may be retrieved. In a simple example, a new problem may have two goals for which plans have only been generated independently (i.e., two cases exist in the case base, each solving one of the goals). If more than one case is retrieved, then the manner in which the cases are merged can be selected via the Merge Strategy button at the bottom of the main window (this button is an analogy-specific operation).

The strategies available to the user are as follows:

`saba`: (default) sub-goal always before applying, and apply in right order
`saba-cr`: saba and lets other control rules apply to goal choices
`user`: by asking the user
`serial`: in a pre-defined serial order
`random`: randomly
`smart`: by using a *set-of-interacting-goals*

The Help pull-down menu offers comprehensive technical details on PRODIGY under analogical mode. Select "Prodigy/Analogy Guide." See also the section entitled "The Derivational Analogy (CBR) Planning Algorithm" above.

Hierarchical Planning - Alpine

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. PRODIGY/Alpine 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. For details, see Section 9 of the PRODIGY Manual and Tutorial.

The following is an extension to the Alpine system:

The use of abstraction depends on the :use-abs-type key of the "run" function. This key may take one of the following values:

nil -- do not use abstraction
'in, 'indep, or 'independent -- use problem-independent abstraction
'sp, 'spec, 'specific, 'dep, or 'dependent -- use problem-specific one
anything else -- give warning about incorrect argument and do not use abstraction

The function generates and caches an abstraction as necessary. It automatically notices the change in the current problem and in the domain. If there is not abstraction, it generates and stores a new one. If the user defines a new operator or inference rule, or loads a new domain, the system discards the old abstraction. If the user loads a new problem, the system discards the old abstraction only if the abstraction is problem-specific.

Due to the organization of data storage, the problem space may contain only one abstraction at a time. This limitation means, unfortunately, that the generation of a problem-specific abstraction requires to delete the old problem-independent abstraction, and vice versa. Thus, if we use a problem-independent abstraction, then problem-specific, and then problem-independent again, we need to re-build the problem-independent abstraction.

We provide four functions for building and discarding abstractions, though normally the user does not have to call them, since the "run" function performs these actions automatically, as necessary. These four functions do not have any arguments:

`gen-problem-independent` generate a problem-independent abstraction; return nil if there is no current problem space or if abstraction has collapsed to a single level; return T otherwise

`gen-problem-specific` or `gen-problem-dependent` these are two different names for the same function, which generates a problem-specific abstraction, for the current problem; it returns nil if there is no current domain, no current problem, or if abstraction have collapsed to a single level; it returns T otherwise

`clear-problem-independent` discard abstraction (whether problem-specific or problem-independent); do nothing if there is no current problem space or no abstraction

`clear-problem-specific` or `clear-problem-dependent` discard abstraction if it is problem-specific; if abstraction is problem-independent, there is no abstraction, or there is no current problem space, do nothing

`show-abstraction` function for displaying current abstraction, in the ASCII text from:

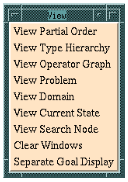
:pspace problem space (by default, current)
:new nil for showing the cached abstraction, nonnil for generating a new abstraction 1st
:goal-exp used only when :new is nonnil; nil for generating problem-independent abstraction, nonnil for generating problem-specific abstraction

Conditional Planning - B(ranching) Prodigy

Under a conditional planning mode, Prodigy creates plans containing branches. Thus when executed, the plan can be flexible in a changing environment, taking choices depending on current states in the environment at execution time. This planning variation is as yet to be fully integrated into the User Interface at the current time.

Viewing Plan and Domain Objects

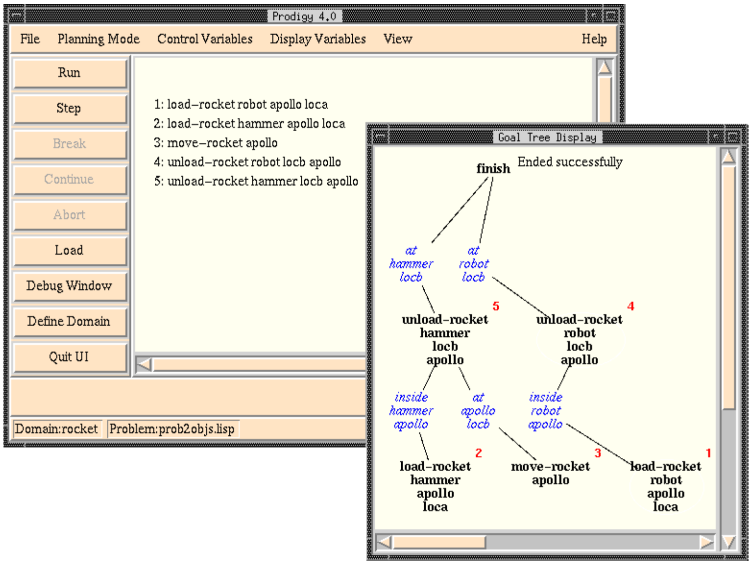
View Menu Items



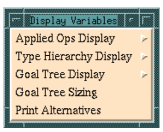
The View pull-down menu offers the user a number of functions to visualize or inspect aspects of the plan and domain objects or files associated with such. Once a problem domain is loaded, the user can view the domain or the type (abstraction) hierarchy by selecting “View Domain” or “View Type Hierarchy” from the pull-down menu. The system displays the first item in a browsing window as the domain file itself. The second is shown as a directed graph and can be saved to file as a postscript image. Once the user loads a problem, the “View Problem” selection will also become active. The user can then choose to view the problem file. Finally after a plan has been created, the remaining selections from the View pull-down menu will become active. These allow the user to view the plan’s partial order as a directed graph (PRODIGY creates a fully ordered plan that represents perhaps only one of many from the partial order). The user can also view the state of the world as it would be after the execution of the plan. This is performed by choosing “View Current State.”

The View Operator Graph displays the operator-assertion dependency graph for the currently loaded domain. A graphical representation that can be saved to postscript form indicates operators in boxes, states in ovals, and dependencies with arrows. An arrow exists from a state to an operator if that state depends on the operator; that is, if the operator adds or deletes the state. An arrow links an operator to a state when the operator requires the state as a precondition. Negative states are shown for those in the delete list of an operator, or when negative states constitute a precondition.

Search nodes can be viewed directly with the View Search Node choice, rather than having to search for them using the mouse in the display windows. The Clear Windows selection allows the user to clear the two displays of any information within them. Finally, the Goal Tree Display can be separated from the main menu as an independent graphical window by selecting Separate Goal Display. If this is performed then the Plan Display expands to fill the entire space within the main window area (see below).



Control of Viewing through Parameters



The Display Variable pull-down menu organizes a number of operations that affect the display of information in the UI. The “Applied Ops Display” selection allows the user to change the manner in which the plan is displayed in the Plan Display window. The plan can be displayed as either a partial or total order. The “Type Hierarchy Display” selection allows the user to view instances as well as types in the “View Type Hierarchy” selection of the “View” menu. The “Goal Tree Display” selection contains the option of printing polygons around nodes in the Goal Tree graph; whereas, the “Goal Tree Sizing” selection presents the parameters that affect the spacing of nodes in the Goal Tree. Finally, the “Print Alternatives” affects the display of the planning trace in the Lisp listener. Alternatives are printed when it is set to true.

The Mouse Access from the Plan Displays

Both the Goal Tree Display and the Plan Display are mouse sensitive. All of the objects displayed within these two Tk canvases can be inspected with the left button. The middle button can be used to drag the viewing area within the window in lieu of the scroll-bar mechanism.

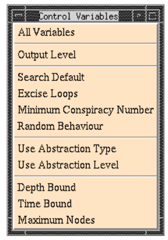
Each node of the Goal Tree represents either a (sub)goal or an operator bound to particular objects and states in the world, while the steps listed in the Plan Display represent an instantiated operator. For each of these objects, the left mouse button activates a display for the search tree node that contains information regarding the planning decision made by PRODIGY when constructing that object. That is, the left mouse button displays the goal node, the operator binding node, or the instantiated operator node of the internal search tree.

For each search-tree node displayed, a button is presented that connects with the parent of the node, and a button is presented for each child of the node. Clicking one of these buttons creates a display of the corresponding search tree node and the previously displayed node is removed. Multiple nodes can be displayed, however, by clicking on multiple objects in the Goal Tree Display and the Plan Display. (Alternative, if the user knows a node number, he can access it directly through the "View Search Node" entry of the View pull-down menu.) Finally, a button is presented for the node's plist. Clicking on this button generates a subwindow to display all properties on the plist.

Within the display of the search-tree node, the system displays critical elements that affect the respective planning decision. Each object in this display is also mouse sensitive so that the user can recursively explore all of the data structures that bear on a given planning decision. For example, an applied-operator node includes the instantiated-op structure, a list of application structures (a given instantiated operator may have been applied a number of times with backup before a successful sequence is generated), and for each application structure, a list of state deletions and additions (operator add list and delete list).

A final feature to mention is that for operator binding nodes in the Goal Tree Display, the middle mouse button will display the operator definition from the domain file that corresponds to that node.

Program Parameters



The user can use the Control Variables pull-down menu to change various program parameters of the PRODIGY system. These correspond to "run" command keyword parameters and the planner flags in the manual version of PRODIGY (see section 6.4 in the Prodigy 4.0 Manual and Tutorial available from the Help pull-down menu; i.e., Carbonell, et al., 1992).

Output Level (value: 0, 1, 2 or 3, defaults to 2) This controls how much trace information (in the Lisp listener) is printed out during a run. 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.

Search Default This can be used to switch between depth-first and breadth-first search strategies for problem solving.

Excise Loops (values: true or false, default is true) If this is not false, dependency-directed backtracking will be used to try to cut down the search space. See Section 8 of the PRODIGY Manual and Tutorial (Carbonell, et al., 1992) for more detail about dependency-directed backtracking.

Minimum Conspiracy Number (values: true or false, default is true) If this is not false, 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.

Abstraction Type (values: independent, dependent, default is nil) Available only under Hierarchical Planning mode. If this is either dependent or independent, Prodigy/Alpine will use the specified type of abstraction hierarchy (i.e., either a problem-specific or problem-independent hierarchy respectively).

Use Abstraction Level (values: true or false, default is true) If this is not false, 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: true or false, default is false) If this is not false, 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.

Depth Bound (values: nil or an integer, default is 50) 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.

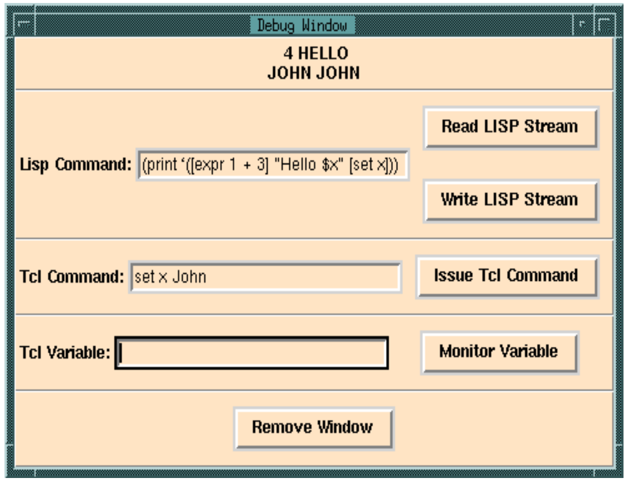
Time Bound (value: a number or nil, defaults to nil) If this is a number, then it limits the maximum run time in seconds that will be allowed before the problem solving is stopped. Nil means that there is no time bound.

Maximum 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.

Debugging

From the command buttons on the left of the main UI window, the user can select the Debug Window. This action allows a secondary window to be displayed that has a number of features. At the top of the window, the system displays the value of an internal variable called debug_msg (which is initially set to an informative message). The next window panel contains an entry point for reading and writing to the underlying Lisp steam. The following panel allows the user to input Tcl/Tk commands. The final panel allows the user to specify global variables for the system to monitor.

If reprogramming the User Interface, the user can write to the debug_msg variable in order to display intermediate results of Tcl/Tk code. These results are automatically printed at the top of the



Debug Window. Note, however, that results of the “Read LISP stream” and “Issue Tcl Command” buttons are also displayed in this location. So as an alternative, the user can declare a separate global variable, enter the variable identifier in the “Monitor Variable” selection, and the results of changes to the variable will be automatically printed at the top of the window (just below the value of debug_msg).

One of the useful attributes of the “Lisp Command” feature is the ability to mix Tcl/Tk instructions with the Lisp instructions. The system will interpret the results appropriately. Beware however, that Lisp and Tcl interpret types such as strings in different manners. For example, if the user first issues the Tcl command “set x John”, and then at Lisp Command entry box, the user types (print ‘([expr 1 + 3] “Hello $x” [set x])), a write to the Lisp stream will print the list (4 HELLO JOHN JOHN). Reading from the Lisp stream in the Debug Window will produce 4 HELLO JOHN JOHN at the top of the window. However, if the user enters (print (list ([expr 1 + 3] “Hello $x” [set x]))) instead, then Lisp generates an error “Attempt to take the value of the unbound variable `HELLO’.” Quoting both the string and the set command generates no Lisp error. Yet the output is distinctly different. Lisp prints (4 “Hello John” JOHN). Tcl prints 4 {“Hello John”} JOHN

The facilities of the debug window are extremely useful, but the programmer should be acutely aware of the semantics of both languages. As one final hint, if the UI freezes, it may actually be waiting for input. For instance, in the example above, the user may have clicked on the “Read Lisp Stream” button once too many times. In such cases, issue a call to the Lisp function ping to send to Tcl something to input.

UI Customization

A number of way exist to customize the User Interface to PRODIGY. The ambitious programmer can of course reprogram the interface since the source code is publicly available. As explained above, the Debug Window can assist with this kind of change. This section explains two less severe ways to change the UI. The user can include custom Tcl/Tk code in two customization files that will overwrite existing variable assignments and procedure definitions. We will call this “overload customization.” The user can also modify the values of program parameters and global variables. We will call this “parameter customization.”

In the file ui.lisp, two special Lisp variables exist that should be changed by the user to incorporate changes to the UI. The variable *tcl-customizations* contains a string specifying a file with tcl code meant to alter program variables such as uiColor (i.e., institutes parameter customization). It is loaded after the file ui-start.tcl is loaded. Ui-start contains most of the program parameters that are used by the remaining system files. By providing *tcl-customizations* a string to a file that alters the program parameters, you can affect the behavior of the program in significant ways. An example of parameter customization is presented the file example-param-custom.tcl.

As an aside, note that a variable called stackedCanvases can be set in example-param-custom.tcl by uncommenting (remove the # symbol) the line that references it. If this is done, the system will arrange the goal-tree and plan displays on top of each other, rather than side-by-side.

A more significant change to the interface occurs when a programmer alters the code itself. A safe way to do this without affecting other users that may use the interface in your environment, is to create a file with code that overrides the functions in the UI files or adds new functions. Then by providing a string that specifies the file name for the value of the global variable *post-tcl-customizations* (also found within file ui.lisp) this code will be interpreted after the rest of the system code is loaded.

NOTE that the full file loading sequence is enumerated in the Help pull-down menu.

Appendix B: The One-Way Rocket Domain

```
;;; This is the rocket problem, using identical operators to nolimit's
;;; in order to make comparison easy.

(create-problem-space 'nolimit-rocket :current t)

(ptype-of OBJECT    :top-type)
(ptype-of LOCATION  :top-type)
(ptype-of ROCKET    :top-type)


(pinstance-of LocA LocB LOCATION)

(Operator Load-Rocket
 (params <object> <rocket> <place>)
 (preconds
  ((<object> OBJECT)
   (<place> LOCATION)
   (<rocket> ROCKET))
  (and (at <rocket> <place>)
       (at <object> <place>)))
 (effects
  ()
  ((del (at <object> <place>))
   (add (inside <object> <rocket>)))))

(Operator
 Unload-Rocket
 (params <object> <place> <rocket>)
 (preconds
  ((<object> OBJECT)
   (<place>  LOCATION)
   (<rocket> ROCKET))
  (and (inside <object> <rocket>)
       (at <rocket> <place>)))
 (effects
  ()
  ((del (inside <object> <rocket>))
   (add (at <object> <place>)))))

(Operator
 Move-Rocket
 (params <rocket>)
 (preconds
  ((<rocket> ROCKET))
  (at <rocket> LocA))
 (effects
  ()
  ((del (at <rocket> LocA))
   (add (at <rocket> LocB)))))
```

```
(setf *class-short-names*
 .  . `(
 .  . (OBJECT . o)
 .  . (LOCATION . l)
 .  . (ROCKET . r)))

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

(if *analogical-replay*
    (load (concatenate
 .  . 'string
 .  . *analogy-pathname*
 .  . "replay-crs")))
```

Appendix C

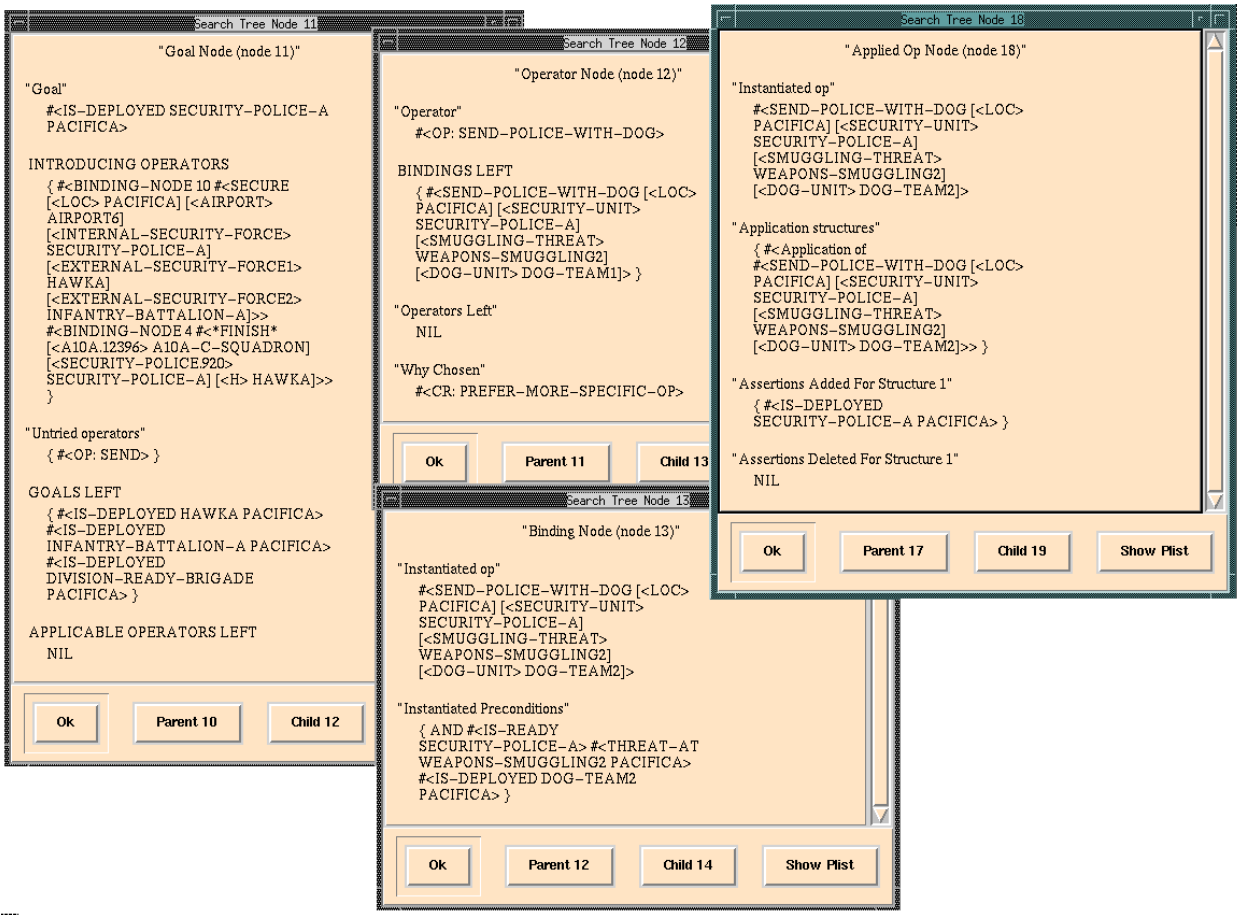


Figure 11. The full decision cycle

References

Blythe, J. (1996). Event-based decompositions for reasoning about external change in planners. In *Proceedings of Third International Conference on AI Planning Systems (AIPS-96)*. Menlo Park, CA: AAAI Press.

Blythe, J. & Veloso, M. M. (1997). Analogical Replay for Efficient Conditional Planning. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference* (pp. 668-673). Menlo Park, CA: AAAI Press.

Blythe, J., Veloso, M. M., & de Souza, L. (1997). *The Prodigy user interface* (Tech. Rep. No. CMU-CS-97-114). Pittsburgh: Carnegie Mellon University, Department of Computer Science.

Carbonell, J. G. (1986). Derivational analogy: A theory of reconstructive problem solving and expertise acquisition. In R. Michalski, J. Carbonell & T. Mitchell (Eds.), *Machine learning: An artificial intelligence approach, Vol. 2* (pp. 371-392). San Mateo, CA: Morgan Kaufmann.

Carbonell, J. G., Blythe, J., Etzioni, O., Gil, Y., Joseph, R., Kahn, D., Knoblock, C., Minton, S., Pérez, A., Reilly, S., Veloso, M. M., & Wang, X. (1992). *PRODIGY4.0: The manual and tutorial* (Tech. Rep., CMU-CS-92-150). Dept. of Computer Science, Carnegie Mellon University.

Carbonell, J. G., Knoblock, C. A., & Minton, S. (1991). PRODIGY: An integrated architecture for planning and learning. In K. VanLehn (Ed.), *Architectures for intelligence: The 22nd Carnegie Mellon symposium on cognition* (pp. 241-278). Hillsdale, NJ: Lawrence Erlbaum Associates.

Cox, M. T. (1996). Introspective multistrategy learning: Constructing a learning strategy under reasoning failure. Doctoral dissertation, Tech. Rep. No. GIT-CC-96-06, Georgia Institute of Technology, College of Computing, Atlanta. (Available at URL ftp://ftp.cc.gatech.edu/pub/ai/ram/git-cc-96-06.html)

Cox, M. T., & Veloso, M. M. (1997). *Controlling for unexpected goals when planning in a mixed-initiative setting*. To appear in *Proceedings of the Eighth Portuguese Conference on Artificial Intelligence*.

Ferguson, G., Allen, J. F., & Miller, B. (1996). TRAINS-95: Towards a mixed-initiative planning assistant. In *Proceedings of the Third International Conference on AI Planning Systems (AIPS-96)*, Edinburgh, Scotland, May 29-31, 1996.

Fink, E., & Veloso, M. M. (1996). Formalizing the Prodigy planning algorithm. In M. Gahllab & A. Milani (Eds.), *New Directions in AI Planning* (pp. 261-271). Amsterdam: IOS Press

Gansner, E. R., North, S. C., & Vo, K. P. (1988). DAG - A program to draw directed graphs. *Software Practice and Experience, 17*(1): 1047-1062.

Hammond, K. J. (1989). *Case-based planning: Viewing planning as a memory task. Vol. 1.* of *Perspectives in artificial intelligence*. San Diego, CA: Academic Press.

Kolodner, J. L. (1993). *Case-based reasoning*. San Mateo, CA: Morgan Kaufmann Publishers.

Oates, T. & Cohen, P. R. (1994). Toward a plan steering agent: Experiments with schedule maintenance. In *Proceedings of the Second International Conference on Planning Systems (AIPS-94)* (pp. 134-139). Menlo Park, CA: AAAI Press.

Riesbeck, C. K., & Schank, R. C. (1989). *Inside case-based reasoning* (pp. 1-24). Hillsdale, NJ: Lawrence Erlbaum Associates.

Seifert, C. M., Hammond, K. J., Johnson, H. M., Converse, T. M., McDougal, T. F., & Vanderstoep, S. W. (1994). Case-based learning: Predictive features in indexing. *Machine Learning, 16*, 37-56.

Veloso, M. (1994). *Planning and learning by analogical reasoning.* Springer-Verlag.

Veloso, M. (1997). *Merge Strategies for Multiple Case Plan Replay.* In D. B. Leake & E. Plaza (Eds.), *Case-Based Reasoning Research and Development: Second International Conference on Case-Based Reasoning* (pp. 413-424). Berlin: Springer-Verlag.

Veloso, M., & Carbonell, J. G. (1993). Towards scaling up machine learning: A case study with derivational analogy in PRODIGY (pp. 233-272). In S. Minton (Ed.), *Machine learning methods for planning.* Morgan Kaufmann.

Veloso, M. M., Mulvehill, A. M., & Cox, M. T. (1997). Rationale-supported mixed-initiative case-based planning. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference* (pp. 1072-1077). Menlo Park, CA: AAAI Press / The MIT Press.