PRODIGY: An Integrated Architecture for Planning and Learning

Jaime Carbonell, Oren Etzioni*, Yolanda Gil, Robert Joseph
Craig Knoblock, Steve Minton†, and Manuela Veloso

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

†NASA Ames Research Center
AI Research Branch, Mail Stop: 244-17
Moffett Field, CA 94035

*Department of Computer Science
University of Washington
Seattle, WA 98195

ABSTRACT

Artificial intelligence has progressed to the point where multiple cognitive capabilities are being integrated into computational architectures, such as SOAR, PRODIGY, THEO, and ICARUS. This paper reports on the PRODIGY architecture, describing its planning and problem solving capabilities and touching upon its multiple learning methods. Learning in PRODIGY occurs at all decision points and integration in PRODIGY is at the knowledge level; the learning and reasoning modules produce mutually interpretable knowledge structures. Issues in architectural design are discussed, providing a context to examine the underlying tenets of the PRODIGY architecture.

1 Introduction

A common dream for many AI researches, present authors included, is the construction of 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, implemented in PRODIGY [2], 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. The paper is divided into two parts. The first part describes the basic architecture, including the problem solver and the various learning modules. The second part discusses the design issues in building an integrated architecture.

2 The PRODIGY Architecture

2.1 The Problem Solver

PRODIGY's basic reasoning engine is a general-purpose problem solver and planner [10] that searches for sequences of operators (i.e., plans) to accomplish a set of goals from a specified initial state description. Search in PRODIGY is guided by a set of *control rules* that apply at each decision point. Search control rules may be general or domain specific, hand-coded or automatically acquired, and may consist of heuristic preferences or definitive selections. In the absence of any search control, PRODIGY defaults to depth-first means-ends analysis. But, with appropriate search control knowledge it can emulate other search disciplines, including breath-first search, depth-first iterative-deepening, best-first search, and knowledge-based plan instantiation.

2.2 Knowledge Representation

Each operator has a precondition expression that must be satisfied before the operator can be applied, and a list of effects that describe how the application of the operator changes the world. Precondition expressions are well-formed formulas in a form of predicate logic encompassing negation, conjunction, disjunction, and existential and universal quantification. The effects are atomic formulas that describe the facts that are added or deleted from the current state when the operator is applied. Operators may also contain conditional effects, which represent changes to the world that are dependent on the state in which the operator is applied.

2.3 Problem Definition and Problem Solving

A problem consists of an initial state and a goal expression. To solve a problem, PRODIGY must find a sequence of operators that, if applied to the initial state, produces a final state satisfying the goal expression. The search tree initially starts out as a single node containing the initial state and goal expression. The tree is expanded by repeating the following two steps:

1. **Decision phase:** There are four types of decisions that PRODIGY makes during problem solving. First, it must decide what node in the search tree to expand next, defaulting to a depth-first expansion. Each node consists of a set of goals and a state describing the world. After a node has been selected, one of the node's goals must be selected, and then an operator relevant to this goal must be chosen. Finally, a set of bindings for the parameters of that operator must be decided upon.

2. **Expansion phase:** If the instantiated operator's preconditions are satisfied, the operator is applied. Otherwise, PRODIGY subgoals on the unmatched preconditions. In either case, a new node is created with updated information about the state or the subgoals.

The search terminates after creating a node whose state satisfies the top-level goal expression.

2.4 Control Rules

As PRODIGY attempts to solve a problem, it must make decisions about which node to expand, which goal to work on, which operator to apply, and which objects to use. These decisions can be influenced by control rules to increase the efficiency of the problem solver's search and to improve the quality of the solutions that are found.

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 a quick, arbitrary choice. If in fact the wrong choice is made, and costly backtracking proves necessary, an attempt will be made to learn the control knowledge that must be missing. The rationale for PRODIGY's *casual commitment* strategy is that for any

decision with significant ramifications, control rules should be present; if they are not, the problem solver should not attempt to be clever without knowledge, rather, the cleverness should come about as a result of learning. Thus, our emphasis is on an elegant and simple problem-solving architecture which can produce sophisticated behavior by learning the appropriate, domain-specific control knowledge.

Control rules can be employed to guide the four decisions described in Section 2.3. Each control rule has a left-hand side condition testing applicability and a right-hand side indicating whether to SELECT, REJECT, or PREFER a particular candidate. To make a control decision, given a default set of candidates (nodes, goals, operators, or bindings, depending on the decision), PRODIGY first applies the applicable selection rules to select a subset of the candidates. If no selection rules are applicable, all the candidates are included. Next rejection rules further filter this set by explicit elimination of particular remaining candidates, and finally preference rules are used to find the most preferred alternative. If backtracking is necessary, the next most preferred candidate is attempted, and so on, until a global solution is found, or until all selected and non-rejected candidates are exhausted.

2.5 The Learning Modules

PRODIGY's general problem solver is combined with several learning modules. The PRODIGY architecture was designed both as a unified testbed for different learning methods and as a general architecture to solve interesting problems in complex task domains. Let us now focus on the architecture itself, as diagrammed in Figure 1.

The problem solver produces a complete search tree, encapsulating all decisions – right ones and wrong ones – as well as the final solution. This information is used by each learning component in different ways. In addition to the central problem solver, PRODIGY has the following learning components:

APPRENTICE: A user interface that can participate in an apprentice-like dialogue [5], enabling the user to evaluate and guide the system's problem solving and learning. The interface is graphic-based and tied directly to the problem solver, so that it can both acquire domain knowledge or accept advice as it is solving a problem.

EBL: An explanation-based learning facility [9] 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 produces control rules without utilizing any training examples. STATIC can be viewed as a compiler for PRODIGY's domains [4]. In experimental tests, STATIC produced control knowledge that was superior to that of EBL and did so one to two orders of magnitude faster. However, not all problem spaces permit purely static learning, requiring EBL's dynamic capabilities. STATIC's design is based on a detailed predictive theory of EBL, which is described in [3].

ANALOGY: A derivational analogy engine [13] 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. Both analogy and EBL are independent mechanisms to acquire domain-specific control knowledge.

ALPINE: An abstraction learning and planning module [6]. 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 in an abstract space and then uses the abstract solution to guide the search for solutions in more detailed problem spaces. This method is orthogonal to analogy and EBL, in that both can apply at each level of abstraction.

EXPERIMENT: A learning-by-experimentation module for refining domain knowledge that is incompletely or incorrectly specified [1]. 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 control knowledge.

All of the learning modules are loosely integrated in that they are all given the same domain definition and share the same basic problem solver and data structures. The knowledge learned by each module is then incorporated back into the knowledge base, but this knowledge is not yet fully exploited by the other modules.

We are currently investigating tighter integration of the learning modules. For example, the EBL and abstraction modules can be combined such that the control rule learning is applied within each abstraction space. This simplifies the learning process and results in more general control rules since the proofs in an abstract space contain fewer details [7]. Similarly, abstraction and analogy can be integrated by applying analogy in the abstract problem spaces. The analogy module will then learn skeletal plans, which should apply in a wider variety of situations.

3 Dimensions of the Architecture

We now characterize PRODIGY along some of the proposed dimensions.

Generality: PRODIGY has been applied to a wide range of planning and problem-solving tasks: robotic path planning, the blocksworld, an augmented version of the STRIPS domain, matrix algebra manipulation, discrete machine-shop planning and scheduling, process planning, computer configuration, logistics planning, and several others.

Versatility: Because of the modularity of PRODIGY's structure, it can support different types of tasks, goals, and methods. As a simple example, the clear separation between domain and control knowledge enables PRODIGY to employ different search strategies, e.g. breadth-first, depth-first, or best-first search. The modularity of PRODIGY permits extensions with additional learning

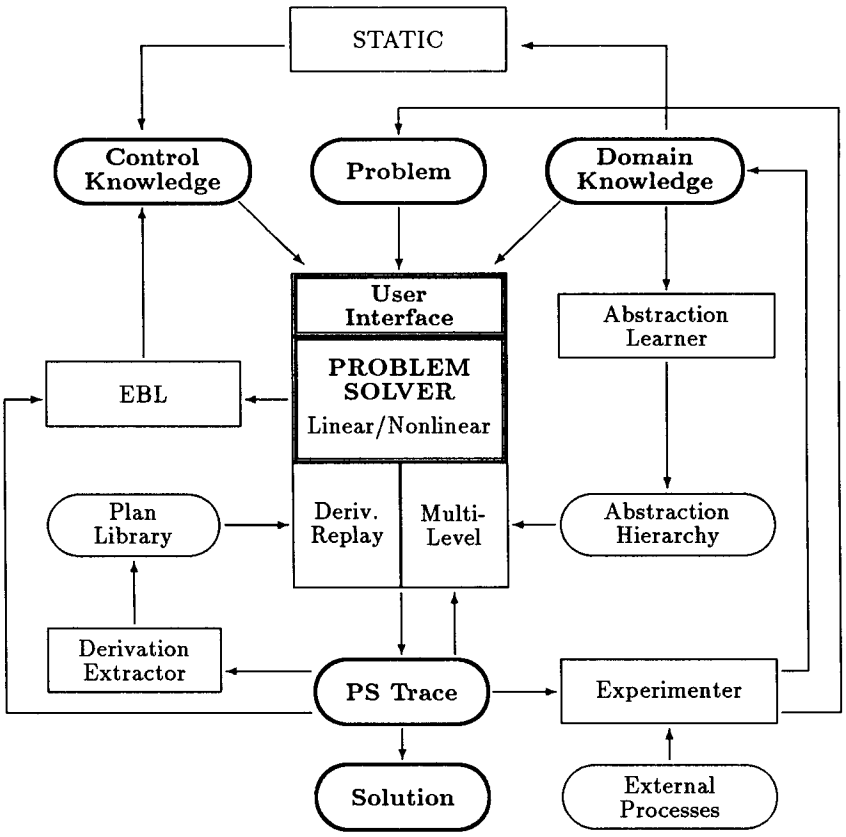

Figure 1: The PRODIGY Architecture

or task modules. The integration focuses on establishing the channels of communication among the different modules.

Rationality: The reasoning cycle of the general problem solver of PRODIGY assumes that the domain specification is correct and complete. Solutions produced to given problems are therefore always consistent with PRODIGY's knowledge and goals. The experimentation work in progress relaxes the assumption that the domain knowledge is correct and complete and allows the architecture to show increasing rationality by refining its knowledge through interaction with a simulated outside world.

Ability to add new knowledge: The modularity of the domain and control languages in PRODIGY make it easy to add new domain and control knowledge. PRODIGY also provides a powerful user interface that simplifies the task of debugging and refining both domain and control knowledge.

Ability to learn: The different learning modules incorporate new knowledge into PRODIGY. The knowledge includes control knowledge in the form of control rules (EBL and STATIC), abstraction hierarchies (ALPINE), or past problem solving episodes (ANALOGY). Knowledge acquisition (APPRENTICE) and experimentation (EXPERIMENT) techniques refine and extend the domain definition.

Scalability: Any integrated architecture must address increasingly large tasks, whether its objective is to model human cognition or to build useful knowledge-based systems for complex tasks. Scalability can be calibrated in multiple ways, but all relate to efficient behavior with increasing complexity, as measured by:

- Size of the domain: total number of objects, attributes, relations, operators, inference rules, etc.
- Size of the problem: number of steps in the solution plan, number of conjuncts in the goal expression, size of the visited search space, etc.
- Variety: number of qualitatively different actions and object types in the domain.
- Perplexity: average fan-out at every decision point in the search space (with and without learned control knowledge).

In PRODIGY we seek to achieve a reasonable measure of scalability in all these dimensions. The learning techniques strive to reduce the visited search space in future problems with respect to the virtual (complete) search space.

Reactivity: Within the class of deliberative reasoning, one can distinguish real-time decision making and long-term planning. PRODIGY models only deliberative planning and problem-solving, albeit in resource-limited domains.

Efficiency: As in any planning system the search space is highly explosive. The purpose of most of the learning modules is to improve the efficiency of the system.

Psychological Validity:
As mentioned earlier, the PRODIGY project strives to produce a useful, scalable, and maintainable reasoning and learning architecture. Where this matches human

cognition, it is so by accident, by the limited imagination of the PRODIGY designers, or perhaps because the human mind has indeed optimized such aspects of cognition. In all other aspects, the goal is at reengineering cognition the way it ought to be, in order to be most useful in problem solving, planning, and learning. Here we enumerate a few additional ways in which PRODIGY differs from human thought and from other cognitive architectures:

- PRODIGY forgets when it chooses to do so. For instance, a control rule whose testing and application overhead is greater than the search reduction benefits accrued over time may be discarded. Minton [9] demonstrated that the effectiveness of explanation-based learning is improved by measuring the utility of acquired knowledge and retaining only those rules with positive utility.
- PRODIGY deliberates on any and all decisions: which goal to work on next, which operator to apply, what objects to apply the operator to, where to backtrack given local failure, whether to remember newly acquired knowledge, whether to refine an operator that makes inaccurate predictions, and so on. It can introspect fully into its decision cycle and thus modify it at will. This is not consistent with the human mind, yet it is an extremely useful faculty for rapid learning.
- PRODIGY's knowledge acquired in one module is open to inspection and interpretation by other modules. Abstracted operators can be used to plan, to drive EBL, to analogize with past memory, and so forth. The compartmentalization is at the level of learning methods, and the sharing is at the level of all knowledge acquired.

4 Comparison with Other Architectures

There are multiple dimensions one can use to contrast and compare different integrated architectures. We list each dimension, situating PRODIGY and contrasting it to SOAR [12], THEO [11], and occasionally ICARUS [8]. Each dimension in the design space addresses a major component of the architecture:

Central Problem Solver - Each architecture relies on problem solving. Whereas THEO has no general problem solver (it is guided by the acquisition of domain-specific methods, overriding more general ones), PRODIGY and SOAR each have an architecturally defined general problem solver whose performance improves incrementally through the acquisition of factual and control knowledge for each domain.

Deliberative vs. Reflexive Learning - PRODIGY acquires new knowledge only when it believes that knowledge will be useful; learning is a deliberate meta-reasoning process. SOAR, on the other hand, cannot help but learn; chunking is a reflex process in the architecture. THEO's caching mechanism is closer to SOAR's chunking, but more recent developments indicate some motion towards the PRODIGY philosophy. ICARUS grows its decision trees as an incremental reflex process, much like SOAR's chunking philosophy.

Multiple vs. Single Learning Strategies - PRODIGY employs multiple learning strategies: explanation-based learning, analogy, abstraction, experimentation, static analysis, tutoring, and so on. SOAR employs only chunking, through which it attempts to emulate some of the other learning methods. THEO takes an intermediate position with two learning mechanisms: caching and explanation-based learning.

Inductive vs. Analytical Learning - Only ICARUS employs purely inductive learning techniques. PRODIGY, SOAR and THEO combine both, relying more heavily on the EBL-like deductive methods for acquiring control knowledge. In PRODIGY, induction is employed in the experimentation techniques and in extensions to purely deductive derivational analogy.

Modular vs. Monolithic Architecture - We do not know whether the human mind compartmentalizes distinct cognitive abilities, or distinct methods of achieving similar ends (such as our multiple learning methods). However, modularization at this level is a sound engineering principle, and therefore we have adhered to it. Integration is brought about by sharing both a uniform knowledge representation, and a common problem-solving and planning engine. Other systems, such as SOAR, take on a monolithic structure. There is only one learning mechanism, chunking, which can never be turned off or even modulated. Which is better? Clearly, we believe the former to be superior – but only from engineering principles rather than psychological ones.

Acknowledgements

The authors gratefully acknowledge the contributions of the other members of the PRODIGY project: Jim Blythe, Daniel Borrajo, Jose Brustoloni, Dan Kahn, Dan Kuokka, Alicia Perez, William Reilly, Santiago Rementeria, and Xuemei Wang. Other research directions in PRODIGY include meta reasoning, multi-agent planning, and interactive fiction planning.

This research was sponsored in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, Amendment 20, under contract number F33615-87-C-1499, monitored by the Air Force Avionics Laboratory, in part by the Office of Naval Research under contracts N00014-84-K-0345 (N91) and N00014-86-K-0678-N123, in part by NASA under contract NCC 2-463, in part by the Army Research Institute under contract MDA903-85-C-0324, and in part by small contributions from private institutions. 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 DARPA, ONR, NASA, ARI, or the US Government.

References

[1] J. G. Carbonell and Y. Gil. Learning by experimentation: The operator refinement method. In *Machine Learning, An Artificial Intelligence Approach, Volume III*, pages 191–213. Morgan Kaufman, San Mateo, CA, 1990.

[2] Jaime G. Carbonell, Craig A. Knoblock, and Steven Minton. PRODIGY: An integrated architecture for planning and learning. In Kurt VanLehn, editor, *Architectures for Intelligence*. Erlbaum, Hillsdale, NJ, 1990. Available as Technical Report CMU-CS-89-189.

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

[4] Oren Etzioni. STATIC: A problem-space compiler for PRODIGY. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, Anaheim, CA, 1991.

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

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

[7] Craig A. Knoblock, Steven Minton, and Oren Etzioni. Integrating abstraction and explanation-based learning in PRODIGY. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, Anaheim, CA, 1991.

[8] Pat Langley, Kevin Thompson, Wayne Iba, John H. Gennari, and John A. Allen. An integrated cognitive architecture for autonomous agents. Technical Report 89-28, Department of Information and Computer Science, University of California, Irvine, 1989.

[9] Steven Minton. *Learning Effective Search Control Knowledge: An Explanation-Based Approach.* PhD thesis, Computer Science Department, Carnegie Mellon University, 1988.

[10] Steven Minton, Jaime G. Carbonell, Craig A. Knoblock, Daniel R. Kuokka, Oren Etzioni, and Yolanda Gil. Explanation-based learning: A problem solving perspective. *Artificial Intelligence*, 40(1-3):63–118, 1989.

[11] Tom M. Mitchell, John Allen, Prasad Chalasani, John Cheng, Oren Etzioni, Marc Ringuette, and Jeffrey C. Schlimmer. Theo: A framework for self-improving systems. In Kurt VanLehn, editor, *Architectures for Intelligence.* Erlbaum, Hillsdale, NJ, 1990.

[12] Paul S. Rosenbloom, Allen Newell, and John 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.

[13] Manuela M. Veloso and Jaime G. Carbonell. Integrating analogy into a general problem-solving architecture. In *Intelligent Systems.* Ellis Horwood Limited, West Sussex, England, 1990.