

Softbots as Testbeds for Machine Learning

Oren Etzioni and Richard Segal*

Department of Computer Science and Engineering, FR-35

University of Washington

Seattle, WA 98195

etzioni@cs.washington.edu

segal@cs.washington.edu

Abstract

“If you want to think about thinking, you have to think about thinking about something.”

— Seymour Papert

A softbot (*software robot*) is a program that interacts with a software environment by issuing commands and interpreting the environment's feedback. Because softbots are much easier to build than physical robots, softbots are an attractive substrate for machine-learning and agent-architecture research. To illustrate this point, we consider Rodney, a UNIX¹ softbot under development at the University of Washington. Rodney uses PRODIGY as its planner and the THEO frame system to represent its world model. Custom-built execution and sensing modules allow Rodney to interact with a UNIX shell in real time.

Motivation

Much of the recent work in machine learning and planning has been influenced by two themes:

- *integrated architectures*: focusing on programs that combine planning, learning, and execution capabilities (e.g. [Laird and Rosenbloom, 1990, Mitchell, 1990, Sutton, 1990].)
- *realistic tasks*: moving beyond the classical “toy worlds” such as the Blocksworld and the Tower of Hanoi to a variety of tasks such as manufacturing, scheduling, database management, robotics, and others [Sycara, 1990].

This trend is positive to the extent that it leads us to focus on real problems and to build systems that are more likely to scale to real-world tasks. This trend also has negative aspects. The amount of preliminary groundwork (before any machine learning research

takes place!) necessary to build an integrated system that tackles a fairly realistic task can be staggering. Furthermore, the cost (in both time and money) of maintaining and experimenting with such systems is high. Finally, the degree of realism ultimately achieved is questionable in many cases. The recent focus on learning robots (e.g. [Maes and Brooks, 1990, Mitchell, 1990, Tan, 1991]) is a good example.

Learning Robots

In principle, learning robots offer excellent testbeds for machine learning and agent-architecture research. In practice, building robust systems that successfully interact with an unpredictable physical environment is a rigorous challenge, given existing technology [Brooks, 1991a]. The monetary cost of such robots (including laser range finders, sonars, grippers, television cameras, etc.) is non-trivial, and the effort and expertise required to assemble and operate such an apparatus are considerable. For example, building an operational prototype of a single Mars Rover leg took over a year and cost over a million dollars [Simmons, 1991]. Clearly, *only a limited number of labs can afford to study robots on this scale.*

Even in these labs, conducting experiments is often time-consuming and difficult. Experiments are frequently hampered by a wide variety of hardware difficulties and malfunctions [Brooks, 1991b, Tan, 1991]. Days and even weeks go by in which the robot is not operational. Even when the robot is operational, the mean time between failures can be as short as fifteen minutes [Brooks, 1991a]. As a result, *carrying out empirical machine-learning research using robots is, per force, slow.*

Furthermore, while robotic task-environments are much more realistic than the Blocksworld, introducing the problems of sensing, uncertainty, and noise, the environments often remain highly unrealistic due to hardware limitations. Fetching cups, gripping geometric shapes, and avoiding walls are examples of typical tasks being studied by machine-learning researchers. Much more realistic robots have been built, of course, but

*To appear in the Proceedings of the 1992 AAAI Spring Symposium on Knowledge Assimilation.

¹UNIX is a trademark of AT&T Bell Labs.

they require orders of magnitude *more* investment of time, money, and expertise in robotics before machine-learning research can take place.

Simon [1983] defines learning (roughly) as improvement in performance. Our point, in general terms, is that *the cost and difficulty of building and experimenting with a performance system can impede machine-learning research*, resulting in a conflict between the desire to tackle realistic tasks, utilizing integrated agent architectures, and the desire to maintain reasonable progress in machine learning.

A solution pursued by some (e.g., [Al-Badr and Hanks, 1991, Carbonell and Hood, 1986, Philips *et al.*, 1991, Pollack and Ringuette, 1990, Russell, 1989, Vere and Bickmore, 1990]) is agent testbeds that simulate robotic environments with varying degrees of fidelity. Simulated environments have the advantage of providing relatively low cost arenas for research. However, even simulated environments can be difficult to build [Carbonell and Hood, 1986]. Furthermore, simulated worlds raise a host of thorny issues concerning the realism of the simulated worlds and the validity of the lessons learned [Al-Badr and Hanks, 1991].

Software Task-environments

In this paper we propose a solution to these problems: *building agents that interact with realistic software environments as a substrate for machine-learning and agent-architecture research*. We are not suggesting that *all* research take place using such agents, but merely that software task-environments merit exploration because they provide a relatively low-overhead arena for building integrated architectures that tackle realistic tasks. For example, an important pragmatic advantage of softbots over physical robots is survival time: the mean time between hardware failures is much higher for a workstation supporting a software environment than for a mobile robot. Rebooting a workstation and restoring the softbot “from disk” is much easier than fixing a broken gripper in a physical robot or identifying and replacing a malfunctioning chip.

We do not view software environments as idealizations of robotic environments but believe, rather, that software environments are intrinsically interesting. Writing programs that exhibit intelligent behavior in these environments is a difficult and exciting challenge in its own right. Furthermore, attempting to do so will shed light on a variety of fundamental machine-learning questions such as: How to learn from a dialogue with human instructors? How to build maps based on exploration? How to refine action models based on experience? How to learn the appropriate mix of “blind execution” and sensing for a given environment? These questions can (and should) be studied in a wide range of environments including non-physical ones.

Softbots

This section introduces *softbots* (*software robots*). A softbot is an AI program that interacts with a software environment by issuing commands and interpreting the environment’s feedback. A list of fundamental softbot characteristics follows:

1. **Software environment** — This is the distinguishing factor between softbots and robots. A robot exists in the physical world whereas a softbot exists in a software world within a computer or network of computers.
2. **Unpredictable environment** — The software environment is unpredictable and changing due to the presence of agents and processes external to the softbot. As a result, it is impossible to provide the softbot with a complete and correct model of its environment—sensing and learning are essential elements of “softbotics.”
3. **Effectors** — The effectors (or actuators) of a softbot are software commands interpreted and executed by its environment (e.g. UNIX shell commands). It might be argued that we can “invent” a softbot arbitrarily by encapsulating a component of a software environment, pointing to the commands issued by the component as its effectors. Below, we distinguish softbots from “mere” programs by enumerating softbot capabilities such as goal-directed behavior, mobility, learning, planning, and more.
4. **Sensors** — Unlike classical planners (e.g. STRIPS), a softbot cannot “read” its environment from a global data structure. Instead it has to develop a model of its environment by activating a collection of limited bandwidth sensors. Sensory actions for Rodney (the UNIX softbot) include commands such as “pwd” (what is the current directory?) and “ls” (what are the files in the current directory?).
5. **Continuous operation** — A softbot continuously operates in its environment. This constraint forces softbot designers to address problems such as surviving, co-existing with other agents, and being productive over time.

These fundamental characteristics alone do not fully distinguish softbots from computer viruses or from *knowbots* [Kahn and Cerf, 1988]² The key difference between softbots and related software is the commitment to AI capabilities inherent in the softbot paradigm. Thus, we expect softbots to have the following capabilities:

1. **Goal-directed behavior** — a softbot attempts to achieve explicit goals. Unlike a “Brooksian Creature,” it does not merely following pre-programmed

²Knowbots are agents that mediate between humans and vast knowledge stores, which have generated much attention in the popular press [Waldrop, 1990] and in the database community [Messinger *et al.*, 1991].

instincts. Thus, a human can “program” a softbot by specifying goals for it.

2. **Mobility** — Unlike most software, a softbot is mobile. For example, Rodney will be able to move from its home machine to a remote machine by logging into a the new machine, copying itself, and starting a Lisp process on the remote machine.
3. **Cloning** — Unlike most software (and all hardware), a softbot can make multiple copies of itself. This particular skill (possessed by computer viruses) is an intriguing aspect of software environments. Cloning is easy and can be critical to survival, particularly for mobile softbots.
4. **Planning, executing, and error recovery** — A softbot is able to compose the actions it knows about into sequences that, once executed, will achieve its goals. Furthermore, a softbot actually executes such sequences, monitors their progress, and attempts to recover from any unanticipated failures.
5. **Declarative knowledge representation** — A softbot stores its knowledge declaratively, allowing the same knowledge to support a variety of tasks. For example, Rodney represents the strategy of retrying a failed action as a THEO frame. This representation enables Rodney to use “retry” as a rudimentary error recovery strategy for *any* failed action.
6. **Learning and adaptation** — A softbot is able to improve its performance over time by generalizing from its experiences. For example, we would like Rodney to learn new UNIX commands, the locations of various objects (e.g. location of the Lisp executable) and the preferences of its human partners.
7. **Natural-language capabilities** — Much of the information potentially available to a softbot is encoded in natural-language (e.g. UNIX man pages). A softbot’s ability to understand its sensory inputs scales with its ability to understand natural language. Thus, a softbot requires strong natural-language capabilities.

Once a variety of softbots are built, we will need to compare and evaluate different softbots. Metrics for evaluation include: scope (what are the tasks the softbot can handle? what is the range of environments it can operate in? For example, can a UNIX softbot be used in a VMS system?), survival time (how long before the softbot is destroyed?), success rate in achieving its goals, ability to co-exist with human users, ability to adapt to a changing environment, and more.

Rodney (The UNIX Softbot)

We are in the process of building Rodney, a softbot whose environment is the UNIX operating system shell. Rodney is goal-directed and mobile. It will have

cloning, planning, and learning capabilities.³ Rodney’s effectors are the UNIX commands it knows about, and its sensors record and interpret UNIX output. To explore its environment, Rodney can execute sensory actions such as “pwd” (to find its current directory) or “ls” (to list the files in the current directory).

We have identified a variety of simple, but useful, tasks Rodney could perform including:

- Carrying out anonymous FTP, handling broken connections and protected files.
- Scanning bulletin boards for time-critical information (e.g. parties, talks, scheduled machine maintenance), issuing alerts when appropriate.
- Identifying unprotected files or identifying files that are good candidates for compression or deletion.
- Performing simple library and database searches.
- Informing naive users about UNIX. For example, suggesting “mv” instead of “cp” followed by “rm,” or piping the output of “finger” through “more” to avoid having the output of “finger” scroll off the screen.
- Detecting intruders and responding appropriately.

Learning tasks for Rodney include:

- **Map building** — building and updating a map of the directory hierarchies and file locations on multiple machines.
- **Acquiring and refining action models** — noting additional preconditions (e.g. ones involving file protections), learning about different switches and new commands that extend Rodney’s capabilities.
- **Learning execution/planning/sensing policies** — How much sensing is necessary to verify that a plan was correctly executed? Should Rodney use its sensors to check every effect of every action, or should it merely sense whether its goal has been achieved? Should Rodney create long plans and then execute them, or should it execute each step it considers immediately and “use the world as its model” (cf. [Brooks, 1991a])? The answers to these questions vary from task to task and from environment to environment, posing a set of challenging adaptation or learning problems for any versatile softbot.

Rodney’s Architecture

The spate of recent work in integrated architectures for planning, execution, and sensing has yet to produce a ready-made architecture suitable for building a softbot. Thus, we are designing our own, utilizing “pre-fabricated” software modules whenever possible. Specifically, for Rodney we are using PRODIGY [Minton

³Planning and symbolic learning capabilities distinguish Rodney from most computer viruses and from various “artificial life” programs [Langston *et al.*, 1989].

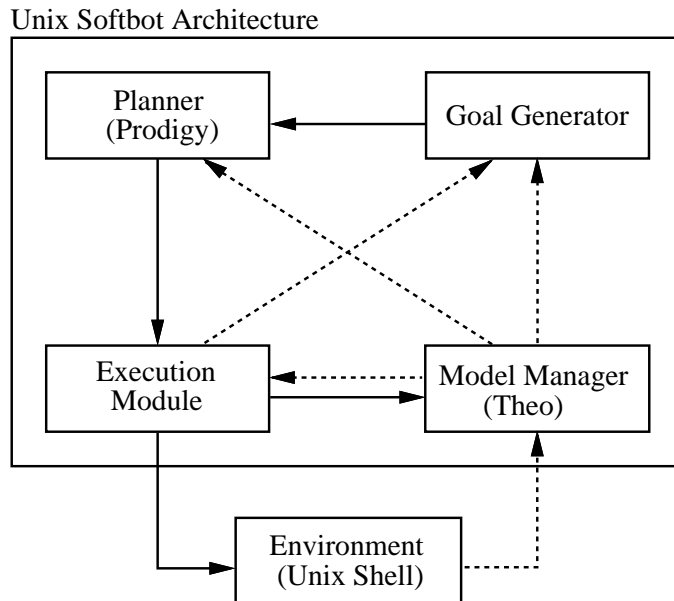


Figure 1: The architecture for Rodney

et al., 1989] as its planner and the THEO frame system [Mitchell *et al.*, 1991] to store its world model. Our architecture, depicted in figure 1, links THEO and PRODIGY to each other and to custom-built execution and sensing modules.

One motivation for utilizing PRODIGY and THEO is their built-in learning mechanisms. PRODIGY has modules for explanation-based learning [Minton, 1988], statically generating search-control knowledge [Etzioni, 1990, Etzioni, 1991b], and automatically generating abstraction hierarchies [Knoblock, 1991]. In addition, mechanisms for refining operator models from experiments, learning by analogy, and graphical knowledge acquisition are under construction [Carbonell *et al.*, 1991]. In addition, THEO supports inductive learning methods such as ID3, statistical learning methods [Etzioni, 1991a], and explanation-based learning.

Many of Rodney’s goals originate with the user. In addition, tasks such as gathering information, avoiding other users, and self protection are generated internally by the *goal generator*. The *planner* (PRODIGY) generates plans to achieve goals, using operator models of UNIX actions and state information from the *model manager* (THEO). THEO determines state information by invoking the *execution module* to execute sensory commands. The execution module also carries out PRODIGY’s plans by sending the UNIX commands, associated with each PRODIGY operator, to the UNIX shell. The execution module can query THEO to check whether the actions are having their desired effects.

Table 1 shows how THEO represents UNIX files. Each file and directory is represented as a separate frame,

indexed by a unique internal name (e.g. FILE125). The UNIX file name is a slot of the frame. This simple scheme allows us to represent multiple files with the same name (in different directories), and also to change the name of a file without modifying any of its other properties when a “mv” command is executed.

Table 2 shows a sample PRODIGY operator representing the UNIX “mv” command.⁴ The predicates are interpreted as statements about THEO’s world model. For example, the binary predicate (`component <frame> <val>`) means that the `component` slot of the `<frame>` frame has the value `<val>`. Unary predicates of the form (`class <frame>`) are interpreted by binding the variable `<frame>` to all specializations of `class`. Thus, (`file <file>`) binds `<file>` to all the frames representing files in THEO’s database. We do not currently use predicates of arity greater than two.

For execution to be possible, Rodney needs to know more about an operator than just its preconditions and effects. For example, Rodney needs to know the UNIX command an operator translates into and how to interpret the output produced by that command when executed. This information is stored in a THEO frame representing the operator.

We are currently at an early stage in the implementation. We have implemented the interface from Lisp to the UNIX shell, which enables Rodney to execute UNIX commands and to transfer UNIX output to THEO

⁴PRODIGY’s operator description language is an extended STRIPS-like language, allowing universal quantification and conditional effects.

```
(FILE125 T
 (GENERALIZATIONS (FILE))
 (FILE.TYPE DIRECTORY)
 (PARENT.DIR FILE124)
 (FILENAME "softbots")
 (PROTECTION 755)
 (OWNER "segal")
 (FILE.LIST (FILE126 FILE127))
```

```
(FILE127 T
 (GENERALIZATIONS (FILE))
 (FILE.TYPE SIMPLE)
 (PARENT.DIR FILE125)
 (FILENAME "files-kb.lisp")
 (PROTECTION 644)
 (OWNER "segal"))
```

Table 1: Two sample THEO frames: a directory (left) and a file (right).

```
(local-mv                                     ; Non-destructive move operation
 (params (<name1> <name2> <file>))
 (preconds (and (current.dir rodney <dir>          ; binds <dir> to Rodney's
 (parent.dir <file1> <dir>                        ; current directory
 (filename <file1> <name1>)
 (not (exists (<file2>) (parent.dir <file2> <dir>
 (filename <file2> <name2>))))))
 (effects ((del (filename <file1> <name1>))
 (add (filename <file1> <name2>))))))
```

Table 2: A sample PRODIGY operator representing the UNIX “mv” command.

in real time. Much of the frame structure representing UNIX files is in place, as well as much of the execution module. Table 3 shows the execution of a simple plan generated by PRODIGY off line. Table 4 shows the execution from the Rodney’s perspective. Work is proceeding on fully-interfacing PRODIGY and THEO to combine planning and execution in real time.

Related work

Since Rodney’s design is not complete, contrasting it with existing agent architectures is not productive at this point. Instead, we mention briefly two projects that have influenced our approach, and position Rodney relative to these projects.

The UNIX Consultant (UC)

UC is a natural-language interface that answers naive user queries about UNIX [Wilensky *et al.*, 1988]. The UC project focused on identifying the user’s plans, goals, and knowledge. UC utilized this information to generate informative responses to queries. For example, if the user asked “is rn used to rename files?” UC not only told her “No, rn is used to read news,” but also said that the appropriate command for renaming files is “mv.” Based on the query, UC hypothesized that the user’s goal is to rename a file and decided that the information regarding “mv” is relevant.

In contrast to Rodney, the UC does not act to achieve its own goals, but responds to user’s queries.⁵ Although

⁵The UC did note when the user’s goals were in conflict with its internal agenda and refused to answer queries such

the UC had a limited capability to learn about UNIX commands from natural-language descriptions, it did not have the capacity to explore its environment or actually execute any UNIX commands. In short, the UC was a sophisticated natural-language interface, not a mobile, goal-directed softbot. The UC project did demonstrate the fertility of UNIX as a realistic yet manageable domain for AI research, a lesson we have taken to heart.

Brooksian Creatures

In his *Computers and Thought* paper, Brooks [1991a] outlines his vision of “behavior-based” AI systems, which obviate explicit representation and reasoning capabilities. Brooks’ vision can be decomposed into three components:

- **Realism** — Brooks argues for building systems that perform tasks in realistic environments, under realistic time constraints.
- **Robotics** — Brooks argues that building robots is critical for two reasons. First, most of “evolutionary time” was spent building systems that can sense, move, and survive rather than plan or solve abstract problems (and AI researchers may wish to emulate evolution). Second, a robot’s interactions with its environment *ground* (or give meaning to) its internal symbols and processes.
- **Architecture** — Brooks argues for a particular robotic architecture (called the “subsumption architecture”) in which asynchronous networks of active as “how do I crash the system?”

```

% cd softbots
% pwd
/var/a/bowfin/u1/segal/softbots
% mv files-kb.lisp file-frames.lisp
% ls
execute.lisp          file-frames.lisp

```

Table 3: The execution of a plan that renames the file “files-kb.lisp.”

```

<cl> (execute-plan '((cd-into "softbots" file124 file125)
                    (pwd)
                    (local-mv "files-kb.lisp" "file-frames.lisp" file127)
                    (local-ls)))

0: (SHELL-COMMAND "cd softbots")
0: returned ""
0: (SHELL-COMMAND "pwd")
0: returned "/var/a/bowfin/u1/segal/softbots"
0: (PWD-SENSOR)
0: returned FILE125
0: (REALIZE-POSTCONDITION (DEL (CURRENT.DIR RODNEY FILE124)))
0: returned T
0: (REALIZE-POSTCONDITION (ADD (CURRENT.DIR RODNEY FILE125)))
0: returned FILE125
0: (SHELL-COMMAND "mv files-kb.lisp file-frames.lisp")
0: returned ""
0: (SHELL-COMMAND "ls")
0: returned "execute.lisp          file-frames.lisp"
0: (LS-SENSOR)
0: returned (FILE126 FILE127)
0: (REALIZE-POSTCONDITION (DEL (FILENAME FILE127 "files-kb.lisp")))
0: returned T
0: (REALIZE-POSTCONDITION (ADD (FILENAME FILE127 "file-frames.lisp")))
0: returned "file-frames.lisp"

```

Table 4: Rodney executing a sample plan, using sensing after each action to check that the action had the anticipated effect. Arguments such as “file125” are internal file designators generated by Rodney. Sensor commands (e.g. `pwd-sensor`) process strings returned from UNIX and create appropriately-indexed THEO frames. After sensing, the function `realize-postconditions` updates the THEO model.

computational elements operate by exchanging messages. In the absence of central control, different elements produce different behaviors (e.g. wall avoidance) in parallel. The elements are appropriately “layered” to produce coherent overall behavior.

Clearly, Rodney (which is goal-directed, has planning capabilities, and an explicit world model) is not a Brooksonian Creature.⁶ Yet our agenda was inspired by elements of Brooks’ point of view that are shared by other members of the AI community. Specifically, Rodney operates in real-time in a realistic software environ-

⁶One *can* build Brooksonian Creatures for UNIX. This is an intriguing idea, but we are not currently pursuing it.

ment. Furthermore, Rodney’s symbols are grounded by its interactions with UNIX. In his paper, Brooks focuses on *physical* environment as a basis for grounding symbols, but we believe that a software environment will do just as well.

It is interesting to note that Brooks does not require his creatures to perform useful or “realistic” tasks in their realistic environments. Indeed, most learning robots are still restricted to “toy” tasks such as fetching cups and navigating corridors. We believe this slow rate of progress is due to the difficulties inherent in robotic tasks. We are hoping that, unencumbered by hardware limitations, Rodney will converge much more rapidly to performing useful tasks.

Conclusion

Software environments (e.g. distributed databases, computer networks) are gaining prominence outside AI, demonstrating their intrinsic interest. Building agents that perform useful tasks in *software* environments is much easier than building the corresponding agents for physical environments. Furthermore, software experiments are easier to control and replicate than robotic experiments, facilitating empirical research of the kind advocated by [Langley and Drummond, 1990]. Thus, softbots are an attractive substrate for machine-learning and agent-architectures research, resolving the potential conflict between the drive for integrated agents operating in real-world task environments and the desire to maintain reasonable progress in machine learning.

Many of the fundamental issues (e.g. how to integrate planning, sensing, and execution, how to learn from experiments, how to learn from failure) are the same in both physical and software environments. However, some intriguing differences exist as well. For example, most of the information available to softbots is in textual form (e.g. UNIX man pages). Thus, a softbot's sensing ability is limited by its ability to comprehend natural-language text. A robot's sensing ability, in contrast, is limited by the fidelity of its physical sensors (e.g. television cameras) and the power of its image understanding algorithms. Studying softbots enables us to explore these differences and their implications for agent architecture.

Acknowledgments

We thank Denise Draper, Steve Hanks, Neal Lesh, Tom Mitchell, Dan Weld, and Mike Williamson for helpful discussions. This research was supported, in part, by the University of Washington Graduate School Research Fund.

References

- [Al-Badr and Hanks, 1991] Al-Badr, Badr and Hanks, Steve 1991. Critiquing the tileworld: Agent architectures, planning benchmarks, and experimental methodology. University of Washington.
- [Brooks, 1991a] Brooks, Rodney A. 1991a. Intelligence without reason. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*.
- [Brooks, 1991b] Brooks, Rodney A. 1991b. Intelligence without representation. *Artificial Intelligence* 47:139–160.
- [Carbonell and Hood, 1986] Carbonell, J.G. and Hood, G. 1986. The world modelers project: Learning in a reactive environment. In *Machine Learning: A Guide to Current Research*. Kluwer Academic Press. 29–34.
- [Carbonell *et al.*, 1991] Carbonell, Jaime; Etzioni, Oren; Gil, Yolanda; Joseph, Robert; Knoblock, Craig; Minton, Steve; and Veloso, Manuela 1991. Prodigy: An integrated architecture for planning and learning. *ACM SIGART Bulletin* 2(4). Also in the working notes of the 1991 AAAI Spring Symposium on integrated architectures.
- [Etzioni, 1990] Etzioni, Oren 1990. Why Prodigy/EBL works. In *Proceedings of the Eighth National Conference on Artificial Intelligence*.
- [Etzioni, 1991a] Etzioni, Oren 1991a. Embedding decision-analytic control in a learning architecture. *Artificial Intelligence* 49(1–3):129–160.
- [Etzioni, 1991b] Etzioni, Oren 1991b. STATIC: A problem-space compiler for Prodigy. In *the Proceedings of the Ninth National Conference on Artificial Intelligence*.
- [Kahn and Cerf, 1988] Kahn, Robert E. and Cerf, Vinton G. 1988. An open architecture for a digital library system and a plan for its development. Technical report, Corporation for National Research Initiatives.
- [Knoblock, 1991] Knoblock, Craig A. 1991. *Automatically Generating Abstractions for Problem Solving*. Ph.D. Dissertation, School of Computer Science, Carnegie Mellon University. Available as technical report CMU-CS-91-120.
- [Laird and Rosenbloom, 1990] Laird, John E. and Rosenbloom, Paul 1990. Integrating planning execution, and learning in soar for external environments. In *Proceedings of the Eighth National Conference on Artificial Intelligence*.
- [Langley and Drummond, 1990] Langley, Pat and Drummond, Mark 1990. Toward an experimental science of planning. In Sycara, Katia P., editor 1990, *Proceedings of the workshop on innovative approaches to planning, scheduling, and control*. Morgran Kaufmann.
- [Langston *et al.*, 1989] Langston, C.; Farmer, D.; and Rasmussen, S., editors 1989. *Proceedings of an interdisciplinary workshop on the synthesis and simulation of living systems*. Addison-Wesley.
- [Maes and Brooks, 1990] Maes, Pattie and Brooks, Rodney A. 1990. learning to coordinate behaviors. In *Proceedings of the Eighth National Conference on Artificial Intelligence*.
- [Messinger *et al.*, 1991] Messinger, Eli; Shoens, Kurt; Thomas, John; and Luniewski, Allen 1991. Rufus: the information sponge. Technical Report RJ 8294, IBM Almaden research center.
- [Minton *et al.*, 1989] Minton, Steven; Carbonell, Jaime G.; Knoblock, Craig A.; Kuokka, Daniel R.; Etzioni, Oren; and Gil, Yolanda 1989. Explanation-based learning: A problem-solving perspective. *Artificial Intelligence* 40:63–118. Available as technical report CMU-CS-89-103.

- [Minton, 1988] Minton, Steven 1988. *Learning Effective Search Control Knowledge: An Explanation-Based Approach*. Ph.D. Dissertation, Carnegie Mellon University. Available as technical report CMU-CS-88-133.
- [Mitchell *et al.*, 1991] Mitchell, Tom M.; Allen, John; Chalasani, Prasad; Cheng, John; Etzioni, Oren; Ringuette, Marc; and Schlimmer, Jeffrey C. 1991. Theo: A framework for self-improving systems. In VanLehn, K., editor 1991, *Architectures for Intelligence*. Erlbaum.
- [Mitchell, 1990] Mitchell, Tom 1990. Becoming increasingly reactive. In *Proceedings of the Eighth National Conference on Artificial Intelligence*.
- [Philips *et al.*, 1991] Philips, Andrew; Swanson, Keith J.; Drummond, Mark E.; and Bresina, John L. 1991. The NASA tileworld simulator. In *Proceedings of the AAAI-91 Workshop on Real-Time AI*.
- [Pollack and Ringuette, 1990] Pollack, Martha E. and Ringuette, Marc 1990. Introducing the tileworld: experimentally evaluating agent architectures. In *Proceedings of the Eighth National Conference on Artificial Intelligence*.
- [Russell, 1989] Russell, Stuart 1989. execution architectures and compilation. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*.
- [Simmons, 1991] Simmons, Reid 1991. Personal communication.
- [Simon, 1983] Simon, Herbert A. 1983. Why should machines learn. In Michalski, S. Ryszard; Carbonell, G. Jaime; and Mitchell, Tom M., editors 1983, *Machine Learning An Artificial Intelligence Approach (volume I)*. Morgan Kaufmann.
- [Sutton, 1990] Sutton, R. 1990. First results with DYNA an integrated architecture for learning, planning, and reacting. In *Proceedings of the AAAI Spring Symposium on Planning in Uncertain, Unpredictable, or Changing Environments*.
- [Sycara, 1990] Sycara, Katia P., editor 1990. *Proceedings of the workshop on innovative approaches to planning, scheduling, and control*. Morgan Kaufmann.
- [Tan, 1991] Tan, Ming 1991. *Cost-sensitive Robot Learning*. Ph.D. Dissertation, Carnegie Mellon University. Available as technical report CMU-CS-91-134.
- [Vere and Bickmore, 1990] Vere, Steven and Bickmore, Timothy 1990. A basic agent. *Computational Intelligence* 6(1):41–60.
- [Waldrop, 1990] Waldrop, M. M. 1990. Learning to drink from a fire hose. *Science* 248(4956).
- [Wilensky *et al.*, 1988] Wilensky, Robert; Chin, David; Luria, Marc; Martin, James; Mayfield, James; and Wu, Dekai 1988. The Berkeley UNIX Consultant project. *Computational Linguistics* 14(4):35–84.