Dynamic Knowledge Integration during Plan Execution

John E. Laird **Douglas J. Pearson** **Randolph M. Jones**
Robert E. Wray, III
Artificial Intelligence Laboratory
The University of Michigan
1101 Beal Ave.
Ann Arbor, MI 48109
laird@umich.edu
FAX: (313) 747-1761

The goal of our work is to develop architectures for general intelligent agents that can use large bodies of knowledge to achieve a variety of goals in realistic environments. Our efforts to date have been realized in the Soar architecture. In this paper we provide an overview of plan execution in Soar. Soar is distinguished by its use of learning to compile planning activity automatically into rules, which in turn control the selection of operators during interactions with the world. Soar's operators can be simple, primitive actions, or they can be hierarchically decomposed into complex activities. Thus, Soar provides for fast, but flexible plan execution.

Following our presentation of plan execution, we step back and explicitly consider the properties of environments and agents that were most influential in Soar's development. From these properties, we derive a set of required general agent capabilities, such as the ability to encode large bodies of knowledge, use planning, correct knowledge, etc. For each of these capabilities we analyze how the architectural features of Soar support it. This analysis can form the basis for comparing different architectures, although in this paper we restrict ourselves to an analysis of Soar (but see Wray et al. (1995) for one such comparison).

Of the capabilities related to plan execution, one stands out as being at the nexus of both the environment/agent properties and architecture design. This is the capability to integrate knowledge dynamically during performance of a task. We assume that central to any agent is the need to select and execute its next action. To be intelligent, general, and flexible, an agent must use large bodies of knowledge from diverse sources to make decisions and carry out actions. The major sources of knowledge include an agent's current sensing of the world, preprogrammed knowledge and goals, the agent's prior experience, instructions from other agents, and the results of recent planning activities (plans). However, many plan execution systems base their decisions solely on their plans, thus limiting their flexibility. One reason is that it is difficult to integrate plan control knowledge dynamically and incrementally with all of the other existing knowledge as the agent is behaving in the world. What we hope to show is that this is an important capability for responding to realistic environments, and that Soar's architectural components provide this capability in a general and flexible way.

Soar has been used for a variety of real and simulated domains, including real and simulated mobile robot control [Laird and Rosenbloom, 1990], real and simulated robotic arm control [Laird *et al.*, 1991], simulated stick-level aircraft control [Pearson *et al.*, 1993], simulated tactical aircraft combat [Tambe *et al.*, 1995], and a variety of other simulated domains [Covrigaru, 1992, Yager, 1992]. We will use a variety of examples from these domains to illustrate our points through-

out the paper.

1 Planning, Execution, and Soar

In the most basic form of planning, a system is given a description of a state in the environment, a goal or goals to achieve, and a set of actions (operators) that can be executed in the environment. If the system does not know how to achieve the goals from the initial state specification, it searches for a sequence of operators that achieve the goal. The planner returns this sequence as a plan (or case or script) that an execution system can follow. The execution systems implements the plan by carrying out each of the actions in the agent's environment. If the overall system has any sort of learning facility, it might also store the plan for future use in similar situations.

In Soar, operators, states, and goals are the basic components for solving problems and planning in Soar fits the basic model of planning. However, the details of Soar lead to interesting variants. In Soar, problem solving occurs through the continual selection and application of operators. Operators in Soar are not represented as monolithic units, such as in STRIPS where there is a precondition-list, an add-and-delete-list and a post-condition list. Instead, Soar employs a more flexible representation, using rules to implement all the basic functions associated with operators:

1. **Operator Proposal:** A rule can propose that an operator should be considered for application in the current situation. Multiple proposal rules for the same operator express disjunctive preconditions.

2. **Operator Comparison:** A rule can create a relative ordering on the preferability of choosing a single operator from a set of proposed operators. These orderings are made using the *preference* data structure.

3. **Operator Application:** A rule can implement the application (execution) of the operator by changing the system's internal state representation or initiating actions in the execution environment. If a rule is conditional on the state of the environment this leads to conditional or sequential execution.

4. **Operator Termination:** A rule can determine that the current operator is finished (so a new operator can be selected).

Each aspect of an operator can be specified (and learned) independently. It is only during run time that all aspects of an operator dynamically come together to produce behavior. The basic operation consists of first matching and firing rules that propose and compare operators relevant to the current situation. Based on the comparison, the architecture selects the current "best" operator. Once selected, relevant rules fire to apply the operator, and then to detect when the operator has completed its actions. In parallel with application and termination, new operators are proposed as the state changes and the cycle continues.

The first step in planning is determining that a plan is required. In many systems, planning and execution are separate modules, with an executive that decides whether the system should be executing or planning. In Soar, the system is continually executing, and only when there is insufficient knowledge to propose operators will planning be invoked. Specifically, during pursuit of a goal, Soar will reach an *impasse* if either no operators are proposed to apply to the current state, or if there are insufficient preferences to select between the set of proposed operators. These situations indicate that the system has either incomplete or conflicting knowledge, and that more knowledge is required to make a decision.

When a system reaches an impasse, the architecture automatically creates a subgoal representing the problem of removing the impasse. The subgoal can trigger rules that implement planning, which also consists of the selection and application of operators. Whether these operators are internal models of actions in the domain, or are more indirect planning operators (as in POP planners), is determined by what knowledge the system has — either one is a possibility. No matter which approach is taken, the end result of planning is the creation of knowledge to allow the execution system to continue, which in Soar means the creation of a preference that allows an operator to be selected. This is a result of the processing in the subgoal and in Soar, all such results lead to the automatic creation of a rule that summarizes the reasoning that was done in the subgoal. This rule is a permanent addition to the system's knowledge and will apply in future situations, thus gradually reducing the need for planning.

Many planning systems return a monolithic structure that specifies an entire solution path (or set of paths) for a particular problem. An alternative approach is for a system to create *reactive plans*, consisting of many fine-grained "plan pieces" that activate themselves dynamically according to a changing environment. Although both approaches are possible within the Soar architecture, the latter approach is supported more naturally by Soar's execution and learning methods. At each step during the planning process where there is indecision, an additional impasse will arise. When a complete path to the goal is found, the system will learn rules for each step in the plan where its current knowledge is either incomplete or in conflict.

Thus, most Soar planning systems do not incorporate plans (sequences of operators) as first-class objects. Rather, components of plans are rules that were incrementally learned during planning and are later retrieved piecemeal, triggered by the current state of the environment. The rules that propose operators depend on the knowledge dependencies of the domain, rather than simply implementing sequences of operators by rote. Thus, the first-class objects in Soar planning systems are usually operators. This means that operator sequences can arise from multiple different planning episodes, allowing the system dynamically to combine plans when it is appropriate to do so.

During both planning (within the context of a subgoal) and execution, Soar systems are continually selecting and applying operators. Once an operator is selected, rules to apply the operator should fire. However, a Soar system might select complex operators. Consider the example of flying a plane. It is unlikely that an agent would initially plan out its actions at the level of individual stick movements. Instead, planning would be based on operators for flying a flight-plan such as trying get from one waypoint to another. Flying the flight-plan would be decomposed into operators for turning, climbing and changing speed, which in turn would be decomposed into operators for moving a stick or throttle. Each of these levels is a problem space with its own operators, control knowledge, and state information (although much of the state may be shared across levels). These are not necessarily abstractions and may include aggregations of data as well.

Soar directly supports this hierarchical organization of knowledge for execution through impasses that arise while attempting to apply complex operators. If the agent starts out in the flight-plan problem space, it may select the fly-to-next-waypoint operator. However, this is such a complex task that it is infeasible to write a set of rules to cover all possible situations directly. Thus, an impasse will arise, and in the ensuing subgoal, Soar will work in the problem space of turning, climbing, and changing speed. Here it may have to plan, but once it selects an operator, it too will lead to an impasse with a subgoal. In that subgoal, operators for moving the stick (and adjusting the throttle) can be selected, and at this level rules can fire that send output commands to the motor systems. When all of the appropriate primitive operators have been used to achieve the turn or climb operator that led to the subgoal, a rule that tests for completion can fire and terminate the operator, allowing the selection of a new turn or climb operator.

In many systems, this type of decomposition would be represented in terms of explicit goals

and subgoals, or as some other type of structure (such as hierarchical tasks). Soar uniformly treats its knowledge as operators, with the following advantages:

1. The same proposal, selection, and termination structure used for primitive operators can be used for these more goal-like operators. This provides a powerful control mechanism and maintains uniformity throughout the architecture.

2. Through learning, it is possible to acquire the requisite rules for these complex operator/goals so that it is no longer necessary for subgoals to be generated with the ensuing decomposition into suboperators. This is a natural result of the learning mechanism, which compiles the problem solving in the subgoals into rules for application of the original "complex" operator [Wray *et al.*, 1996]. These rules will apply in the future, but if there are cases not covered by the compiled rules (because of variation in the task or environment), the subgoals will once again be generated and more reflective problem solving will be used.

Taking all of these issues into account, the picture of integrated planning, execution, and learning in the Soar architecture looks somewhat different from the standard picture of planning that we presented at the beginning of this section. To summarize, a fully integrated Soar system would spend as much time as possible running its execution knowledge, proposing operators in response to the current state and goal representations, and applying operators to execute commands in the external environment. When execution knowledge fails to propose a unique operator for application, an impasse triggers the automatic creation of a subgoal. The Soar system's planning knowledge recognizes these situations and initiates planning to resolve the impasse, so the execution knowledge can continue working on the domain problem. That is, rules match against execution-level impasses to propose operators for creating plans (e.g., carrying out a search through the state space). When a series of planning operators eventually determine an appropriate action (or chain of actions) to take, they return operator proposal "suggestions" to the execution level. The learning mechanism automatically compiles these planning episodes, creating rules that will immediately propose the appropriate operators in the future. In the long term, the execution system decreasingly encounters impasses, as it collects learned rules that represent fully reactive execution knowledge for the domain.

The discussion so far has covered behavior generation in generic problem domains. However, we are interested in scaling this approach to domains with the types of complexities that are usually reserved for humans to deal with. In short, we intend to push the approach to integrated execution, planning, and learning, to create models of general intelligence. In the next section, we offer a functional analysis of some of the different types of complexities such systems must address. We then present our arguments for how the design decisions incorporated into Soar encourage solutions that can handle these complexities, much in the same manner that the architecture encourages a flexible solution to execution and planning in more traditional domains.

2 Functional Analysis

In the remainder of the paper we identify the environmental constraints that must be addressed by models of general intelligent behavior, together with a set of agent capabilities that meet those constraints. A successful architecture must support each of these capabilities, so our discussion includes descriptions of Soar's mechanisms for each. Figure 1 summarizes our analysis.

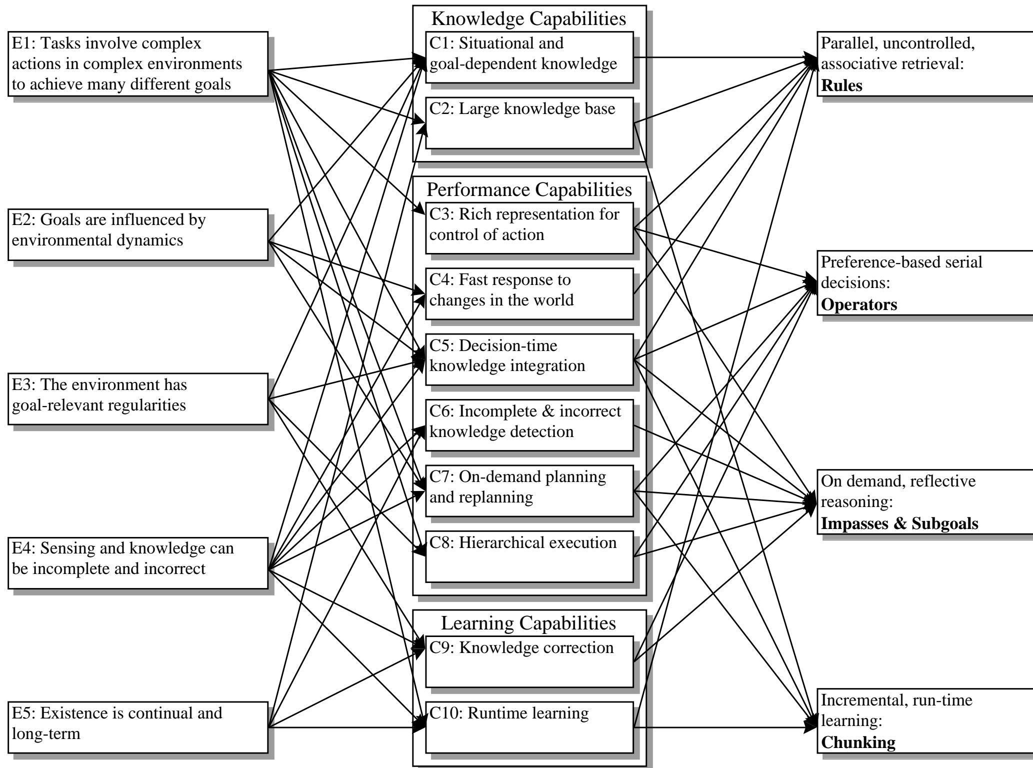

Figure 1: An analysis of Soar in terms of environmental/agent properties and derived required agent capabilities.

2.1 Environmental and Agent Properties

At the knowledge level, the defining characteristics of an agent are its available actions, its knowledge, and its goals. In this section we explore the actions, knowledge, and goals of agents and how they interact with different characteristics of the environment. We have attempted to include properties of environments that are similar to the ones we as humans interact with day to day, because our goal is to create an architecture that supports general intelligent systems. This may not be optimal for more specific applications with different constraints (such as hard real-time constraints), but these properties have been central to our research. Some of these are obvious, or so ingrained in the literature that they are rarely made explicit (such as the fact that there are regularities in the environment). However, they all form the basis for justifying specific knowledge capabilities, and they indirectly justify our architectural design. One useful outcome of the workshop would be a compilation and comparison of such lists.

E1. **Tasks involve complex actions in complex environments to achieve many different goals:**
An agent has many different ways it can interact with its environment, and the environment has many objects with many properties and relations between objects. Some of these complexities are relevant to the agent's goals. The agent's actions can take time to initiate, can have duration, and multiple actions can occur at the same time. An agent can also be tasked with many different goals, which can vary from situation to situation.

E2. **Goals are influenced by environmental dynamics:**
The environment can change independently of the agent, with such changes having an impact

on the agent's ability to achieve its goals.

E3. **The environment has goal-relevant regularities:**
The environment is not just an arbitrary set of objects and activities, but has recurring regularities. There are different regularities at a variety of times scales, making it possible to organize knowledge about action and the environment hierarchically. Without this property, learning would not be a worthwhile activity.

E4. **Sensing and knowledge can be incomplete and incorrect:**
Agents do not have complete and correct sensing of the environment. They often do not know all of the effects of their actions, nor do they correctly know all of the dynamics of other objects and entities in the environment. Incomplete and incorrect knowledge is the problem underlying many issues, including some aspects of dynamism (as long as the agent knows exactly how the world will change, it can plan for them), uncertainty (even if the environment is non-deterministic, the agent can plan if it knows all the possible states and actions), and changes in objectives (non-plan-specific goals should change only if the world changes in ways the agent cannot predict).

E5. **Existence is continual and long-term:**
The agent may always be active in an environment, without any "down time" in which it or its designers can make modifications to its knowledge base. In addition, the length of time an agent can exist makes it necessary that the agent can dynamically scale up its knowledge base as it learns.

It is important to note that the first three of these environmental features specify constraints about the agent's goals. It is not enough to identify features concerning only the environment (dynamics, non-determinism, complexity, etc.). These are only important in so far as they interact with the goals and knowledge of the agent. If there is no potential for interaction, the agent can ignore them.

2.2 Agent Capabilities

The second step of our analysis is to derive a set of capabilities that intelligent agents must possess to function successfully in environments with the properties listed above. In addition, we describe how Soar provides each capability through the interaction of its basic architectural mechanisms. The capabilities are grouped into three categories: capabilities that directly support the use of knowledge, capabilities that directly support an agent's performance in pursuit of its goals, and capabilities that directly support learning and adaptation.

C1. **Situational and goal-dependent knowledge:**
The agent must represent and use knowledge that is sensitive to both the current situation and the agent's goals. Purely reactive, or purely goal-driven agents can function well in some limited environments, but in complex [E1], dynamic [E2] environments where the agent can have many goals, an agent must respond not just to what it expects (which is incomplete [E4]), but also to what actually happens. The fact that there are regularities in the environment makes this possible [E3].

Soar supports this capability by allowing rules to test both the current situation and the current goals, as well as providing mechanisms for creating goals.

C2. **Large bodies of knowledge:**
The agent must represent and use large amounts of knowledge because of the complexity of

the environment, the agent's abilities to interact with the environment, and the variety of goals the agent must pursue [E1].

Soar uses the latest in rule-system technology [Doorenbos, 1993], which makes it possible to create very large rule bases that have low latency of response. For example, the TacAir-Soar system [Tambe *et al.*, 1995] flies simulated tactical aircraft by using more than 3,200 rules. The system is fast enough to run four separate agents simultaneously in real time on a single workstation.

C3. **Rich representation for control of action:**
The agent must have a rich representation for control, because the actions it can perform are complex [E1]. Some actions may need to occur in rapid sequence while others may need to execute in parallel. In addition, some activities may require time to occur, or have conditional effects or looping behavior.

Soar supports such a representation through a combination of rules, operators, and goals. Once an operator is selected, any number of rules can contribute to its application, providing disjunctive, conditional execution. More deliberate control is possible through the selection of operators. Finally, fully reflective reasoning is possible through the generation of subgoals, but at the cost of losing reactivity.

C4. **Fast response to changes in the world:**
The agent must be reactive to the environment, because the achievement of its goals may be related to the dynamics of the environment [E2]. Reactivity would be unnecessary if the agent's knowledge of the environment and other agents was complete and correct — all actions could be completely planned out [E4]. However, there will always be fundamental limits to reactivity, in that the agent will have a basic cycle time that it cannot overcome.

Soar provides rapid responses to environmental changes by encoding plans as rules and employing an efficient matching algorithm. One empirical study found a firing rate of 500 rules/second (2msec/rule) for a moderate size Soar system (3300 rules). By representing plans as a series of rules that guide local decisions, Soar efficiently supports interleaving multiple plans during execution. Unexpected environmental changes, (e.g., strong winds blowing a plane off course) may force the agent away from its original plan. However, if the agent has previously planned a method for making course corrections, those rules can fire, guiding the agent's execution. This behavior is functionally equivalent to switching dynamically to a new plan. Soar's learning mechanism compiles the processing from subgoals into rules that respond directly to the current situation, allowing Soar to translate reflective knowledge (both in terms of planning and hierarchical decompositions of tasks) into reactive knowledge.

However, there are some cases where very large numbers of rules can fire during the application and/or selection of an operator. This can happen when there are massive changes to its sensory data. For example, in the tactical air combat domain, our system can currently sense well over 200 independent types of data, and a large number of those can change at the same time. We are looking at a combination of changes to the architecture to reduce processing during operator application. We are also investigating attention mechanisms that greatly reduce the number of changes in the agent's working memory due to sensing. Even with these changes, it is unlikely that we will be able to guarantee a specific response time; however the system is sufficiently fast for soft-real time problems such as stick-level aircraft control. Of course, this behavior depends on sufficient control knowledge (such as through training), so that impasses requiring planning do not arise.

C5. **Decision-time knowledge integration:**
The agent needs to use all directly available knowledge about the current situation because

of the inherent incompleteness (and possible incorrectness) of it knowledge [E4] and the complexity of its interaction with a dynamic world [E1, E2]. Thus, it must continually re-evaluate which action it should take to best achieve its goal in light of the current situation. This knowledge may come from recent planning activity, but the agent will inevitably find itself in a situation that the plan does not cover. It should be able to use knowledge from whatever sources it has available during planning and execution. For example, knowledge can come from an instructor, from additional domain theories (e.g. a theory of failure states), direct planning knowledge, etc. These sources have the possibility of contributing because there are some regularities in the environment, without which, learning would be useless [E3].

All aspects of the Soar architecture contribute to its ability to integrate its knowledge for run-time decision making. Consider the selection of an operator. A Soar system's rules fire in parallel, acting as an associative retrieval mechanism, producing whatever preferences are relevant. These rules can be written by a programmer, learned from prior planning episodes, or learned from exploration or instruction. Soar's decision scheme integrates this knowledge to select the most appropriate operator given the system's knowledge. If the knowledge is insufficient or in conflict, reasoning within a subgoal must determine the appropriate action. The processing in the subgoal has access to all of the current situation as well as all of the rules (although they can be accessed only indirectly by creating situations in which they will fire). The processing in the subgoal is then compiled by the learning mechanism so that in the future, the subgoal is unnecessary.

C6. **Detect incomplete and incorrect knowledge:**
The agent must detect when its knowledge is incomplete or incorrect [E3], so that it can both correct its current behavior and correct its long-term knowledge for future situations [E5].

Soar detects when there is incomplete or conflicting knowledge through its decision procedure, which creates an impasse. This is only a weak method for detecting knowledge errors, but can be effective in many situations. This approach can be supplemented by task knowledge (additional rules) that detects impossible or undesirable situations. After completing a plan, subsequent impasses during execution indicate that either an unexpected change occurred in the environment (e.g., if there's a sudden down-draft while flying a plane) or that the agent's knowledge is wrong (e.g., pulling back too far on the stick during a landing takes the plane out of the glide-path, forcing an impasse and the need to replan the landing). In these situations, an impasse indicates the agent is unsure how to proceed towards its current goal and therefore implicitly indicates that it has left its planned path.

C7. **On-demand planning and replanning:**
The agent must be able to (re)plan whenever its earlier plans did not adequately predict the results of its actions or the dynamics of the environment [E4, E2]. The complexity of the world precludes completely planning for all possible world states [E1]. Thus, although runtime integration of knowledge is necessary [C5], there will be times when it is inadequate, and the agent must fall back on planning.

Soar's impasse and subgoaling mechanisms directly support on-demand planning and replanning. In essence, the architecture recognizes no difference between planning and replanning. Each time an impasse occurs, a system can plan from the current situation. Any partial plans created earlier will automatically help constrain the search performed in generating a new plan. Planning can be carried out completely at an abstract level, waiting until execution of an abstract operator to fill in the details, or planning can initially include details. The choice is determined by the task knowledge used to encode the planning process.

C8. **Hierarchical execution:**

The agent must organize its knowledge in line with the regularities of the environment [E3], maximizing the generality of the knowledge it encodes because of the complexity and variability of the environment and the agent's goals [E1]. In many cases, this means organizing knowledge about action hierarchically in terms of goals and subgoals. With such an organization, the agent can decompose some of its actions into sequences of more primitive actions, using the context of the higher level action to constrain the choices and reducing the knowledge required to perform a task.

Hierarchical execution occurs naturally in Soar when complex operators are selected. When a hierarchical operator is selected, an impasse arises during operator application. In the ensuing subgoal, operators are proposed, selected, and applied at a more detailed level (this requires task knowledge of course). Through chunking, this hierarchical decomposition is compiled so as to avoid decomposing the same problem anew every time it is encountered.

C9. **Knowledge Correction:**

The agent must be able to correct its knowledge [E4] so that in the future [E5], when it encounters similar situations [E3], previous errors can be overcome by correct responses.

If Soar only had rules but no operators, it would be very difficult to correct an agent's knowledge. It would either have to mask the incorrect rule (very difficult to do without some syntactic conflict resolution scheme, which in turn limits the system's ability to integrate its knowledge), or modify the incorrect rule (which requires identifying which rule is incorrect and modifying its conditions and actions; a very difficult process when large numbers of rules contribute to an incorrect action). However, with the addition of operators, Soar agents have the ability to correct the *decisions* for selecting operators, as opposed to the rules themselves. This is possible because the preference-based decision procedure has a monotonic property, where it is possible to correct any decision through the addition of new preferences [Laird, 1988]. This property has been the basis for a Soar system called IMPROV, which is able to detect and correct both operator proposal and operator application knowledge automatically, based on feedback and experimentation in an external environment [Pearson and Laird, 1996].

C10. **Runtime learning:**

The agent must learn and modify its knowledge base while it is running, because it has a continual existence [E5], as well as goals that require continual maintenance [E1]. Additionally, the mechanics of the environment may change [E2], leading to incomplete or incorrect planning knowledge [E4]. Thus, the agent must support the ability to learn while it is behaving.

In Soar, rules are learned dynamically as impasses are resolved during the life of the agent. The new rules are integrated with the existing rule base, possibly leading to new behavior at each decision point. Soar's learning mechanism is efficient enough so that it does not significantly disturb reasoning. It is difficult to measure, but Soar's mechanisms associated with learning appear to consume 15-20% of reasoning time. However, there could be cases where extended, complex problem solving in a subgoal, could cause the creation of a new rule to take a significant amount of time. Such episodes might severely disrupt an agent's ability to react. Theoretically, this processing could be evenly distributed over the problem solving in the subgoal; however this is not the case in the current implementation, and to date it has not caused problems.

3 Conclusion

The purpose of this paper was to provide an overview of plan execution in Soar. Moreover, we wished to ground the design of Soar in specific properties of agents and their environments. Our approach was to use the properties related to general intelligent behavior as a basis for necessary agent capabilities. These capabilities are each related to specific architectural features. We hope that the workshop will provide a forum for us to compare our architectural choices and desired agent capabilities with other systems. In no way do we expect that our list of properties and capabilities are universally shared, as different problems, domains, and agents will have different properties and different required capabilities. However, we believe this type of analysis is a stepping stone to understanding the strengths and weaknesses of various approaches to planning and execution.

References

[Covrigaru, 1992] A. Covrigaru. *Emergence of meta-level control in multi-tasking autonomous agents*. PhD thesis, University of Michigan, 1992.

[Doorenbos, 1993] R. Doorenbos. Matching 100,000 learned rules. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*. AAAI, AAAI Press, 1993.

[Laird and Rosenbloom, 1990] J. E. Laird and P. S. Rosenbloom. Integrating execution, planning, and learning in Soar for external environments. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 1022–1029, July 1990.

[Laird *et al.*, 1991] J. E. Laird, E. S. Yager, M. Hucka, and C. M. Tuck. Robo-Soar: An integration of external interaction, planning and learning using Soar. *Robotics and Autonomous Systems*, 8:113–129, 1991.

[Laird, 1988] J. E. Laird. Recovery from incorrect knowledge in Soar. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 618–623, August 1988.

[Pearson and Laird, 1996] Douglas J. Pearson and John E. Laird. Toward incremental knowledge correction for agents in complex environments. In Stephen Muggleton, Donald Michie, and Koichi Furukawa, editors, *Machine Intelligence*, volume 15. Oxford University Press, 1996.

[Pearson *et al.*, 1993] D. J. Pearson, S. B. Huffman, M. B. Willis, J. E. Laird, and R. M. Jones. A symbolic solution to intelligent real-time control. *Robotics and Autonomous Systems*, 11:279–291, 1993.

[Tambe *et al.*, 1995] M. Tambe, W. L. Johnson, R. M. Jones, F. Koss, J. E. Laird, P. S. Rosenbloom, and K. Schwamb. Intelligent agents for interactive simulation environments. *AI Magazine*, 16(1), 1995.

[Wray *et al.*, 1995] Robert Wray, Ronald Chong, Joseph Phillips, Seth Rogers, William Walsh, and John Laird. Organizing information in mosaic: A classroom experiment. *Computer Networks and ISDN Systems*, 28:167–178, 1995.

[Wray *et al.*, 1996] Robert Wray, John Laird, and Randolph Jones. Compilation of non-contemporaneous constraints. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, Portland, OR, August 1996.

[Yager, 1992] E. S. Yager. *Resource-dependent behavior through adaptation.* PhD thesis, University of Michigan, 1992.