An introduction to Soar as an agent architecture

Article · January 2005

CITATIONS
11

READS
1,279

2 authors, including:

Randolph M. Jones
Soar Technology, Inc.
97 PUBLICATIONS **1,890** CITATIONS

SEE PROFILE

An Introduction to Soar as an Agent Architecture

Robert E Wray
Soar Technology
3600 Green Road Suite 600
Ann Arbor MI 48105
wrayre@acm.org

Randolph M Jones
Colby College & Soar Technology
5857 Mayflower Hill
Waterville, ME 04901-8858
rjones@soartech.com

1 Introduction

The Soar architecture was created to explore the requirements for general intelligence and to demonstrate general intelligent behavior (Laird, Newell, & Rosenbloom, 1987; Laird & Rosenbloom, 1995; Newell, 1990). As a platform for developing intelligent systems, Soar has been used across a wide spectrum of domains and applications, including expert systems (Rosenbloom, Laird, McDermott, Newell, & Orciuch, 1985; Washington & Rosenbloom, 1993), intelligent control (Laird, Yager, Hucka, & Tuck, 1991; Pearson, Huffman, Willis, Laird, & Jones, 1993), natural language (Lehman, Dyke, & Rubinoff, 1995; Lehman, Lewis, & Newell, 1998), and executable models of human behavior for simulation systems (Jones et al., 1999; Wray, Laird, Nuxoll, Stokes, & Kerfoot, 2004). Soar is also used to explore the integration of learning and performance, including concept learning in conjunction with performance (Chong & Wray, to appear; Miller & Laird, 1996), learning by instruction (Huffman & Laird, 1995), learning to correct errors in performance knowledge (Pearson & Laird, 1998), and episodic learning (Altmann & John, 1999; Nuxoll & Laird, 2004).

This chapter will introduce Soar as a platform for the development of intelligent systems. Soar can be viewed as a theory of general intelligence, as a theory of human cognition, as an agent architecture, and as a programming language. This chapter reviews the theory underlying Soar but introduces Soar primarily as an agent architecture. The architecture

point-of-view is useful because Soar integrates into one comprehensive system a number of different algorithms common in artificial intelligence. This view of Soar also facilitates comparison to other agent approaches, such as beliefs-desires-intentions (BDI) (Bratman, 1987; Wooldridge, 2000), and to rule-based systems, two approaches with which Soar shares many features. An Additional Resources section provides pointers to papers, tutorials, and other resources interested readers may access to learn the details of Soar at the programming level.

The Soar architecture grew out of the study of human problem solving. Soar is often used as a tool for the creation of fine-grained cognitive models that detail and predict aspects of human behavior in the performance of a task. (Newell, 1990) has taken this effort as far as proposing Soar as a candidate unified theory of cognition (UTC), a theory of human cognition that spans and unifies the many observed regularities in human behavior. Evaluating Soar as a UTC remains an active area of work. An example is Chong's development of a hybrid architecture that incorporates Elements of EPIC, ACT-R and Soar (EASE) (Chong, 2003; Chong & Wray, to appear). However, Soar is increasingly used as a tool useful for building intelligent agents, especially agents that individually encode significant knowledge and capability. Obviously, these agents could behave in ways comparable to humans in particular application domains, but the focus is not limited to human behavior representations. This chapter therefore describes the general commitments of the Soar architecture as a platform for intelligent systems (human and/or otherwise) and the application of these principles in the development of intelligent systems.

2 Soar as a general theory of intelligence

As an intelligent agent architecture, the theoretical principles motivating Soar's design are important for two reasons. First, the theory provides insight in understanding Soar as an implementation platform, especially in terms of agent design decisions. The processes and representations of the Soar architecture are derived directly from the theory. Second, just like any software architecture, Soar biases agent implementations towards particular kinds of solutions. Allen Newell referred to this as "listening to the architecture" (Newell, 1990). Understanding the theory makes it easier to understand these biases in approach and implementation.

2.1 The knowledge level, symbol level, and architecture

An agent can be described at three distinct levels: the knowledge level, the symbol level, and the architecture level (Newell, 1990). The *knowledge level* refers to an external, descriptive view of an agent (Newell, 1982). For example, if a robot-controlled car was observed to yield to pedestrians, stop at red lights, and drive more slowly than the speed limit when driving in inclement weather, an observer could ascribe knowledge to it, such as the general rules of the road, the ability to observe and interpret situations and make judgments about pedestrians and road conditions, and the ability to adapt its behavior to the observed situation. The knowledge level assumes the *principle of rationality*, which says that if an agent has some knowledge that is relevant to the situation, it will bring it to bear. For example, if the robot continued to drive as fast as the speed limit allowed on a winding mountain road in icy conditions, one might conclude that it lacked knowledge to modify its behavior with respect to road conditions. It might be that the agent has this knowledge but also has a goal that justifies the additional speed (such as the transport of a critically injured person). However, as long as an observer is reasonably certain that the

situation being viewed is the same as the agent is viewing, the principle of rationality provides the ability to ascribe knowledge to the agent.

The knowledge level is a level for analysis; one observes the actions of an agent and makes assumptions about the knowledge it has (and does not) based on the observations. However, that knowledge must be encoded in some form. Soar assumes knowledge is encoded in a symbol system, which provides the means for universal computation (Newell, 1980a, 1990). The *symbol level* is the level in which the "knowledge" of a Soar agent (or any other agent using a symbolic representation) is represented. We say "knowledge" because, while it is common to think of an agent as having knowledge, in reality every system (human or otherwise) has only a representation of knowledge. The knowledge representations of the symbol level must be accessed, remembered, constructed, acted on, etc. before an observer can ascribe knowledge to the agent. The fixed mechanisms and representations that are used to realize the symbol system comprise the agent *architecture*. The architecture defines the processes and base-level representations with which agent programs can be constructed (Russell & Norvig, 1995).

An architecture enables the distinct separation of content (the agent program) from its processing substrate. Thus, the primary difference in Soar applications, from simple expert systems, to natural language interpretation, to real-time models of human behavior, consists of differences in the encoding of knowledge for these applications. The architecture is (largely) the same across applications. Because Soar (as a symbol system) provides universal computation, it should be sufficient for any application requiring intelligent behavior (assuming intelligence can be captured in computational

terms). However, performance efficiency and the ease with which particular algorithms are encoded and retrieved also impact the sufficiency of the architecture for producing intelligent behavior in a particular application. When researchers discover that Soar is unable to produce some desired behavior or that representation of some behavior is too costly (in terms of performance or solution encoding), a search is begun to extend or change the architecture to address the requirements of the missing capability. Laird and Rosenbloom (1995) discuss why and how the Soar architecture has evolved since its initial implementation in the early 1980's.

More practically, from an implementation point-of-view, an architecture provides the foundation for significant reuse from one application to another. The architecture itself is reused, and, at least theoretically, any relevant encodings of knowledge in former applications can also be reused. In practice, the knowledge representations of Soar applications have been only rarely reused in subsequent applications. Exceptional cases include Soar-Teamwork (STEAM), an implementation of the theory of teamwork as joint intentions (Tambe, 1997) and a Soar-based model of natural language, NL-Soar (Lehman et al., 1998). NL-Soar has been used as a component in a tactical aircraft pilot demonstration (Lehman et al., 1995), a system that learns from instruction (Huffman & Laird, 1995), and a cognitive model of a space mission test director's countdown-to-launch activities (Nelson, Lehman, & John, 1994). STEAM was explicitly designed as a reusable component and has been used in many different agent applications. However, in general, the Soar research community has concentrated more on realizing individual capabilities and creating complex agent systems than on understanding how to manage the knowledge that these systems require. At present, much research in the Soar

community is directed towards addressing this oversight by developing more efficient methods of capturing, encoding, and reusing agent knowledge. Examples of this work include learning from observation to automate the synthesis of agent knowledge representations (van Lent & Laird, 2001), developing tools for knowledge capture (Pearson & Laird, 2003), designing and implementing language extensions to support more scalable knowledge representations (Crossman, Wray, Jones, & Lebiere, 2004; Wray, Lisse, & Beard, to appear), developing algorithms for knowledge verification and agent validation (Wallace & Laird, 2003), and formalizing design patterns for frequently used capabilities (Taylor & Wray, 2004).

Finally, while some symbol systems may attempt to approximate the knowledge level, they will necessarily always fall somewhat short of the perfect rationality of the knowledge level. One can think of the way in which a system falls short of the knowledge level as its particular "psychology;" it may not act in time to *appear* to have the knowledge, it may use some fixed process for conflict resolution that leads to a failure to consider some relevant knowledge, etc. One of the fundamental tensions in the development of Soar has been whether its "psychology" should be minimized as much as possible, in order to better approximate the knowledge level, or if its limitations (because every symbol level system will have some limitations) should attempt to reflect human limitations. Superficially, a single architecture probably cannot satisfy both constraints. However, one counter-argument is that evolution has provided a good approximation of the knowledge level in human symbol processing, and taking advantage of that evolutionary design process, by attempting to replicate it, will result in better symbol systems. For example, a memory decay mechanism for Soar was resisted for a long time

because it appeared to be an artifact of the human symbol system and provided no functional advantage. However, recent research has suggested that the functional role of decay is to reduce interference (Altmann & Gray, 2002) and a recent Soar-based architecture, EASE, incorporates a decay mechanism (Chong, 2003).

2.2 Problem Space Computational Model

The Problem Space Computational Model (PSCM) (Newell, Yost, Laird, Rosenbloom, & Altmann, 1991) defines the entities and operations with which Soar performs computations. Soar assumes any problem can be formulated as a problem space (Newell, 1980b). A problem space is defined as a set of (possible) states and a set of operators, which individually transform a particular state within the problem space to another state in the set. There is usually an initial state (which may describe some set of states in the problem space) and a desired state, or goal. Operators are iteratively selected and applied in an attempt to reach the goal state. The series of steps from the initial state to a desired state forms the solution or behavior path.

<< Insert
Figure 1 here >>

Figure 1 illustrates a problem space for the well-known blocks world domain. The states consist of the arrangement of blocks on the table and on each other. The agent perceives the current configuration of blocks and monitors a specified goal configuration. We assume this problem space includes only two operators, **stack** and **put-on-table**. The diagram highlights a solution path from the initial state to the goal state. One important

contribution of the PSCM, which is not often found in other formulations of problem spaces, is a distinction between selection of an operator and its application. Under the PSCM, knowing that some operation can be applied in some situation is distinct from knowing how to execute that operation, or if it is likely to result in progress within the problem space. The knowledge representations of Soar reflect this distinction by requiring independent representations of these separate classes of knowledge.

An individual problem space defines one view of a particular problem and any single problem space may be insufficient for completely solving a problem. For example, in
Figure 1, the problem space provides enough information to specify the stacking and unstacking of blocks, but it does not provide any guidance on how to choose between different operators that might be simultaneously applicable. Similarly, the example ignores how a robot would actually move in space to accomplish the problem space operations. Unless **stack** is a primitive operation of the robot, once the robot has chosen to stack two blocks, it next has to decide *how* to perform this task.

When the knowledge represented within the problem space is not sufficient to solve the problem at hand, an *impasse* is said to have occurred. An impasse represents a lack of immediately applicable knowledge. An obvious response to an impasse is to establish a goal to resolve the impasse. The PSCM specifies that this goal (deciding between potential candidates, implementing an operator, etc.) should be pursued in another problem space. This second problem space is subordinate to the first and implements some aspect of the original problem space. The initial conditions and the specific goal are derived from the original problem space. Figure 2 illustrates possible implementation problem spaces for a version of the blocks world where a robot must execute a series of

primitive movement operations in order to stack a block on another.

<< Insert Figure 2 about here >>

Every problem space other than the initial (base) problem space is invoked to help a parent problem space.[1] The PSCM defines an ontology of impasses, detailing all the situations that can stop progress in a problem space; common impasse types are shown in Table 1. Each impasse prescribes a specific class of subordinate problem space and the states and operators in this problem space can be structured to solve the particular impasse. One significant advantage of the PSCM is that implementations of different domains all share this same ontology of impasse types. The PSCM informs agent development by guiding task formulation and termination in child problem spaces and identifying and discriminating potential problems in operator selection and application. This type of informed problem representation is one of the major advantages of developing agents within general architecture like Soar.

Soar implements the PSCM. All versions of Soar have adhered to it (Laird & Rosenbloom, 1995). However, the PSCM alone is an incomplete specification of Soar. For example, early versions of Soar did not include a reason maintenance component, but in more recent versions this functionality has become a distinctive characteristic of the Soar implementation (Laird & Rosenbloom, 1995; Wray & Laird, 2003). Similarly, different versions of Soar have implemented a variety of methods for agents to remember historical state information (Laird & Rosenbloom, 1995).

[1] In Soar, the stack of problem spaces is assumed to grow in a downward direction and the initial problem space is referred to as the "top level space" as well as the base level space.

The PSCM is the over-arching constraint in Soar architecture research. While the high-level nature of the PSCM enables the exploration of many alternative implementations, any Soar-based implementation that violated the PSCM could not be said to be a version of Soar. More strongly, in terms of Soar as a Lakatosian research program under Sophisticated Methodological Falsification (Lakatos, 1970), the PSCM forms the "hard core" of Soar as a general theory of intelligence. Whether the PSCM is, itself, strong enough and constraining enough to constitute a falsifiable hypothesis, and thus a basis for the core assumptions of Soar as a general theory on intelligence, is debatable. It is also unclear if non-PSCM aspects of Soar (e.g., parsimony, below) also should be considered among the core assumptions.

2.3 Parsimony

The continuing thread of Soar research has been to find a sufficient but *minimal* set of mechanisms that can be used to realize the full range of intelligent behavior. Soar commits to individual, uniform representations of long-term and dynamic knowledge representations, a single symbol level learning mechanism, and the uniform process of behavior execution and problem solving defined by the PSCM. Introducing multiple representations or mechanisms for the same function would violate this principle of parsimony, and, until recently, has not been a seriously challenged assumption.

This commitment to parsimony provides a stable system that is relatively easy to learn at the level of the software architecture. However, as a consequence, Soar defines a sort of low-level machine for implementing algorithms and representations not directly supported by the architecture. Soar programs directly refer to architectural elements

rather than higher-level constructs, resulting in a situation akin to an "assembly language" for intelligent systems. For example, Soar does not directly support the representation of plans. One can represent plans in Soar, but to do so one must build them from the lower level representations of the architecture. Similarly, most work in Soar assumes a single, symbol-level learning mechanism, chunking (Laird, Rosenbloom, & Newell, 1986). Additional types of learning must be realized by mapping the learning requirements to chunking and structuring and formulating agent knowledge within problem spaces to implement the learning algorithm. This mapping can be onerous in comparison to implementing a learning algorithm in a less constrained environment. More recently, students in John Laird's research group have been exploring additional symbol level learning mechanisms in Soar such as episodic learning (Nuxoll & Laird, 2004). What impact, if any, these changes will have on the PSCM, on Soar as a general theory of intelligence, and Soar as a minimal set of mechanisms for intelligence is unresolved.

3 The Soar architecture

Recall from the above that an architecture comprises the fixed mechanisms and knowledge representations of the symbol system. Because these elements are fixed, they transfer from one domain to another. The number of implemented representations and mechanisms is as small as possible, as dictated by Soar's assumption of parsimony. We enumerate each of these below, describing Soar's architecture-supported representations, the basic sense-decide-act cycle of processing, and individual processes that act within the basic control loop.

3.1 Architectural representations

Soar supports three basic representations, productions, asserted memory objects, and

preferences, which are represented in production memory, blackboard memory, and preference memory respectively. Soar operators are composed from these others, and their representation spans the three memories.

3.1.1 Productions and production memory

Soar is a production system (Newell, 1973). Each production (or rule) is specified by a series of conditions and a set of actions. Conditions are matched against the contents of a blackboard memory, and, when all conditions are satisfied, the rule actions are executed, usually specifying changes to objects on the blackboard. Figure 3 shows a Soar production for the blocks world robot illustrated in Figure 2. The production matches against Soar's input representation (the "input-link") to determine if a block meets a desired state in the problem space, represented by the **current-goal** object. The action of the production is to create an object that indicates the block is in the desired position. This new object may trigger other productions; for example, a production might match against **in-desired-position** and then explicitly mark the **current-goal** object as having been met when the condition is satisfied.

<< Insert Figure 3 about here >>

Productions are often described as "if-then" rules, comparable to the "case" statements of mainstream programming languages such as C++. However, Soar expresses the conditions and actions of productions in a form of predicate logic, rather than the propositional logic used in procedural programming languages. Thus, a production like the one in Figure 3 simultaneously considers all blocks and all **current-goals** represented on the state. The match process can generate multiple *instantiations* of the production,

with variable bindings specific to each match. Thus, in the example, if two blocks satisfied the current goal description, two instances of the production would match. In Soar, both instances would fire in parallel, resulting in two **in-desired-position** objects, one for each block.

3.1.2 Assertions and blackboard memory

Soar asserts and maintains active memory objects in a blackboard memory, called the *working memory*. As the objects expressed on the blackboard change, they trigger new productions, resulting in further changes to the blackboard. Unlike many blackboard systems, Soar's working memory is highly structured. Working memory is a directed graph, with each object described by a triple [identifier, attribute, value]. Complex objects can be composed by making the value of an object another identifier, as shown in Figure 4.

<< Insert Figure 4 about here >>

The blackboard is also segmented into state partitions. Soar assigns each problem space created in response to an impasse a distinct state object. Each state partition encapsulates the search for additional knowledge to resolve that state's impasse. Every state includes a pointer to its superstate, so that the reasoning within the state can access relevant context from the problem space in which the impasse occurred. Because the top state is the root of the working-memory graph, and each state is the root of some sub-graph, every object in memory can be traced to a state (and possibly a chain of states). The "lowest" such state in a chain is the object's "local state." Soar's top state also includes an input/output partition, which is divided into *input-link* and *output-link* objects.

Individual input and output objects are represented in the same representation language as other objects. However, input objects are placed on the input-link by an "input function" that transforms environmental percepts into the [identifier, attribute, value] representation required by Soar. Similarly, an output function interprets objects on the output-link as commands and attempts to execute them.

3.1.3 Preferences and preference memory

The *preference* data structure expresses preferences between candidate operators competing for selection. Table 2 lists some of the preferences available in Soar for selecting and comparing operators. Unary preferences such as "acceptable" and "best" express preferences about a single candidate; binary preferences compare one operator to another.

When it is time for Soar to select an operator, a *preference semantics* procedure interprets all the preferences to determine if a unique option can be identified. If no unique choice can be made, Soar generates an impasse; the impasse type is indicated by the problem in the preferences. For example, if all candidates are acceptable, but no other preferences are asserted, an operator-tie impasse will be generated, allowing Soar to initiate a search for knowledge that indicates which of the candidates should be chosen. The specific interpretation of preferences is dictated by the preference semantics procedure, which is detailed in the *Soar Users' Manual* (Laird & Congdon, 2004).

Soar stores preferences in a preference memory, which is *impenetrable* to productions. That is, productions cannot test whether one operator is better than another, or if an indifferent preference has been asserted for a particular operator. The exception is the

preference that represents whether an operator should be considered at all. This "acceptable" preference is represented in Soar's blackboard memory and thus can be tested by productions. Testing the acceptable preference allows productions to assert additional preferences about the acceptable candidate(s).

Preferences are used in Soar programs to distinguish between situations in which some operation *could* apply, and when it *should* apply (it has the highest preference). For example, in Figure 1, when the top block is placed on the table, the **stack** operator could certainly be used to put the block back on the stack of remaining blocks. A Soar production would propose the **stack** operator, as shown in the figure, making it an acceptable action to take at this point in the problem solving. However, additional preference productions could be used to prohibit or reject this candidate, because it undoes a previous step, or because in the current situation, the block already meets a partial condition of the goal. The advantage of the preference mechanism is that all the options and constraints on them do not need to be worked out at design time, but the agent can make a choice based on its current situation, resulting in *least-commitment* execution. Further, because "proposal" and "evaluation" productions are distinct representations, an agent can learn to change its preferences is useful ways, without having to modify the representation of operator pre- or post-conditions.

3.1.4 Operators

Soar's operators are equivalent, conceptually, to operators in state-based planning systems, such as those based on STRIPS (Fikes & Nilsson, 1971, 1993). These operators represent small procedures, specifying preconditions (what must be true for the operator to be activated) and actions (what the operator does). Unlike STRIPS, in which an operator is a knowledge representation unit, in Soar the representation of an operator is

distributed across productions, preferences, and memory objects within the architecture. The preconditions of an operator are expressed in one or more proposal productions. The action of a proposal production is to assert into preference memory the acceptable preference for the operator. As above, acceptable preferences are also represented on the blackboard, which allows evaluation productions to compare and evaluate acceptable candidates. When an operator is selected (during the execution of the decision procedure, described below), Soar creates an operator object in the blackboard, as shown in Figure 4. Soar allows each state exactly one selected operator at any time. Therefore, attempting to create zero or multiple operator objects will result in an impasse for that state's problem space. Once the selected operator is represented in the blackboard, it can trigger productions that produce the post-conditions of the operator, resulting in operator *application*. In the blocks world example, this could mean internally changing the position of the blocks (in a planning task) or sending output commands to the robot for execution in the environment.

3.2 The Soar decision cycle

At a high level, most agent systems can be described by a sense-decide-act (SDA) cycle (Russell & Norvig, 1995; Wooldridge, 2000), as represented in the left of Figure 5. Soar's general processing loop, its *decision cycle*, maps directly to the SDA loop, as shown in the middle diagram. Individual components of the Soar decision cycle are termed phases. During the INPUT PHASE, Soar invokes the input function (as described above), communicating any changes indicated by the environment to the agent through the input-link portion of the blackboard. In the OUTPUT PHASE, the agent invokes the output function, which examines the output-link and executes any new commands indicated there. Proprioceptive feedback about the execution of commands is provided

during the INPUT PHASE.

<< Insert Figure 5 about here >>

The reasoning within Soar's decision cycle is focused on the selection (and application) of operators. Each Soar decision consists of three phases within the "decide" portion of the SDA loop. During the ELABORATION PHASE, the agent iteratively fires any productions other than operator applications that match against the current state, including new input. This process includes "elaborating" the current state with any derived features (such as **in-desired-position** in Figure 3), proposing new operators, and asserting any preferences that evaluate or compare proposed operators. This phase uses Soar's truth maintenance system to compute all available logical entailments (i.e., those provided by specific productions) of the assertions in the blackboard.

When no further elaboration productions are ready to fire, the decision cycle is said to have reached *quiescence*. At this point, the elaboration process is guaranteed to have computed the complete entailment of the current state; any immediately knowledge applicable to the proposal and comparison of operators will have been asserted. At quiescence, Soar enters the DECISION PHASE and invokes the preference semantics procedure to sort and interpret preferences for operator selection. If a single operator choice is indicated, Soar adds the operator object to memory and enters the APPLICATION PHASE. In this phase, any operator application productions fire, resulting in further changes to the state, including the creation of output commands. Application productions are similar to elaboration productions with two exceptions: their

conditions must include a test for the existence of a selected operator, and any changes they make to the blackboard are *persistent.* Persistent objects do not get retracted automatically by Soar's truth maintenance system but must be deliberately removed by another operator. If there is not a unique choice for an operator, Soar creates a new state object in response to the impasse, so that the agent can undertake a deliberate search for knowledge that will resolve the impasse and thus enable further progress in the original state.

Soar's decision cycle conceptually is divided into these five distinct phase, as shown in the rightmost diagram of Figure 5. In this abstract form, Soar's decision cycle is roughly equivalent to the reasoning loop of BDI agents (Wooldridge, 2000). The BDI control loop consists of polling the world for new input (corresponding to Soar's INPUT PHASE), updating the world model (ELABORATION PHASE), generating desires (roughly comparable to the proposal of operators in ELABORATION PHASE), selecting an intention from the desires (DECISION PHASE), and then choosing and acting on a plan (APPLICATION PHASE).

Soar does not directly support a plan representation. Operators are used to execute individual actions (corresponding to plan steps in BDI) as well as to represent the plan itself (i.e., via problem space-based hierarchical decomposition, as in Figure 2). Another important difference between Soar and BDI at the level of the control loop is that the preference semantics for making decisions is fixed in Soar, whereas in BDI, decision and reconsideration can be customized for decisions about specific types of objects (e.g., intentions vs. plans). Thus, when using Soar, one must map alternate kinds of decision

strategies (e.g., decision theory) to Soar preferences. This is another example where Soar programming can seem like assembly language. However, because the basic decision process is uniform and fixed, it is reasonably straightforward both to implement and explore a range of decision strategies and also to cache specific decisions using Soar's built-in learning mechanism. This uniformity provides structure for additional reuse across Soar models.

3.3 Architectural processes

Within the decision cycle, Soar implements and integrates a number of influential ideas and algorithms from artificial intelligence. In particular, Soar is a production system that performs operator-based reasoning within problem spaces. The mix of productions and operators is not unique; most rule-based systems can also be seen as operator-like systems. The main difference in Soar is that individual rules do not map to individual operators; rather, as outlined above, a Soar operator is implemented by a collection of rules that individually perform one of the PSCM functions of proposal, comparison, or application. The following introduces algorithms within the decision cycle, focusing on highlighting the differences between Soar and traditional rule-based systems (RBS) such as OPS5 (Forgy, 1981), CLIPS (e.g., Giarratano, 1998) and JESS (e.g., Friedman-Hill, 2003), and production system cognitive architectures, such as ACT-R (Anderson & Lebiere, 1998) and EPIC (Kieras & Meyer, 1997). Examples will be presented from both the blocks world and TacAir-Soar, a tactical aircraft pilot model (Jones et al., 1999) that better demonstrates the role of Soar in dynamic domains.

3.3.1 Pattern-directed control

Soar brings to bear any knowledge relevant to the current problem in via associative pattern matching in a parallel match-fire production system. Thus, Soar models determine

their flow of control in Soar by the associations made to memory, rather than a sequential, deterministic control structure. Because the reasoning of the agent is always sensitive to the context, Soar readily supports both reactive and goal-driven styles of execution, and is able to switch between them during execution. Soar uses an extension of the Rete algorithm (Forgy, 1982) to ensure efficient pattern matching across the entire knowledge base. A research demonstration showed that Soar can handle as many as one million rules without a significant slow down in reasoning (Doorenbos, 1994) and TacAir-Soar, which has over 8,000 productions, runs in real-time.

Typical rule-based systems also are pattern directed. However, most use conflict resolution to choose between matching rules rather than firing all of them. Conflict resolution in typical RBS usually depends on syntactic features of rules; for example, preferring the rule instantiated with the most recent memory elements or the largest number of them. Soar uses no conflict resolution at the level of individual rules. Instead, conflict resolution occurs when choosing between operator candidates, allowing the decision to be mediated by available knowledge (in the form of preferences) rather than relying on syntactic features of the situation.

3.3.2 Reason maintenance

Soar uses computationally inexpensive reason maintenance algorithms (Doyle, 1979; Forbus & deKleer, 1993) to update its beliefs about the world. For example, if a pilot agent computes a collision course with a target it is intercepting, and the target's course changes, a Soar agent can use its justification-based truth maintenance (JTMS) to update the collision course automatically without additional deliberation. In Figure 3, if the input changed so the production no longer matched (e.g., a block was moved), Soar

would respond by automatically retracting the **in-desired-position** object. The JTMS ensures that any non-persistent changes made to the blackboard remain in memory only as long as the production that added the object continues to match. Reason maintenance ensures that agents are responsive to their environments. It also embeds knowledge about the dynamics of belief change in the architecture, with the result that agent developers are freed from having to create knowledge to manage revisions to current beliefs.

Every non-persistent object in Soar's blackboard memory is subject to reason maintenance. These objects include not only individual elaborations of the state, as in the example above, but also operator selections and impasses. When an operator proposal no longer matches, Soar immediately retracts the operator object in memory, even if the operator is executing. Similarly, as soon as an impasse is resolved, the architecture retracts the impasse state that was created to solve it. This is similar to automatic constraint checking in plan-based systems. The only persistent objects in memory are those created via operator application; all other objects must be justified. However, even operator-created objects are partially justified, as Soar also includes algorithms for maintaining consistency among assertions across problem space hierarchies (Wray & Laird, 2003).

Typical rule-based systems do not include reason maintenance, meaning that every change to an agent's context must be the result of a deliberate commitment. This requirement is one source of the perception that rule-based systems are generally brittle and inflexible, because they over-commit to particular courses of action. However, there are alternative approaches to re-assessment of beliefs. For example, ACT-R uses an

activation and decay mechanism that provides a functionally similar role, allowing elements to disappear from active memory after their activation falls below threshold (Anderson & Lebiere, 1998). This ACT-R mechanism has been used within EASE, a variant of Soar (Chong, 2003).

While implementations of reason maintenance within Soar can sometimes make Soar agents overly reactive to their environments, they guarantee Soar agents take persistent actions only when the agent internal state is fully entailed and consistent with external perceptions. Further, they encourage the development of fully reentrant agent programs, so that an agent can generally recover from interruption and resume its activity if warranted (Wray & Laird, 2003).

3.3.3 Preference-based deliberation

An agent in a dynamic environment must be able to deliberate and commit to goals. Soar balances automatic reason maintenance within the decision cycle with the deliberate selection of operators. Assertions that result from deliberation (i.e., operator applications) persist independently of reason maintenance. A Soar pilot agent could commit to a particular course based on a target's position at a particular point in time. Even as the target's position changed, the agent would remember its previously derived course.

In RBS, individual rules are the operators. Because the precondition and action components of the operator are implemented as separate rules, Soar agents recognize available options and reason about which option to take. Although this separation may appear to require more productions, in practice it can also result in *fewer* total productions. A single precondition production can pair with any number of action

productions (and vice versa). In contrast, when precondition and action combine in a single rule, as in RBS, the agent needs rules for every possible combination of pre-condition and action, leading to a potential combinatorial explosion in rules.

3.3.4 Automatic subgoaling and task decomposition

In some cases, an agent may find it has no available options or has conflicting information about its options. Soar responds to these impasses and automatically creates a new problem space, in which the desired goal is to resolve the impasse. The agent can now bring new knowledge to bear on the problem. It might use planning knowledge to consider the future and determine an appropriate course for this particular situation. It might compare this situation to others it knows about and, through analogy, decide on a course of action. The full range of reasoning and problem solving methods available to the agent can be brought to bear to solve the problem indicated by the particular impasse. However, these methods must be encoded by the agent developer.

Automatic subgoaling gives Soar agents the ability to reason about their own reasoning. Thus, Soar agents can use identical knowledge representations both to act in the world (push the fire button) and reason about actions (what will happen if I push the fire button?). This contrasts with RBS, where there is typically only a single state, making it difficult to use the same rules in multiple contexts.

Automatic subgoaling enables task decomposition. At each step in decomposition, the agent is able to focus its knowledge on the particular options at just that level, filtering considerations at other levels. Automatic subgoaling leads to a hierarchy of distinct states; agents can use multiple problem spaces simultaneously, and knowledge can be

created that is specific to the unique states. This process of hierarchical decomposition narrows a potentially exponential number of considerations into a much smaller set of choices (Erol, Hendler, & Nau, 1994). Moreover, the resulting knowledge base is naturally compartmentalized, providing a scalable infrastructure with which to build very large knowledge bases.

3.3.5 Adaptation via generalization of experience

Knowledge search refers to the process of searching one's knowledge base to attempt to find knowledge representations relevant to a given situation. *Problem search* refers to the deliberate attempt to solve a problem by analyzing the situation, considering and weighing alternative responses, etc. In general, Soar performs knowledge search at the architecture level, employing the Rete match process, while problem search engages both the architecture and the production knowledge representations. A fundamental assumption in Soar is that knowledge search should be less expensive than problem search, because knowledge search is an "inner loop" used in the problem search process (Laird & Newell, 1993).

Soar's built-in learning mechanism, *chunking* (Laird et al., 1986), converts the results of problem search within an impasse to new production representations that summarize the problem search that occurred within the impasse. Once the agent comes to a decision that resolves the impasse, chunking generates a new production that has as conditions those memory objects that were referenced in the solution of the impasse and as actions the result of the problem solving in the impasse state. This process results in new knowledge that will allow the agent to avoid a similar impasse. For example, without any preference knowledge, Soar will reach an operator-tie impasse after moving the top block to the table in

Figure 1. In the resulting impasse state, Soar might simulate the action of the two operators and recognize that re-stacking the block should be rejected in this situation because it undoes the previous action, resulting in a cycle. When the impasse is resolved,

Soar will create a new production that rejects operators that undo a preceding action.[2]

Soar's chunking mechanism is fully integrated within the architecture, pervasive (it automatically applies to all reasoning) and flexible (it can be used to realize a variety of different kinds of adaptation). Because the learning algorithm is an integral part of the overall system, Soar also provides a structure that addresses when learning occurs (when impasses are resolved), what is learned (a summarization of impasse processing), and why learning occurs (a preference for knowledge search over problem search). The drawback of Soar's learning mechanism is that all higher-level learning styles must be realized within the constraints imposed by Soar's basic learning mechanism. For example, while chunking can be used for knowledge level-learning (Dietterich, 1986; Rosenbloom & Aasman, 1990), achieving this new learning generally requires the execution of a multi-step inductive reasoning algorithm carefully constructed to provide the desired result (Young & Lewis, 1999). Thus, while Soar provides constraint in integrating multiple learning methods with behavior, realizing any individual learning style is often more straightforward in a typical RBS.

Chunking is wholly focused on caching the results of problem search into new knowledge because of the assumption that knowledge search is less expensive. Not all systems share this assumption. For example, in ACT-R, models repeatedly consider whether to attempt to retrieve an object from declarative memory (knowledge search) or to construct the desired object (problem search). In ACT-R, declarative retrieval depends

[2] Conceptually, this description is correct. In practice, generating a chunk with the correct function, at the right level of generality, is often highly dependent on the developer designed knowledge representations. Creating agents that learn automatically is more a design art than a wholly automatic function of the architecture.

on processes other than matching, such as the activation of the declarative memory objects and the perceived utility of the rule attempting retrieval. Soar's retrieval is based on complete matching and its efficiency derives from the underlying Rete match algorithm. However, unless one chooses object representations carefully, chunking can result in rules that are computationally expensive to match (Tambe, Newell, & Rosenbloom, 1990). As Soar evolves and additional mechanisms complement or augment the Rete match, the general assumption of preferring knowledge search to problem search may need to be reconsidered.

4 Listening to the Architecture: Comparing Soar to BDI

Earlier, we suggested that Soar biases solutions to behavior representation problems in unique ways. This section explores some of the repercussions of the Soar approach and contrasts Soar solutions to those within the Beliefs-Desires-Intentions (BDI) framework (Bratman, 1987; Wooldridge, 2000). Because both BDI and Soar can be viewed as alternatives for the implementation of knowledge-intensive, multiagent systems (Jones & Wray, 2004), this comparison highlights some of the tradeoffs one encounters when using Soar to develop multiagent systems.

While the control loops of Soar and BDI are similar, the representations and processes comprising BDI architectures are quite different from those of Soar. For example, BDI architectures do not make an explicit distinction between justified assertions and persistent assertions. Instead, they usually use some form of belief revision (Gardenfors, 1988). However, the most important difference between Soar and BDI is Soar's assumption of parsimony and the consequences of this assumption on knowledge representations.

Soar accomplished all deliberation via a single representation: the operator. In contrast, BDI specifies multiple representations that are mediated by deliberation, including desires, intentions, plans, and, in some cases, beliefs. For each of these representations, there can be a distinct mechanism of choice. Committing to an intention may use some decision-theoretic computation, while committing to a particular plan could result from a simple table lookup. Similarly, the process of reconsideration (deciding if a commitment should be continued), can also be tailored to the specific representation and its role in the agent system (Crossman et al., 2004; Wooldridge, 2000).

Because Soar uses only operators for deliberation, there is one mechanism each for commitment (the decision procedure) and reconsideration (reason maintenance). Essentially, the reconsideration algorithms assume it is cheaper to retract and repeat some problem search, if necessary, rather than attempt to decide whether some deliberate selection should continue to be supported (Wray & Laird, 2003).

This uniform approach to reconsideration has important consequences for the design of agent systems. For example, because reconsideration will interrupt a deliberate process as soon as a potential inconsistency is detected, no deliberate step can be assumed to directly follow another. Thus, (robust) Soar systems must be designed to be reentrant at every step in execution. These reentrancy requirements contrast with some BDI implementations, which enable the execution of plans of arbitrary length or even traditional, serial procedures (Howden, Rönnquist, Hodgson, & Lucas, 2001), within a single pass of the agent control loop. BDI systems thus provide immediate power in the

representation of complex procedures and plans, but at the cost of having to manage the execution of these procedures with other (potentially competing) procedures. The lack of an explicit plan representation in Soar lends flexibility in terms of plan execution (including interleaved execution with other plans). However, it also requires that a developer consider plan representation in the design of agent knowledge and plan for interruption and reentrant execution without exception.

Another consequence of the uniformity in representation in Soar is that any new representations must be implemented as symbolic representations of knowledge, rather than at the architecture level. Within the BDI community, there is presently a focus on extending the basic BDI representations of beliefs, desires, and intentions, to other, multiagent-oriented representations, such as teams, values, and norms (e.g., see Beavers & Hexmoor, 2002; Broersen, Dastani, Hulstijn, Huang, & van der Torre, 2001 and other chapters in this volume). Within a BDI framework, these new representations must be integrated with the other representations and processes used for commitment and reconsideration, which leads to exploration at the architecture level. Within Soar, operators and the processes of the decision cycle define the basic architectural mechanisms. Algorithms that make decisions about new representations map to different Soar operators, where they are integrated by the built-in conflict resolution procedure. For example, the Soar-Teamwork (STEAM) model mentioned previously[3] (Tambe, 1997) annotated individual Soar operators with team goals, to enable team-specific processing for each operator. The drawback of the Soar approach is that the architecture will not readily support decision and conflict resolution that does not map easily to the

[3] Tambe and colleagues have extended their original Soar-Teamwork model to more general computational approach to teamwork within the BDI framework; a chapter in this volume introduces this work.

architectural decision process. For example, to make decision-theoretic communication decisions, STEAM relies on extra-architectural procedures.

Of course, one could have used Soar operators to make those decision-theoretic calculations. One of the fundamental tensions that arises in "listening to the Soar architecture" is whether actually to follow its advice. In general, listening to Soar requires mapping any deliberate step to an operator. In the most recent version of Soar, any sequence of deliberate steps can be interrupted, which encourages fine-grained, single-step operator implementations (Wray & Laird, 2003). However, because elaboration occurs as a loop within the decision, splitting the execution of a procedure over multiple deliberation steps can seem inefficient. Many Soar developers spend significant time and effort attempting to implement computations within the elaboration cycle of Soar that would be trivial to accomplish via deliberation. In theory, Soar also resolves this dilemma by compiling the results of multi-step sequences into more compact, efficient representations. However, in practice, using chunking in performance environments remains difficult, even after the successful resolution of a number of recognized interactions between chunking and interaction with external environments (Wray & Jones, 2001; Wray, Laird, & Jones, 1996). In summary, it is often difficult to discern if an implementation challenge is the result of not "listening to the architecture" closely enough, a flaw in the current implementation, or an inadequacy in the theory. Of course, these challenges are not unique to Soar and all approaches to computational intelligence must be evaluated both in the context of idealized theories and implementations of those theories which may not fully live up to those ideals.

5 Summary

The Soar project represents an attempt to articulate a theory of general intelligence through a specific computational model, an architecture foundation that implements the computational model, and artifacts that guide evaluation and refinement of the theory. Defining characteristics of the Soar computational model include pattern-directed processing, least-commitment execution, subgoaling and task decomposition, knowledge-mediated conflict resolution and learning integrated with performance.

A downside of the Soar approach is that, by specifying very general mechanisms, it under specifies some capabilities that must be built into intelligent agents. Most of an agent's competence arises from the encoded knowledge representations (i.e., the set of rules) that Soar's mechanisms operate on. Thus, agent knowledge representations must be created to realize any high-level intelligent capability. For instance, while Soar has been used to build planning systems, in comparison to other AI planning systems, Soar offers little immediately evident power. Soar only specifies very low-level constraints on how planning can occur, so Soar agent designers must develop their own plan languages and algorithms, while these are provided in most planning systems. However, Soar does provide a natural, scalable methodology for integrating planning with plan execution, as well as natural language understanding, reasoning by analogy, etc. By focusing on a uniform substrate that allows any available knowledge to mediate any decision, Soar provides a tool with which to realize integrated approaches. Soar therefore trades off powerful, but often overly constrained processes for the flexibility to integrate solutions, and this integration has been demonstrated across a broad spectrum of intelligent systems applications.

6 Additional Resources

Soar is supported by a community of academic and industry researchers, developers, and users. The Soar homepage (http://sitemaker.umich.edu/soar) includes links to the executable and source versions of the Soar software, tutorials that introduce Soar as a programmable system, a Soar programming manual, and tools for creating and debugging Soar programs. Soar is a freely-available, open-source project and continuing architecture development is hosted at Source Forge (http://sourceforge.net/projects/soar/). The multi-site, multinational Soar community interacts via the Soar mailing list (see http://sourceforge.net/mail/?group_id=65490 for subscription information) and a yearly "Soar Workshop," usually held in June in Ann Arbor, Michigan, USA. The Soar Frequently Asked Questions (FAQ) (http://acs.ist.psu.edu/soar-faq/soar-faq.html) answers common questions about the theory, software architecture, and programming of Soar. Theoretical motivations and descriptions of the basic principals of Soar may be found in *The Soar Papers* (Rosenbloom, Laird, & Newell, 1993).

7 References

Altmann, E. M., & Gray, W. D. (2002). Forgetting to remember: The functional relationship of decay and interference. *Psychological Science, 13*(1), 27-33.

Altmann, E. M., & John, B. E. (1999). Episodic Indexing: A model of memory for attention events. *Cognitive Science, 23*(2), 117-156.

Anderson, J., & Lebiere, C. (1998). *The Atomic Components of Thought*: Lawrence Erlbaum.

Beavers, G., & Hexmoor, H. (2002). Adding values, obligations, and norms to BDI framework. *Cognitive Science Quarterly.*

Bratman, M. (1987). *Intentions, Plans, and Practical Reason*. Cambridge, MA: Harvard University Press.

Broersen, J., Dastani, M., Hulstijn, J., Huang, Z., & van der Torre, L. (2001). *The BOID Architecture.* Paper presented at the Fifth International Conference on Autonomous Agents, Montreal, Canada.

Chong, R. S. (2003). *The addition of an activation and decay mechanism to the Soar architecture.* Paper presented at the Fifth International Conference on Cognitive Modeling, Bamberg, Germany.

Chong, R. S., & Wray, R. E. (to appear). Constraints on Architectural Models: Elements of ACT-R, Soar and EPIC in Human Learning and Performance. In K. Gluck & R. Pew (Eds.), *Modeling Human Behavior with Integrated Cognitive Architectures: Comparison, Evaluation, and Validation*.

Crossman, J., Wray, R. E., Jones, R. M., & Lebiere, C. (2004). *A High Level Symbolic Representation for Behavior Modeling.* Paper presented at the 2004 Behavioral

Representation in Modeling and Simulation, Arlington, VA.

Dietterich, T. G. (1986). Learning at the knowledge level. *Machine Learning, 1*, 287-315.

Doorenbos, R. B. (1994). *Combining left and right unlinking for matching a large number of learned rules.* Paper presented at the Twelfth National Conference on Artificial Intelligence (AAAI-94), Seattle, Washington.

Doyle, J. (1979). A truth maintenance system. *Artificial Intelligence, 12*, 231-272.

Erol, K., Hendler, J., N, & Nau, D. S. (1994). *HTN planning: Complexity and expressivity.* Paper presented at the 12th National Conference on Artificial Intelligence.

Fikes, R., & Nilsson, N. (1971). STRIPS: A new approach in the application of theorem proving to problem solving. *Artificial Intelligence, 2*(3-4), 189-208.

Fikes, R., & Nilsson, N. (1993). STRIPS: A retrospective. *Artificial Intelligence, 59*(1-2), 227-232.

Forbus, K., & deKleer, J. (1993). *Building Problem Solvers*. Cambridge, MA: MIT Press.

Forgy, C. L. (1981). *OPS5 User's Manual* (CMU-CS-81-135): Computer Science Dept. Carnegie-Mellon University.

Forgy, C. L. (1982). RETE: A fast algorithm for many pattern/many object pattern matching problem. *Artificial Intelligence, 19*, 17-37.

Friedman-Hill, E. (2003). *Jess in Action: Rule-Based Systems in Java*: Manning Publications Company.

Gardenfors, P. (1988). *Knowledge in Flux*. Cambridge, MA: MIT Press.

Giarratano, J. C. (1998). *Expert Systems: Principles and Programming* (Third ed.): Brooks Cole.

Howden, N., Rönnquist, R., Hodgson, A., & Lucas, A. (2001). *JACK: Summary of an Agent Infrastructure.* Paper presented at the Workshop on Infrastructure for Agents, MAS, and Scalable MAS at the Fifth International Conference on Autonomous Agents, Montreal, Canada.

Huffman, S. B., & Laird, J. E. (1995). Flexibly instructable agents. *Journal of Artificial Intelligence Research, 3*, 271-324.

Jones, R. M., Laird, J. E., Nielsen, P. E., Coulter, K. J., Kenny, P. G., & Koss, F. V. (1999). Automated Intelligent Pilots for Combat Flight Simulation. *AI Magazine, 20*(1), 27-42.

Jones, R. M., & Wray, R. E. (2004). *Comparative Analysis of Frameworks for Knowledge-Intensive Intelligent Agents.* Paper presented at the AAAI Fall Symposium Series on Achieving Human-level Intelligence through Integrated Systems and Research, Alexandria, VA.

Kieras, D. E., & Meyer, D. E. (1997). An overview of the EPIC architecture for cognition and performance with application to human-computer interaction. *Human-Computer Interaction, 12*, 391-438.

Laird, J. E., & Congdon, C. B. (2004). *The Soar Users' Manual: Version 8.5*. Ann Arbor, MI: University of Michigan.

Laird, J. E., & Newell, A. (1993). A universal weak method. In P. S. Rosenbloom & J. E. Laird & A. Newell (Eds.), *The Soar Papers: Research on Integrated Intelligence* (Vol. 1, pp. 245-292). Cambridge, MA: MIT Press.

Laird, J. E., Newell, A., & Rosenbloom, P. S. (1987). Soar: An architecture for general intelligence. *Artificial Intelligence, 33*(3), 1-64.

Laird, J. E., & Rosenbloom, P. S. (1995). The evolution of the Soar cognitive architecture. In D. Steir & T. Mitchell (Eds.), *Mind Matters*. Hillsdale, NJ:

Lawrence Erlbaum Associates.

Laird, J. E., Rosenbloom, P. S., & Newell, A. (1986). Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning, 1*(1), 11-46.

Laird, J. E., Yager, E. S., Hucka, M., & Tuck, M. (1991). Robo-Soar: An integration of external interaction, planning and learning using Soar. *Robotics and Autonomous Systems, 8*(1-2), 113-129.

Lakatos, I. (1970). Falsification and the methodology of scientific research programmes. In I. Lakatos & A. Musgrave (Eds.), *Criticism and the Growth of Knowledge* (pp. 91-196). Cambridge, UK: Cambridge University Press.

Lehman, J. F., Dyke, J. V., & Rubinoff, R. (1995, May). *Natural language processing for Intelligent Forces (IFORs): Comprehension and Generation in the Air Combat Domain.* Paper presented at the Fifth Conference on Computer Generated Forces and Behavioral Representation, Orlando, FL.

Lehman, J. F., Lewis, R. L., & Newell, A. (1998). Architectural influences on language comprehension. In Z. Pylyshyn (Ed.), *Cognitive Architecture*. Norwood, NJ: Ablex.

Miller, C. S., & Laird, J. E. (1996). Accounting for graded performance within a discrete search framework. *Cognitive Science, 20*, 499-537.

Nelson, G., Lehman, J. F., & John, B. (1994). *Integrating Cognitive Capabilities in a Real-Time Task.* Paper presented at the Proceedings of the Sixteenth Annual Conference of the Cognitive Science Society.

Newell, A. (1973). Production Systems: Models of Control Structures. In W. Chase (Ed.), *Visual Information Processing*. New York: Academic Press.

Newell, A. (1980a). Physical symbol systems. *Cognitive Science, 4*, 135-183.

Newell, A. (1980b). Reasoning, problem solving and decision processes: The problem space as a fundamental category. In R. Nickerson (Ed.), *Attention and Performance VIII*. Hillsdale, NJ: Erlbaum.

Newell, A. (1982). The knowledge level. *Artificial Intelligence, 18*(1), 82-127.

Newell, A. (1990). *Unified Theories of Cognition*. Cambridge, Massachusetts: Harvard University Press.

Newell, A., Yost, G. R., Laird, J. E., Rosenbloom, P. S., & Altmann, E. M. (1991). Formulating the Problem Space Computational Model. In R. F. Rashid (Ed.),

CMU Computer Science: A 25th Anniversary Commemorative (pp. 255-293): ACM Press/Addison-Wesley.

Nuxoll, A., & Laird, J. E. (2004). *A Cognitive Model of Episodic Memory Integrated with a General Cognitive Architecture.* Paper presented at the International Conference on Cognitive Modeling, Pittsburgh, PA.

Pearson, D. J., Huffman, S. B., Willis, M. B., Laird, J. E., & Jones, R. M. (1993). A Symbolic Solution to Intelligent Real-Time Control. *Robotics and Autonomous Systems, 11*, 279-291.

Pearson, D. J., & Laird, J. E. (1998). Toward incremental knowledge correction for agents in complex environments. *Machine Intelligence, 15*.

Pearson, D. J., & Laird, J. E. (2003). *Example-driven diagrammatic tools for rapid knowledge acquisition.* Paper presented at the Visualizing Information in Knowledge Engineering Workshop, Knowledge Capture 2003, Sanibel Island, FL.

Rosenbloom, P. S., & Aasman, J. (1990). *Knowledge level and inductive uses of chunking.* Paper presented at the Eighth National Conference on Artificial

Intelligence.

Rosenbloom, P. S., Laird, J. E., McDermott, J., Newell, A., & Orciuch, E. (1985). R1-Soar: An experiment in knowledge-intensive programming in a problem-solving architecture. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 7*, 561-569.

Rosenbloom, P. S., Laird, J. E., & Newell, A. (Eds.). (1993). *The Soar Papers: Research on Integrated Intelligence*. Cambridge, MA: MIT Press.

Russell, S., & Norvig, P. (1995). *Artificial Intelligence: A Modern Approach*. Upper Saddle River, NJ: Prentice-Hall.

Tambe, M. (1997). Towards Flexible Teamwork. *Journal of Artificial Intelligence Research (JAIR), 7*, 83-124.

Tambe, M., Newell, A., & Rosenbloom, P. S. (1990). The problem of expensive chunks and its solution by restricting expressiveness. *Machine Learning, 5*, 299-348.

Taylor, G., & Wray, R. E. (2004). *Behavior Design Patterns: Engineering Human Behavior Models.* Paper presented at the 2004 Behavioral Representation in Modeling and Simulation Conference, Arlington, VA.

van Lent, M., & Laird, J. E. (2001). *Learning procedural knowledge through observation.* Paper presented at the International Conference on Knowledge Capture, Victoria, British Columbia, Canada.

Wallace, S., & Laird, J. E. (2003). *Behavior Bounding: Toward Effective Comparisons of Agents & Humans.* Paper presented at the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico.

Washington, R., & Rosenbloom, P. S. (1993). Applying Problem Solving and Learning to Diagnosis. In P. S. Rosenbloom & J. E. Laird & A. Newell (Eds.), *The Soar Papers: Research on Integrated Intelligence* (Vol. 1, pp. 674-687). Cambridge, MA: MIT Press.

Wooldridge, M. J. (2000). *Reasoning about Rational Agents*. Cambridge, MA: MIT Press.

Wray, R. E., & Jones, R. M. (2001). *Resolving Contentions between Initial and Learned Knowledge.* Paper presented at the Proceedings of the 2001 International Conference on Artificial Intelligence, Las Vegas, NV.

Wray, R. E., & Laird, J. E. (2003). An architectural approach to consistency in

hierarchical execution. *Journal of Artificial Intelligence Research, 19*, 355-398.

Wray, R. E., Laird, J. E., & Jones, R. M. (1996). *Compilation of Non-Contemporaneous Constraints.* Paper presented at the Proceedings of the Thirteenth National Conference on Artificial Intelligence, Portland, Oregon.

Wray, R. E., Laird, J. E., Nuxoll, A., Stokes, D., & Kerfoot, A. (2004). *Synthetic Adversaries for Urban Combat Training.* Paper presented at the 2004 Innovative Applications of Artificial Intelligence Conference, San Jose, CA.

Wray, R. E., Lisse, S., & Beard, J. (to appear). Investigating Ontology Infrastructures for Execution-oriented Autonomous Agents. *Robotics and Autonomous Systems*.

Young, R., & Lewis, R. L. (1999). The Soar Cognitive Architecture and Human Working Memory. In A. Miyake & P. Shah (Eds.), *Models of Working Memory: Mechanisms of Active Maintenance and Executive Control* (pp. 224-256). Cambridge, UK: Cambridge University Press.

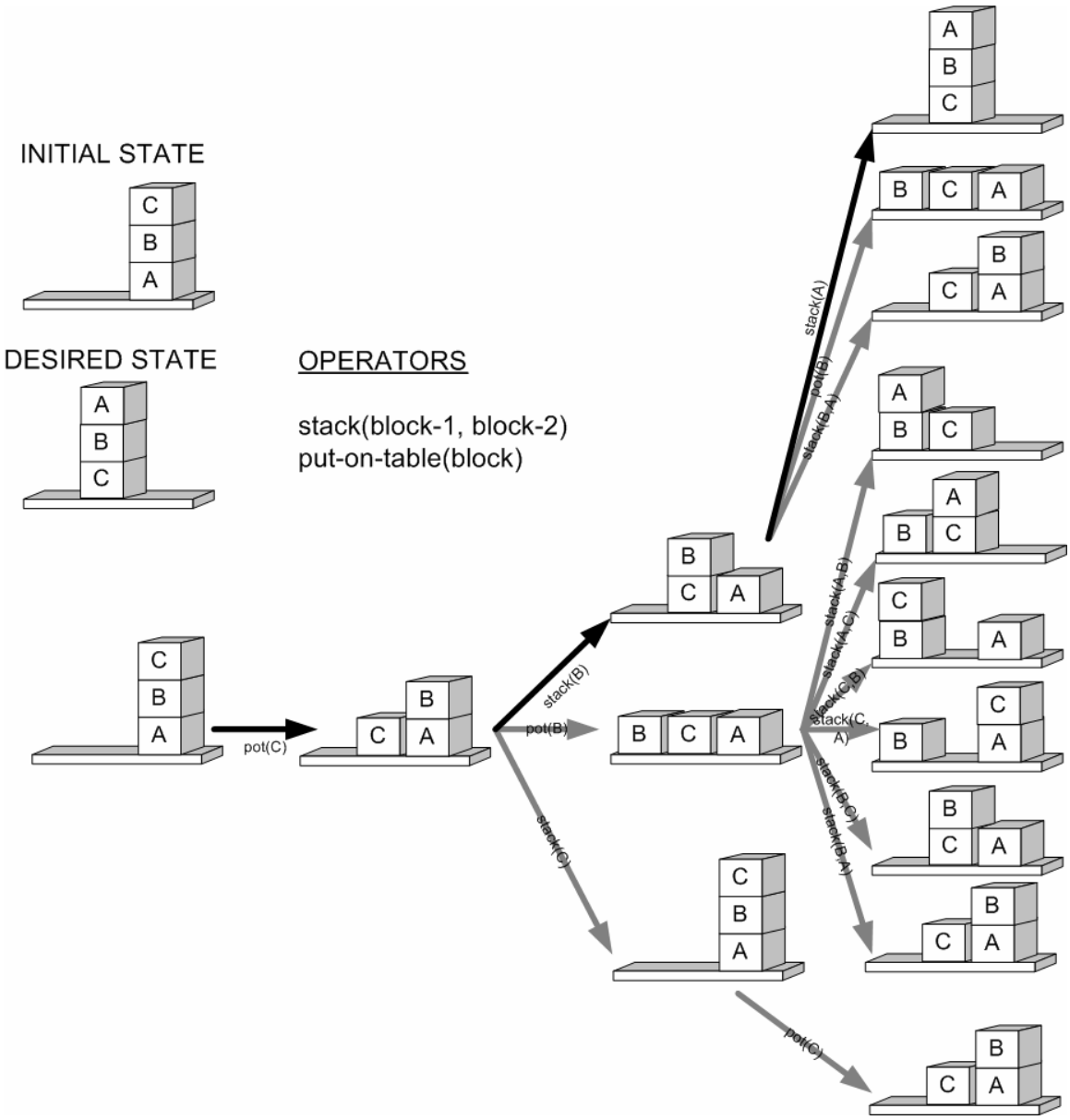

Figure 1: Example problem space for the blocks world domain. The solution path from the initial to the desired state is illustrated with dark arrows.

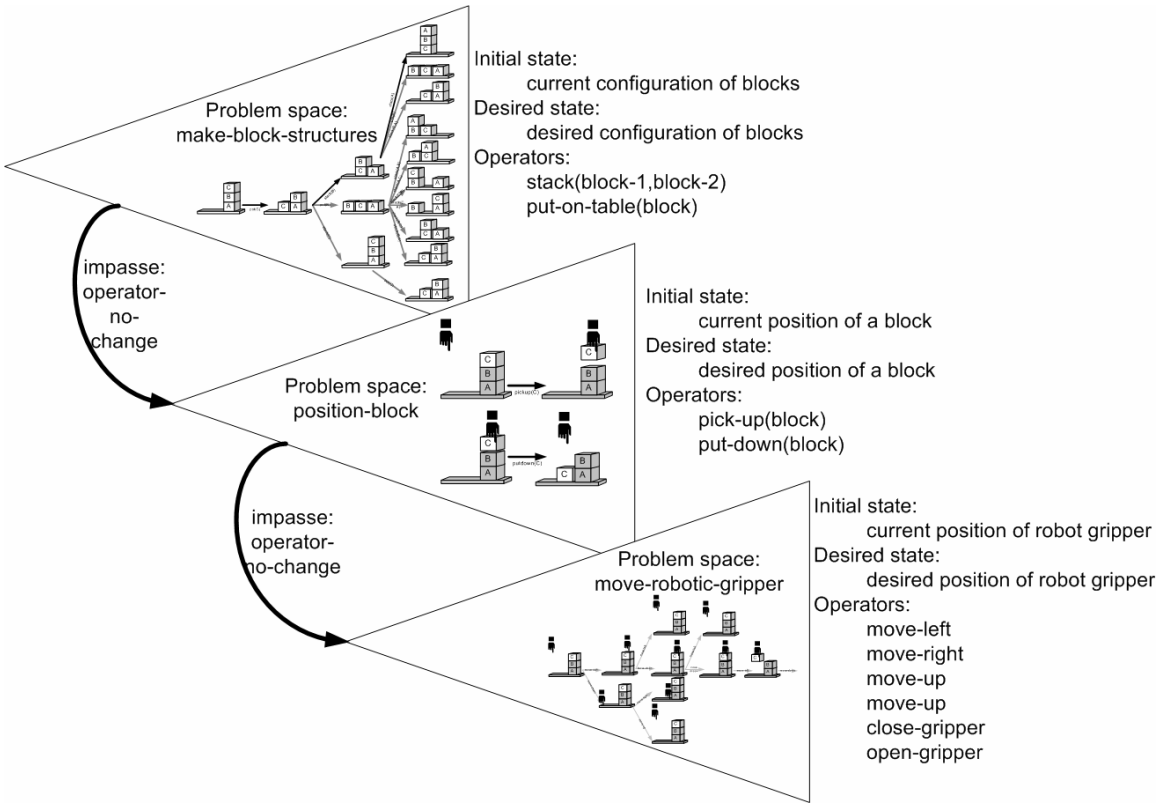

Figure 2: Implementing elements of a problem space with additional problem spaces. The initial state in the *blocks* problem space is defined by the current operator in the top, *structure* problem space. Operators in the *blocks* space pick up and put down individual blocks. This problem space is then implemented by the *gripper* problem space, which moves the gripper in one of four directions, and opens and closes the robotic hand.

```
production structure*elaborate*state*in-position*on-table
    (state <s> ^problem-space.name make-block-structures
               ^top-state.io.input-link <il>
               ^current-goal (^relation on ^top <bl>
                                           ^bottom <t>))
    (<il> ^on (^top <bl> ^bottom <t>)
          ^table <t>)
    -->
    (<s> ^block-in-position <bl> )
```

Figure 3: Example of a Soar production. This production tests if a block (represented on Soar's input-link) meets the constraints of a desired state of the **structure** problem space, which is that the block is on the table. The desired state is represented here by an object called **current-goal**. The action of the production is to add a new object to the blackboard memory, which indicates the block is in desired position.

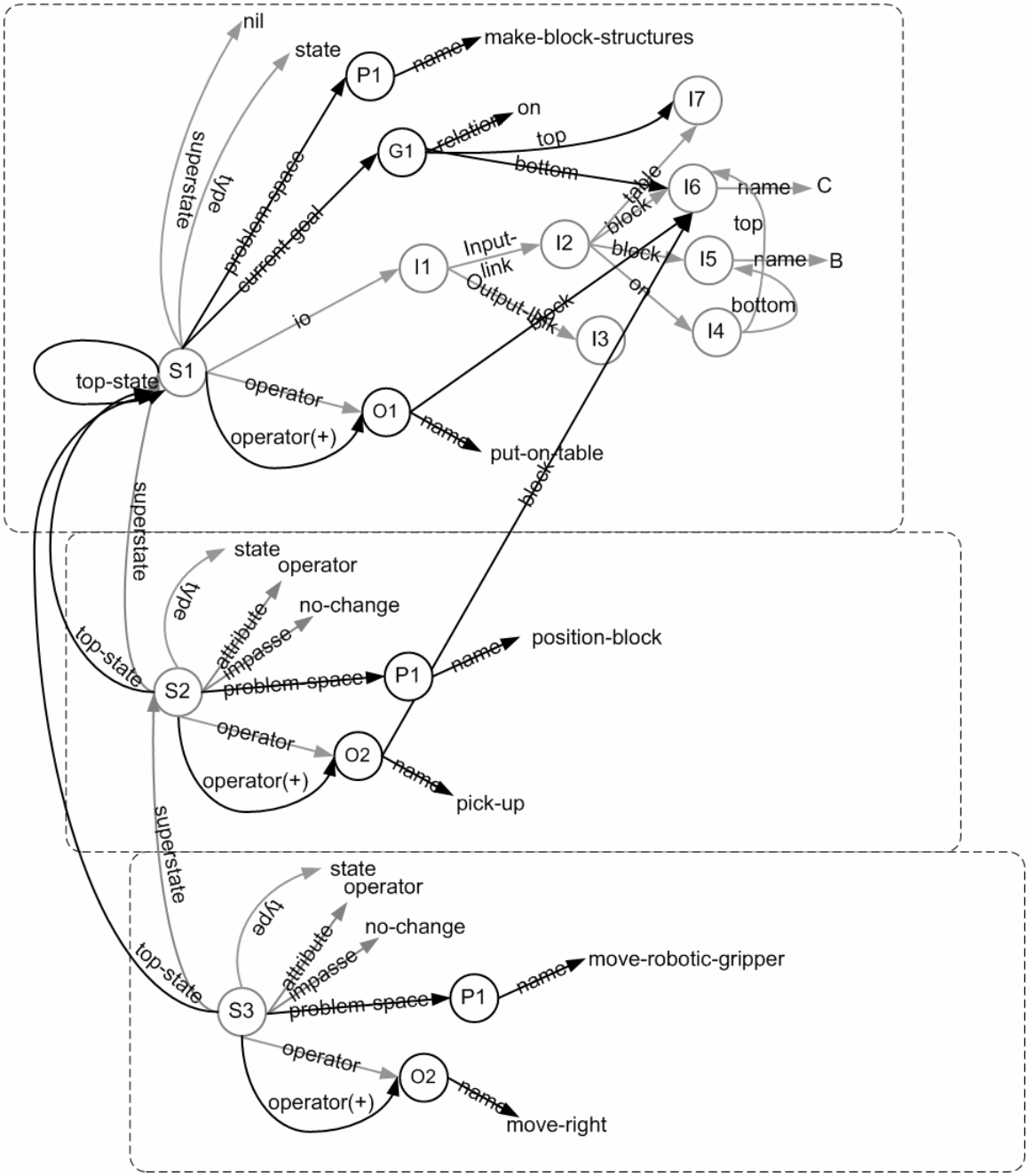

Figure 4: Example of Soar's blackboard memory. Each object consists of the triple (identifier attribute value) and can be traced to the root state object. Soar automatically creates some objects, like impasses and operators; architecture-created objects are shown in grey and objects created by productions are shown in black.

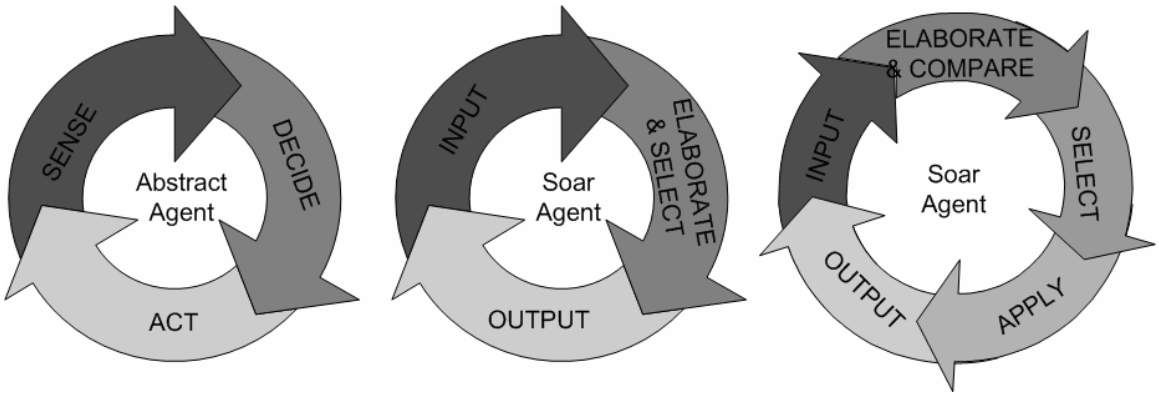

Figure 5: Common representation of an abstract agent as a cycle of perception, reasoning, and action (left), a high level view of Soar's sense-decide-act loop (middle), and a more-detailed Soar representation (right).

Object	Impasse	Description
State	No-change	No operators appear to be acceptable in the current state. The goal in the child problem space is to find an operator to apply in the parent state.
Operator	No-change	An operator appeared to be applicable in the current state but selecting it does not result in changes to the state. The goal in the child problem space is to implement the operator, which may include decomposition (as in Figure 2) or correcting problems in the operator representation in the parent problem space (Pearson & Laird, 1998).
Operator	Tie	Two (or more) operators are applicable to the current state and the parent problem space lacks knowledge to determine which should be chosen. The goal in the child problem space is to compare the options and make a decision about which options should be preferred.
Operator	Conflict	Two (or more) operators are applicable to the current state but the problem space has conflicting knowledge about which operator to pursue. The goal in the child problem space is to resolve the knowledge conflict.

Table 1: The relationship between *impasses* in the parent problem space and goals in a child problem space.

Soar representation	Name	Description
O1 +	Acceptable	Indicates that operator O1 is an acceptable candidate for selection
O1 >	Best	Indicates O1 is the “best” candidate
O1 > O2	Better	Indicates operator O1 is a better candidate than operator O2
O1 !	Require	Indicates operator O1 is required for the (impasse) goal to be achieved
O1 ~	Prohibit	Indicates selection of operator O1 will cause the (impasse) goal to be unable to be achieved
O1 =	Indifferent	Indicates operator O1 can be chosen randomly from the set of all candidates with indifferent preferences.
B1 -	Reject	Indicates object B1 is not a candidate for selection

Table 2: Examples of Soar preferences.