A GENTLE INTRODUCTION TO SOAR, AN ARCHITECTURE FOR HUMAN COGNITION: 2006 UPDATE*

JILL FAIN LEHMAN, JOHN LAIRD, PAUL ROSENBLOOM

1. INTRODUCTION

Many intellectual disciplines contribute to the field of cognitive science: psychology, linguistics, anthropology, and artificial intelligence, to name just a few. Cognitive science itself originated in the desire to integrate expertise in these traditionally separate disciplines in order to advance our insight into cognitive phenomena — phenomena like problem solving, decision making, language, memory, and learning. Each discipline has a history of asking certain types of questions and accepting certain types of answers. And that, according to Allen Newell, a founder of the field of artificial intelligence, is both an advantage and a problem.

The advantage of having individual disciplines contribute to a study of cognition is that each provides expertise concerning the questions that have been at the focus of its inquiry. Expertise comes in two packages: descriptions of regularities in behavior, and theories that try to explain those regularities. For example, the field of psycholinguistics has been responsible for documenting a regularity called the *garden path phenomenon* which contrasts sentences such as (a) below, that are very easy for people to understand, with sentences like (b), that are so difficult that most people believe they are ungrammatical (they're not, but their structure leads people to misinterpret them):

(a) Without her contributions we failed.
(b) Without her contributions failed to come in.

In addition to providing examples of garden path sentences and experimental evidence demonstrating that people find them nearly impossible to understand, psycholinguists have also constructed theories of why they are problematic, (e.g. Gibson, 1990; Pritchett, 1988). Indeed, psycholinguists are interested in phenomena like these because they help to constrain theories of how humans understand language; such a theory should predict that people will have problems on sentences like (b) but not on sentences like (a).

Psychology has also given us descriptions and theories of robust regularities, i.e. behaviors that all people seem to exhibit. It has contributed regularities about motor behavior (e.g. Fitts' Law (Fitts, 1954)), which predicts how long it will take a person to move a pointer from one place to a target location as a function of the distance to be traveled and the size of the target), about item recognition (e.g. that the time to decide whether a test item was on a memorized list of items increases linearly with the length of the list (Sternberg, 1975)), and about verbal learning (e.g. if an ordered list of items is memorized by repeated exposure, then the items at the ends of the list are learned before the items in the middle of the list (Tulving, 1983)), among others. The other

* This version was created by John Laird in January, 2006. It updates the original paper, which was published in Sternberg & Scarborough (1996). The original paper described Soar 6, while this version describes Soar 9, which is under development at the University of Michigan. The major changes for Soar 6 to 9 include the removal of architectural support for the selection of the current goal and problem space, changes to the maintenance of substates (Wray & Laird, 2003), and the addition of new long-term memories and learning mechanisms.
This material is based upon work supported by the National Science Foundation under Grant No. 0413013.

disciplines mentioned above have also described and theorized about regularities in human cognition. How can the expert contributions of all these fields be a problem?

The problem arises not because of the regularities, but because of the theories. Each individual discipline really contributes what Newell called *microtheories* — small pieces of the big picture developed without the constraint of having to fit in with all the other pieces. It is true that the structures and mechanisms that underlie any particular regularity need not underlie every other regularity. But they must at least be compatible with the structures and mechanisms underlying those other regularities; after all, the regularities we observe in human cognition are all produced by a single system, the mind. If we think about cognition as a big picture, then a microtheory is a way of cutting a portion of that picture into a jigsaw puzzle. Each theory may cut up its own portion, even a portion that overlaps with another theory's, in a different way. So, when each discipline throws its set of pieces out on the table, how do we know that there is any set of pieces that will allow us to recover the big picture? The only way, Newell argues, is to go ahead and try to put the whole picture together, to try to build *unified theories of cognition* (UTCs) (Newell, 1990). In the early 1980's, Soar was developed to be a system that could support multiple problem solving methods for many different problems. In the mid 1980's, Newell and many of his students (including the authors of this paper) began working on Soar as a candidate UTC.

As we said above, each of the disciplines that contributes to cognitive science has a history of asking certain types of questions and accepting certain kinds of answers. This is just another way of saying disciplines differ as to what constitutes a theory. For some, a theory is a set of equations, for others it is a computer program, for still others it is a written narrative that makes reference to some objects and ideas that are new in the context of other objects and ideas that are generally accepted. Although the form may differ, the purpose a theory serves — to answer questions — does not. Like Newell, our backgrounds are in both artificial intelligence and psychology. For that reason, the form of theory we are particularly interested in is mechanistic, that is, a theory as an account of underlying mechanisms and structures. Moreover, because artificial intelligence has its roots in computer science, we prefer to state our theory both narratively and as a working computer program. So, for us at least, working on a unified theory of cognition means trying to find a set of computationally-realizable mechanisms and structures that can answer all the questions we might want to ask about cognitive behavior. A key piece of the puzzle, we believe, lies in the idea of architecture.

2. THE IDEA OF ARCHITECTURE

The idea of architecture is not new. In working with computers, we often describe and compare hardware architectures: the set of choices the manufacturer makes for a particular computer's memory size, commands, processor chip, etc. As a trip to any computer store will demonstrate, many hardware configurations are possible. Differences among hardware architectures reflect, in part, designs that are intended to be optimal under different assumptions about the software that architecture will process. Thus, once the many decisions about the hardware components and how they interact are made, the resulting architecture can be evaluated (or compared to that of another machine) only by considering how well it processes software. In other words, asking "Does machine M run applications A and B efficiently?" is a more appropriate question than "Does machine M work well?"

Just as we talk about how a particular hardware architecture processes software applications, we can also talk about how a particular software application processes a set of high-level tasks (see Figure 1). That is, we can think of a software application as also having an architecture. If we want to create documents or organize spreadsheets (two common sets of high-level tasks), we can choose among word-processing or spreadsheet applications. As in the hardware example, there are many possible application programs, with different programs designed to be optimal under different assumptions about the tasks. Which particular application architecture we choose dictates which subtasks will be easy and efficient. If, for example, our set of high level tasks is writing the chapters in a calculus book, we are likely to choose a word-processing program that has functions and commands for formatting mathematical equations. If we are simply interested in writing letters, those functions are not important.

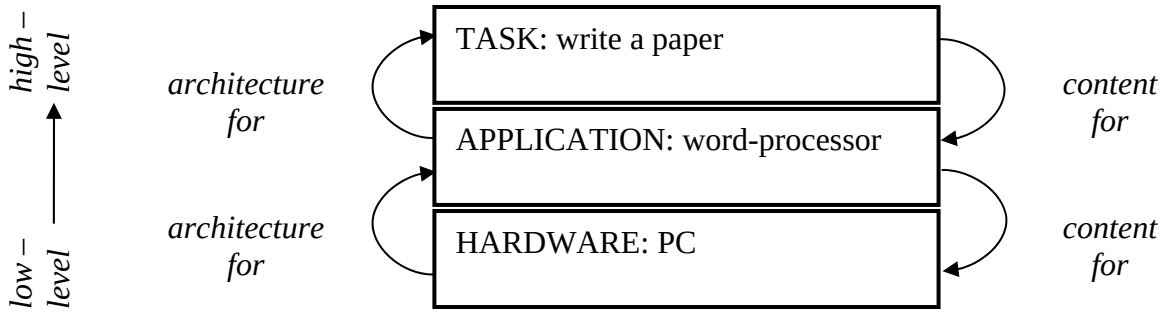

Figure 1: Levels of processing. The middle level can be viewed as both architecture for the level above and content for the level below. As an architecture, a word-processing program offers a fixed set of methods for composing lots of different documents as content.

There are two common threads that underlie these examples of architecture. The first is the idea that for any level of a complex system, we can make a distinction between the fixed set of mechanisms and structures of the architecture itself, and the content those architectural mechanisms and structures process. So, it is fixed hardware that processes software content at one level, and fixed application mechanisms that process high-level task content at the next. Another way of viewing this relationship is to note that an architecture by itself does nothing; it requires content to produce behavior. Because we'll have occasion to revisit this point periodically, it's worth being very clear:

BEHAVIOR = ARCHITECTURE + CONTENT

The second common thread running through the examples is that any particular architecture reflects assumptions on the part of the designer about characteristics of the content the architecture will process. There are, for example, many possible hardware architectures that process the same software, but machines with parallel processors will execute some software much more quickly than serial machines. Similarly, many applications will execute the same high-level task, but some provide single commands that achieve whole subtasks, while others require the user to construct sequences of commands to gain the same effect.

In general, then, the idea of architecture is useful because it allows us to factor out some common aspects of the wide array of behaviors that characterize the content. A particular architecture, that is, a particular fixed set of mechanisms and structures, stands as a theory of what is common among much of the behavior at the level above it.[1] Using this idea, we can define a *cognitive architecture* as a theory of the fixed mechanisms and structures that underlie human cognition. Factoring out what is common across cognitive behaviors, across the phenomena explained by microtheories, seems to us to be a significant step toward producing a unified theory of cognition. Thus, for most of the remainder of this chapter we will concentrate on this aspect of Soar, i.e. Soar as a cognitive architecture. As an example of a cognitive architecture, Soar is one theory of what is common to the wide array of behaviors we think of as intelligent. It is not the only such theory (see, e.g., Anderson, 1993; Kieras, Wood, and Meyer 1997, Langley and Laird 2002), but it is the one we will explore in detail.

In the sections that follow, we motivate the different structures and mechanisms of the Soar architecture by exploring its role in one concrete example of intelligent behavior. Because we are interested in a computationally-realizable theory, our exploration takes the form of constructing a computational model. To construct a model of behavior in Soar we must first understand what aspects of the behavior the architecture will support directly (Section 3), and lay the groundwork for tying the architecture to our sample content (Section 4). Then, piece by piece, we construct our model, introducing each architectural structure and mechanism in response to questions that arise from the behavior we are trying to capture (Sections 5 through 9). In Section 10 we step back and reflect on the architecture as a whole. Although we examine only one architecture and only one model based on that architecture, our broader goal is to provide both an appreciation for the power of the idea of cognitive architecture, and the tools to understand other architectures. Finally, in Section 11, we reflect briefly on where Soar stands as a candidate UTC.

3. WHAT COGNITIVE BEHAVIORS HAVE IN COMMON

To understand how any computational architecture works, we need to use it to model some behavior (remember, the architecture alone doesn't do anything). What sorts of behavior should we model with Soar? A cognitive architecture must help produce cognitive behavior. Reading certainly requires cognitive ability. So does solving equations, cooking dinner, driving a car, telling a joke, or playing baseball. In fact, most of our everyday behavior seems to require some degree of thinking to mediate our perceptions and actions. Because every architecture is a theory about what is common to the content it processes, Soar is a theory of what cognitive behaviors have in common. In particular, the Soar theory posits that cognitive behavior has at least the following characteristics (Newell, 1990):

1. **It is goal-oriented.** Despite how it sometimes feels, we don't stumble through life, acting in ways that are unrelated to our desires and intentions. If we want to cook dinner, we go to an appropriate location, gather ingredients and implements, then chop, stir and season until we've produced the desired result.

[1] The view of architecture as a theory of what is common across cognitive behaviors can be interpreted narrowly or broadly. In a modular architecture, for example, what is common to language might not be what is common to problem solving; each module would have its own architecture, its own factoring of what is common to the behaviors of that module. Our own approach, however, is to look for a minimal set of architectural components that is, as much as possible, common to all cognitive behavior. Thus, we view commonality broadly, both in our research and in this paper.

2. **It takes place in a rich, complex, detailed environment.** Although the ways in which we perceive and act on the world are limited, the world we perceive and act on is not a simple one. There are a huge number of objects, qualities of objects, actions, and so on, any of which may be key to understanding how to achieve our goals. Think about what features of the environment you respond to when driving some place new, following directions you've been given. Somehow you recognize the real places in all their detail from the simple descriptions you were given, and respond with gross and fine motor actions that take you to just the right spot, although you have never been there before.
3. **It requires a large amount of knowledge.** Try to describe all the things you know about how to solve equations. Some of them are obvious: get the variable on one side of the equal sign, move constant terms by addition or subtraction and coefficients by multiplication or division. But you also need to know how to do the multiplication and addition, basic number facts, how to read and write numbers and letters, how to hold a pencil and use an eraser, what to do if your pencil breaks or the room gets dark, etc.
4. **It requires the use of symbols and abstractions.** Let's go back to cooking dinner. In front of you sits a ten-pound turkey, something you have eaten but never cooked before. How do you know it's a turkey? You have seen a turkey before but never *this* turkey and perhaps not even an uncooked one. Somehow some of the knowledge you have can be elicited by something other than your perceptions in all their detail. We'll call that thing a symbol (or set of symbols). Because we represent the world internally using symbols, we can create abstractions. You can't stop *seeing* this turkey, but you can *think* about it as just *a* turkey. You can even continue to think about it if you leave it in the kitchen and go out for dinner.
5. **It is flexible, and a function of the environment.** Driving to school along your usual route, you see a traffic jam ahead, so you turn the corner in order to go around it. Driving down a quiet street, a ball bounces in front of the car. While stepping on the brakes, you glance quickly to the sidewalk in the direction the ball came from, looking for a child who might run after the ball. As these examples show, human cognition isn't just a matter of following a fixed plan, or of always thinking ahead, it's also a matter of thinking in step with the world.
6. **It requires learning from the environment and experience.** We're not born knowing how to tell a joke, solve equations, play baseball, or cook dinner. Yet, most of us become proficient (and some of us expert) at one or more of these activities and thousands of others. Indeed, perhaps the most remarkable thing about people is how many things they learn to do given how little they seem to be born knowing how to do.

There are other properties that underlie our cognitive capabilities (for example, the quality of self-awareness), and there are other ways to interpret the same behaviors we have assigned to the categories above. What does it mean for Soar as an architecture to reflect this particular view of what is common to cognition? It means that the mechanisms and structures we put into Soar will make this view easy to implement, and other views more difficult. After we have constructed our model within this architecture, it will be easy to describe the model's behavior as goal-oriented because the architecture supports that view directly. Any theory establishes a way of looking at a problem. If the theory is useful, it allows us to model the behaviors we want to model in a way that seems easy and natural.

Having identified some common properties of cognitive behavior that Soar must support, we should be able to motivate the specific structures and mechanisms of the architecture by tying them back to these properties. Keep in mind our equation:

BEHAVIOR = ARCHITECTURE + CONTENT

It's clear that if we want to see how the architecture contributes to behavior then we need to explore the architecture in terms of some particular content. For example, describing how Soar supports the underlying characteristic of being goal-oriented won't by itself produce behavior; we have to be goal-oriented *about* something. Let's consider a simple scenario from baseball:[2]

> Joe Rookie is a hypothetical pitcher with the Pittsburgh Pirates, about to throw the first pitch of his major league career. He chooses to throw a curve ball. The batter, Sam Pro, hits the ball, but Joe is able to catch it after it bounces once between home plate and the pitching mound. Then, he quickly turns and throws the batter out at first base.

Figure 2 shows the participants in our scenario in their usual locations and roles, and the path of the ball from Joe's pitch to the out at first. Typical rhetoric about baseball players aside, it is clear that Joe displays many of the characteristics of intelligent behavior in our simple scenario. In particular, he:

1. **Behaves in a goal-oriented manner.** Joe's overriding goal is to win the game. In service of that goal, Joe adopts a number of *subgoals* — for example, getting the batter out, striking the batter out with a curve ball, and when that fails, throwing the batter out at first.
2. **Operates in a rich, complex, detailed environment.** As Figure 2 shows, there are many relevant aspects of Joe's environment he must remain aware of throughout the scenario: the positions and movement of the batter and the other members of his team, the number of balls and strikes, the sound of the bat striking the ball, the angle his body makes with the first baseman as he turns to throw, etc.
3. **Uses a large amount of knowledge.** In deciding on his pitch, Joe probably draws on a wealth of statistics about his own team, his own pitching record, and Sam Pro's batting record. We discuss Joe's knowledge in more detail below.
4. **Behaves flexibly as a function of the environment.** In choosing his pitch, Joe responds to his own perceptions of the environment: Is it windy? Is the batter left- or right-handed? etc. Although not included in our scenario, he may also have to consider the catcher's suggestions. When the ball is hit, Joe must show flexibility again, changing his subgoal to respond to the new situation.
5. **Uses symbols and abstractions.** Since Joe has never played this particular game (or even in this league) before, he can draw on his previous experience only by abstracting away from this day and place.

[2] For those unfamiliar with baseball, it is a contest between opposing sides who try to score the most runs within the fixed time frame of nine innings. To score a run a batter stands at home plate and tries to hit a ball thrown by the opposing team's pitcher in such a way that he and his teammates will advance sequentially around the three bases and back to home plate (see Figure 2). There are a number of ways the team in the field may halt the batter's progress, resulting in an out. An inning is over when each side has been both at bat and in the field for the duration of three outs. For a complete description of the game, see the Commissioner of Baseball, 1998.

6. **Learns from the environment and experience.** Learning is the acquisition of knowledge that can change your future behavior. If he's going to stay in the major leagues, Joe had better learn from this experience and next time throw Sam a fast ball.

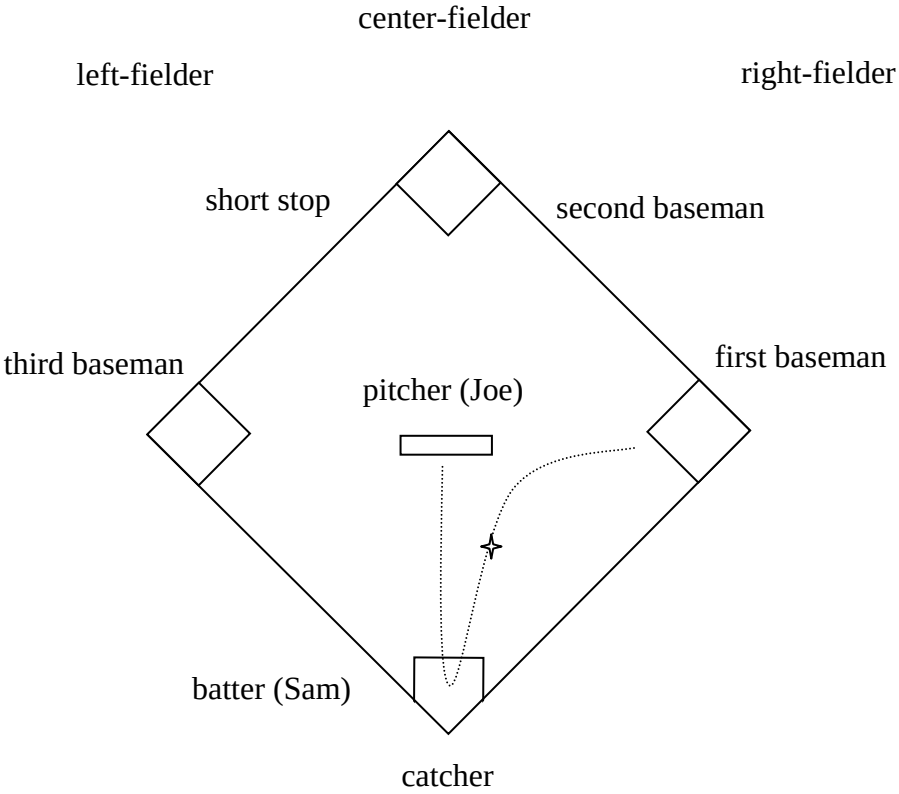

Figure 2: Joe's curve ball is hit by the batter, caught by Joe on a single hop and thrown to first base.

Just as the architecture is a theory about what is common to cognition, the content in any particular model is a theory about the knowledge the agent has that contributes to the behavior. For our model of Joe to act like a rookie pitcher, we will have to give it many different kinds of knowledge, some concrete examples of which are shown in Table 1.

Before our model can throw its first pitch we must find some way to represent and process Joe's knowledge in Soar. Our approach is to assume that there is an underlying structure to behavior and knowledge that is more than just arbitrary computation. In Soar this structure provides a means for organizing knowledge as a sequence of decisions through a *problem space.*

4. BEHAVIOR AS MOVEMENT THROUGH PROBLEM SPACES

Standing on the mound in the hot sun, Joe has a difficult goal to achieve and many ways to achieve it. To start, he must throw one of the many pitches in his repertoire. Once thrown, any of these pitches might result in a strike, a ball, a single, double, triple, or a homerun, and so on. Under some of these conditions Joe will have to pitch again to Sam Pro, under others he will face a different batter. If he faces a different batter, Sam may be on base or not; each of the variations in outcome is relevant to which pitch Joe chooses next. We can graphically represent the space

of possible actions for Joe, as he pursues his goal of getting Sam Pro out, as in Figure 3. The text in the picture describes the situations that Joe might be in at different moments in time. The arrows represent both mental and physical actions Joe might take to change the situation, like deciding on a pitch, throwing a pitch, and perceiving the result.

K1: Knowledge of the objects in the game
e.g. baseball, infield, base line, inning, out, ball/strike count
K2: Knowledge of abstract events and particular episodes
e.g. how batters hit, how this guy batted last time he was up
K3: Knowledge of the rules of the game
e.g. number of outs, balk, infield fly
K4: Knowledge of objectives
e.g. get the batter out, throw strikes
K5: Knowledge of actions or methods for attaining objectives
e.g. use a curve ball, throw to first, walk batter
K6: Knowledge of when to choose actions or methods
e.g. if behind in the count, throw a fast ball
K7: Knowledge of the component physical actions
e.g. how to throw a curve ball, catch, run

Table 1: A small portion of the knowledge needed to model Joe Rookie.

If we try to imagine (and draw) *all* the choices Joe might have to make during a game, given *all* the circumstances under which they might arise, we are quickly overwhelmed. Of course, even though Joe is able to make a choice in any of the circumstances described, he makes only one choice at a time while playing the actual game. In terms of Figure 3, Joe will actually decide on and throw only one first pitch of the game, it will either be hit or not, and so on. Joe must make his decisions with respect to the situation at the moment — based on that situation, what is recalled about the past, and what can be anticipated about the future — but does so over all the moments of his life. What this means for our model of Joe is that it, too, must be able to act based on the choices that make sense at the moment, but it must also be able to act under all the situations that may arise.

Figure 3 is an example of viewing behavior as movement through a *problem space* (Nilsson, 1971; Newell & Simon, 1972). The idea of problem spaces dates back to the earliest days of artificial intelligence research and cognitive modeling using computers. For the purpose of building a model, the representation in Figure 3 is particularly useful because it supports two points of view: a static view of Joe's life, which we can use to talk about all the possible actions he might take in a particular situation, and a dynamic view of Joe's life, which we can use to talk about the actual path his behavior moves him along.

The abstract form of a problem space is depicted graphically in Figure 4. The problem space itself is represented by a triangle to symbolize the ever-expanding set of possibilities that could unfold over time. Some of these possibilities will be represented explicitly, as explained below, while others are represented only implicitly, by ellipsis (...).

The situations described in text in Figure 3 are represented in the general case by *states* (squares in Figure 4). A state is described in terms of a vocabulary of features (in bold face) and their possible values (in italics). (The value of a feature may itself be a set of features and values,

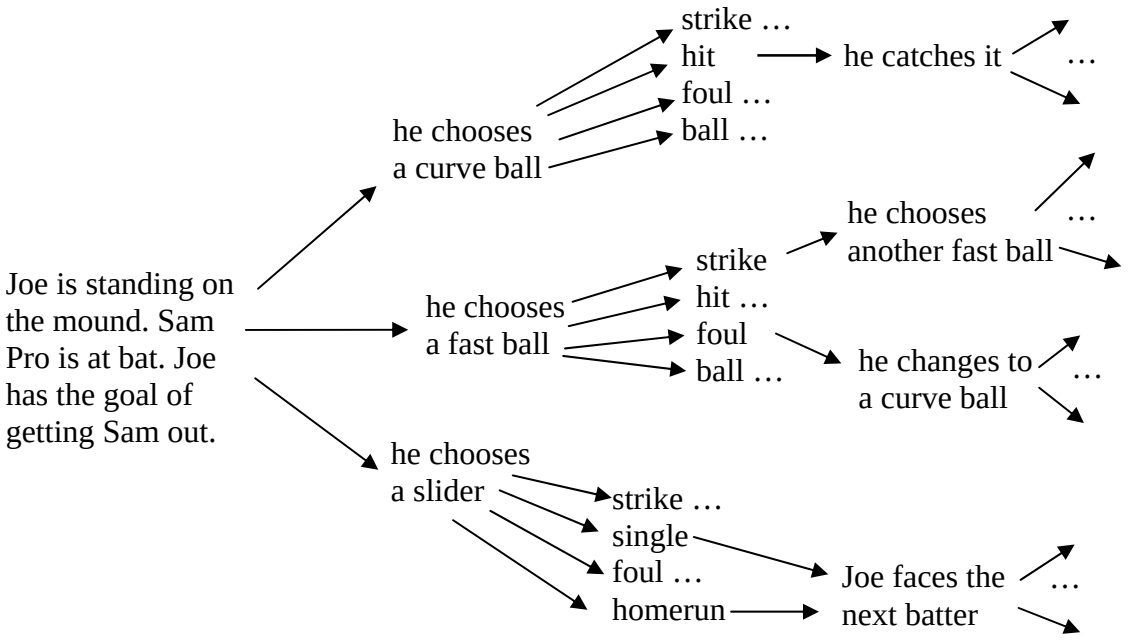

Figure 3: Looking at Joe's behavior as a space of decisions made over time. From the initial situation of holding the ball, Joe must decide what pitch to throw. On the basis of the outcome, he will make his next choice, and so on.

making possible the representation of a situation as richly interconnected sets of objects, as we shall see below.) So we might represent the part of the situation in Figure 3 described by the text "Sam Pro is at bat," by a pair of features and values (e.g. **batter name** *Sam Pro* and **batter status** *not out*), and these features and values would be part of any state intended to represent that part of the situation. In our examples so far we have been using features, values, and states to represent things in the observable situation. But there are always internal, unobservable aspects of any situation as well (e.g. Joe's perceptions, or things Joe may need to reason about that have no physical correlate, like batting average). The state is a representation of *all* the aspects of the situation — internal and external — that the model may need to choose its next action. Although one might think of goals and problem spaces as external to the state, since the state represents all aspects of the situation, it includes any available descriptions of the current goals and problem space.

To model a particular behavior, there must be an initial description of the situation, or *initial state* (S_0). This is the situation that that the model begins in – often defined by input from the environment. There must also be a description of the desired *goal state* or states (shaded squares). In Figure 4, the goal states are exactly those in which feature **f2** has value *v6*, regardless of the other features and their values. If **f2** is **batter status** and *v6* is *out*, then the goal states in Figure 4 would correspond to our informal notion of Joe having achieved his goal of getting Sam Pro out.

Both internal behavior (conscious or unconscious mental actions) and external behavior (actions observable in the world) correspond to moving along a path that leads from the initial state to a goal state via *operators* (arrows). At every point in time, there is a single state designated as the

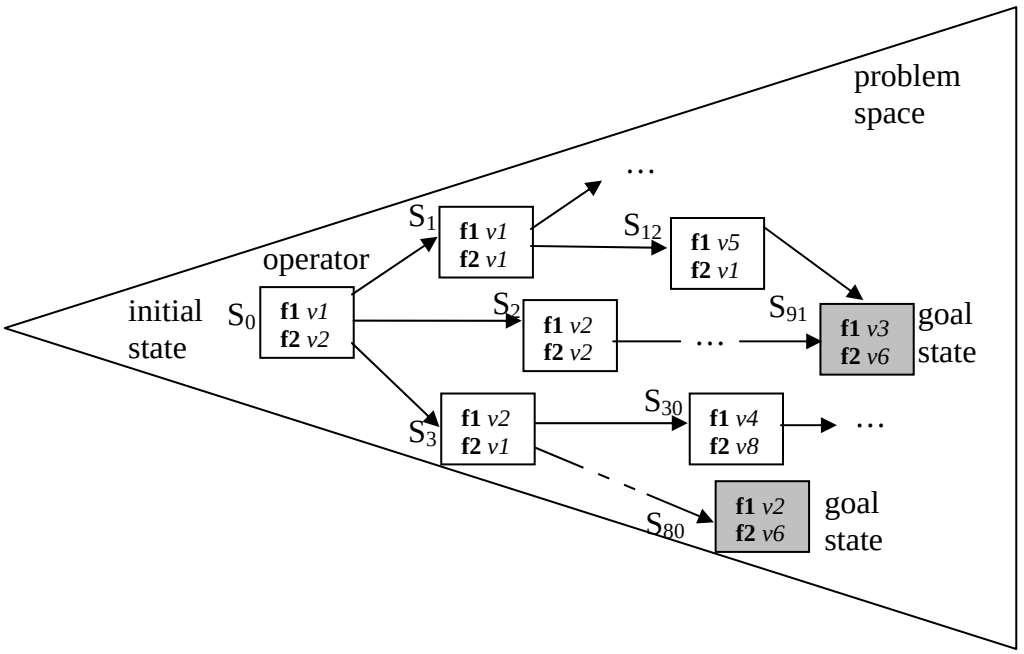

Figure 4: Behavior through time represented as movement through a problem space. The problem space is represented by a triangle to symbolize the ever-expanding set of possibilities that could unfold over time. Squares represent states, the features and values that reflect the internal and external situation. Goal states, states in which features have values that indicate the goal has been achieved, are shaded. Arrows represent operators that change or transform states. A state transformation may correspond to internal behavior (conscious or unconscious) or external behavior (actions observable in the world), or both. The states, features (bold face), and values (italics) are identified with arbitrary symbols, e.g. S1, **f1** *v2*, for convenience.

current state because it represents the current situation. Movement from the current state to a new state occurs through the application of an operator to the current state; an operator transforms the current state by changing some of its features and values. The application of an operator may cause only one feature value to change (e.g. S_0 to S_1) or may change multiple features and values (e.g. S_3 to S_{30}). The changes may be indirect if an operator involves an action in the outside world that leads to changes in perception. There can also be changes in the state that result from the actions of others, such as when the batter swings at a pitch. One implication is that although multiple states are shown in Figure 4, only one state exists at any time and prior states are not directly accessible.

To solve a problem, operators must be applied to move through the problem space. Of course, movement through a problem space could be entirely random. To keep behavior goal-directed, the succession of operators that are applied to the state and the resulting state transformations must be guided by *the principle of rationality*: "if an agent has knowledge that an operator application will lead to one of its goals then the agent will select that operator" (Newell, 1982). This may seem like a simple idea, but we will see that it pervades every aspect of the Soar architecture from operator selection to reinforcement learning.

We are now in a position to answer the question posed at the end of the previous section. How can we express the different kinds of knowledge the model must have so that it acts in a goal oriented way? The answer is: by representing the knowledge in terms of goals, states, and operators and guiding the choice of which operator to apply by the principle of rationality. Rewriting our scenario in these terms, we say that Joe has the goal of getting the batter out in the state defined by the start of the game. Of all the operators available for each state, he should choose the one that he thinks will transform the current state to a new state closer to the desired goal state.

Describing the underlying idea of problem spaces puts us on the path to our own goal of modeling Joe Rookie in Soar. Mapping knowledge into goals, states and operators is the first step toward tying the content of Joe's world to the Soar architecture. Not surprisingly, this first step raises a new set of questions that we, as model builders, must answer: What knowledge becomes part of the state and what knowledge becomes part of the operators? How do we know what an operator application will do? How do we know when the goal has been achieved? To answer these questions, the next section describes how Soar supports goals, problem spaces, states and operators.

5. TYING THE CONTENT TO THE ARCHITECTURE

Consider Table 1 again. The list incorporates many different kinds of knowledge that our model must include: knowledge about things in the world (K1 and K2) and knowledge about abstract ideas (K3 and K4), knowledge about physical actions (K7) and knowledge about mental actions (K5), even knowledge about how to use the other kinds of knowledge (K6). This content is expressed in the familiar, everyday terms of the *domain* of baseball. But the architecture of cognition cannot be designed for baseball alone. Because the architecture must support what is common across many domains, its mechanisms must process a domain-independent level of description. What is common across all domains and problems? In Soar, it is the decomposition of knowledge into goals, problem spaces, states, and operators that is common across all problems.

Figure 5 presents the state and operators for Joe Rookie graphically. At the moment, he is in a state (square) in which the batter, Sam Pro, is not out, and he has selected the throw-curve-ball operator (arrow) to apply to this state to produce a new state. It also shows that has the goal of getting the batter, and that this goal is a feature of the state. Remember that the state represents the current situation of the model, which includes the goal. Thus, a state must have some feature, such as **goal** whose value has features that describe the goal. In this way, the goal is available for examination during problem solving to evaluate progress and direct the selection of operators toward the goal. There must be additional knowledge that detects when the state satisfies the goal, which then notifies the architecture that the goal has been achieved (and as we shall see later, contributes to reinforcement learning).

If we look at Table 1 and Figure 5 together, some simple guidelines for tying domain content to the architecture emerge. The domain knowledge of the objects and people in the game (K1) is represented by the features and values in the state. We use knowledge of actions (K5 and K7) to define our operators and knowledge about objectives (K4) to determine goals.

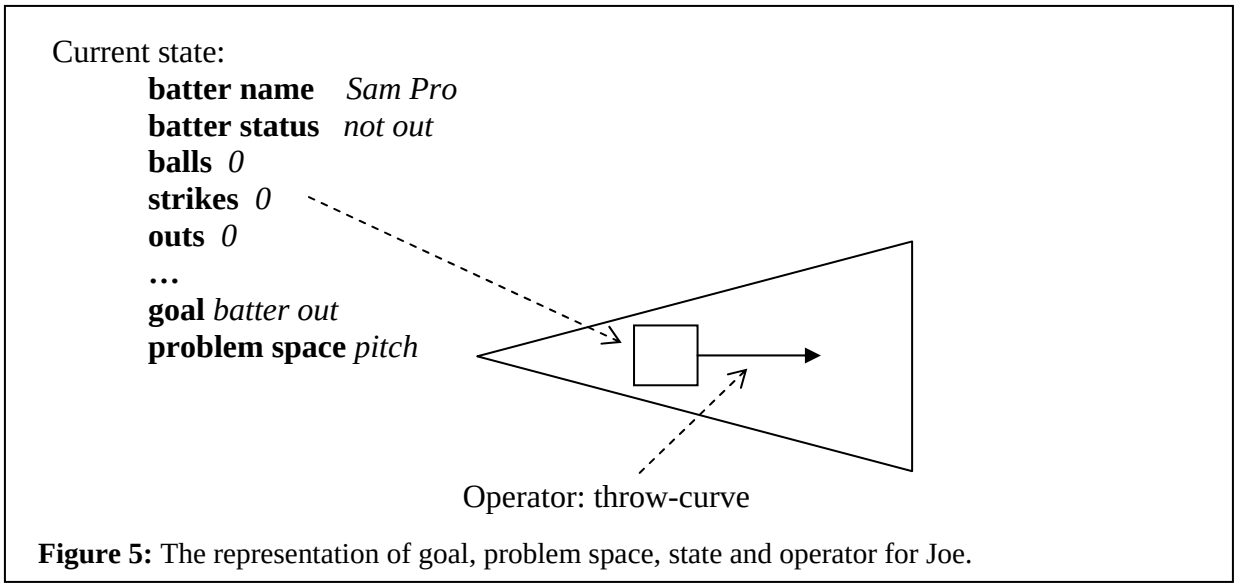

Figure 5: The representation of goal, problem space, state and operator for Joe.

The problem space is also represented as a feature of the state (**problem-space** *pitch*). For simple examples such as this, having an explicit representation of the problem space on the state can be useful, but our experience is that in more complex problems, a problem space is defined by a combination of the knowledge of the available objects that can be manipulated (K1), the rules of the game (K3), the set of available actions (K5 and K7) and the set of objectives (K4). By picking different representations of the situations, deciding to abide by different "rules" of a problem, or by selecting different sets of operators to consider, different problem spaces emerge.

We can now ask the question of "How do operators get selected?" Each operator uses content knowledge to determine when it is relevant to the current goal and the current state. Given that the goal is to get the batter out and the situation is that of being the pitcher in a baseball game, the operators to be considered will be the various kinds of pitches Joe can throw. The model does not consider, for example, operators for fielding or batting or doing laundry; it behaves in a goal-oriented way. Operators that share common tests for goals and situations can be considered to be part of the same problem space, in this case a problem space named "*pitch*." More specific knowledge can also be used to further distinguish between operators. In Table 1, this sort of knowledge (K6) enables the model to choose a curve ball based on particular attributes of the state, such as whether Sam Pro is a left- or right-handed batter. (In Section 7, we will consider what happens when the knowledge available to the model is inadequate to choose a single operator.)

Assume that the model of Joe has chosen to throw a curve ball; how do we know what that (or any other) operator will do? There are really two answers to this question, each suggested by a different source of knowledge. On the one hand, what the operator will do may be defined completely by the execution of the operator in the external world. In our case, imagine our model of Joe as a baseball-playing robot. If knowledge of physical actions is used (K7), the application of the throw-curve-ball operator can result in appropriate motor actions by the robot. Once the motor actions have been performed (and the results perceived), both the world and the model will be in new states. On the other hand, what the operator will do may be defined by knowledge

of abstract events or particular episodes (K2). Imagine that Figure 5 is a snapshot of the model's processing while it is evaluating what to do next. In this case, the application of the throw-curve-ball operator should not result in an action in the world. Instead, the next state should depend on the model's knowledge about curve balls in general, or about what happens when Sam Pro is thrown curve balls. Clearly, the outcome of applying the operator as a mental exercise is a state that may or may not resemble the state that results from throwing the curve ball in the world. What is important is not whether the operator application preserves some kind of truth relation between the external world and its internal simulation. What is important is that using states and operators allow us to model either behavior — acting or thinking about acting — as a function of what kind of knowledge is brought to bear.

Imagine now that our model of Joe has thrown the ball, either in the world or in its head. How do we know if the resulting state has achieved the goal of getting the batter out? In the domain of baseball, determining that the current state is a desired state relies on knowledge of the rules of the game (K3). It is also possible for the environment to give signals of success and failure that the model can interpret. In baseball, a novice player might not know all the rules, but would know to listen to the umpire ("You're out!"), to determine if a goal has been achieved.

By augmenting the state with goals and problem spaces, we also provide a straightforward answer to how to the current goal and problem space change over time. They change through the application of operators – not operators that are part of the original task (pitching), but operators nonetheless. This might seem a bit strange in that operators are traditionally thought of as being used to move through problem spaces to achieve goals, but it also opens up the possibility of deliberately modifying the description of the goal or problem space as the state itself changes from external (or even internal) influences. For example, if Joe's team is ahead in the fifth inning, but rain is on the horizon, Joe might specialize the goal to not just get the batters out, but to get them out very quickly so that the inning is over before the rain comes. If the fifth inning is completed when the game is called because of rain, the game is over and does not have to be replayed. Similarly, if Joe is pitching when there is a member of the opposing team on base, he might change the problem space – use a more restricted problem space – such that he considers only pitches where he has a short windup so that it is more difficult for the opponent to steal a base. This doesn't answer the question of where subgoals are created – but we will address that question later in Section 7.

In this section we have seen how domain content can be tied to operators, states, problem spaces, and goals. The content-independent processing of states, operators, and goals is key to their usefulness. If the architecture is going to support what is common across domains, its behavior can't be limited by the structure or content of any one domain. Although we have demonstrated how different kinds of knowledge map onto goals, problem spaces, states, and operators, we have left quite a few important details unspecified. In particular: How should the knowledge in its general form be represented? How should states and operators be represented? What are the mechanisms for perceiving and acting on the external world? And most important: What are the architectural processes for using knowledge to create and change states and operators?

6. MEMORY, PERCEPTION, ACTION, AND COGNITION

To make sense of our everyday experience and function in the world, we need to use our knowledge about objects and actions in pursuit of our current goals. You know many things about a cup but you use only some of that knowledge when you want a cup of coffee, and some other part of it when you want a place to put pens and pencils. Moreover, once you have applied some of that knowledge, a particular cup will be full of coffee or full of pens. Your ability to reason about other cups as pencil holders or coffee mugs will not have changed but there will be something in your current situation that will have taken on a single role — your general knowledge will have been tied to the particular instance of the cup in front of you. This dichotomy between general knowledge and specific applications of that knowledge is captured in Soar by the existence of two different types of memory. Figure 6 shows the basic structure of Soar's memories.

Knowledge that exists independent of the current situation is held in the architecture's *long-term memory* (LTM). Soar distinguishes three different types of LTM: procedural, semantic, and episodic. Procedural knowledge is about how and when to do things – how to ride a bike, how to solve an algebra problem, or how to read a recipe and use it to bake a cake. Semantic knowledge consists of facts about the world – things you believe to be true in general – things you "know," such as bicycles have two wheels, a baseball game has nine innings, and an inning has three outs. Episodic knowledge consists of things you "remember" – specific situations you've experienced, such as the time you fell off your bicycle and scraped you elbow. LTM is not directly available, but must be "searched" to find what is relevant to the current situation.

The knowledge that is most relevant to the current situation is held in Soar's *working memory* (WM). Much of this is knowledge that is specific to the current situation – what is true at this instant of time. WM can also contain general knowledge that is relevant to the current situation – general facts or memories of specific previous situations that are useful in making decisions about the current situation. In Soar, WM is represented as a set of the features and values that make up the current state (and substates), which might include representations of the current goal, problem space, and operator.

To say this a little more intuitively, it is useful to think about LTM as containing what can be relevant to many different situations but must be explicitly retrieved, and WM as containing what the model thinks is relevant to the particular situation it is currently in. One of the key distinctions between WM and LTM is that knowledge in working memory can be used to retrieve other knowledge from LTM, whereas LTM must first be retrieved into WM. Knowledge moves from LTM to WM by both automatic and deliberate retrieval of relevant LTM structures.

Although all types of long-term knowledge (procedural, semantic, and episodic) are useful, procedural knowledge is primarily responsible for controlling behavior and maps directly onto operator knowledge. Semantic and episodic knowledge usually come into play only when procedural knowledge is in some way incomplete or inadequate for the current situation, and thus we will treat them in more detail in Section 7. Table 2 shows some examples of long-term procedural knowledge that might be available for a model of Joe Rookie. The exact format of the knowledge as it is given to Soar is unimportant; for expository purposes, we write the knowledge as English-like if-then rules. The rules represent associations between a set of conditions, expressed in terms of features and values and specified by the "if" part of the rule, and a set of

actions, also described by features and values, in the "then" part. As in Figure 5, when describing information that is part of the state, we use boldface for features and italics for values.

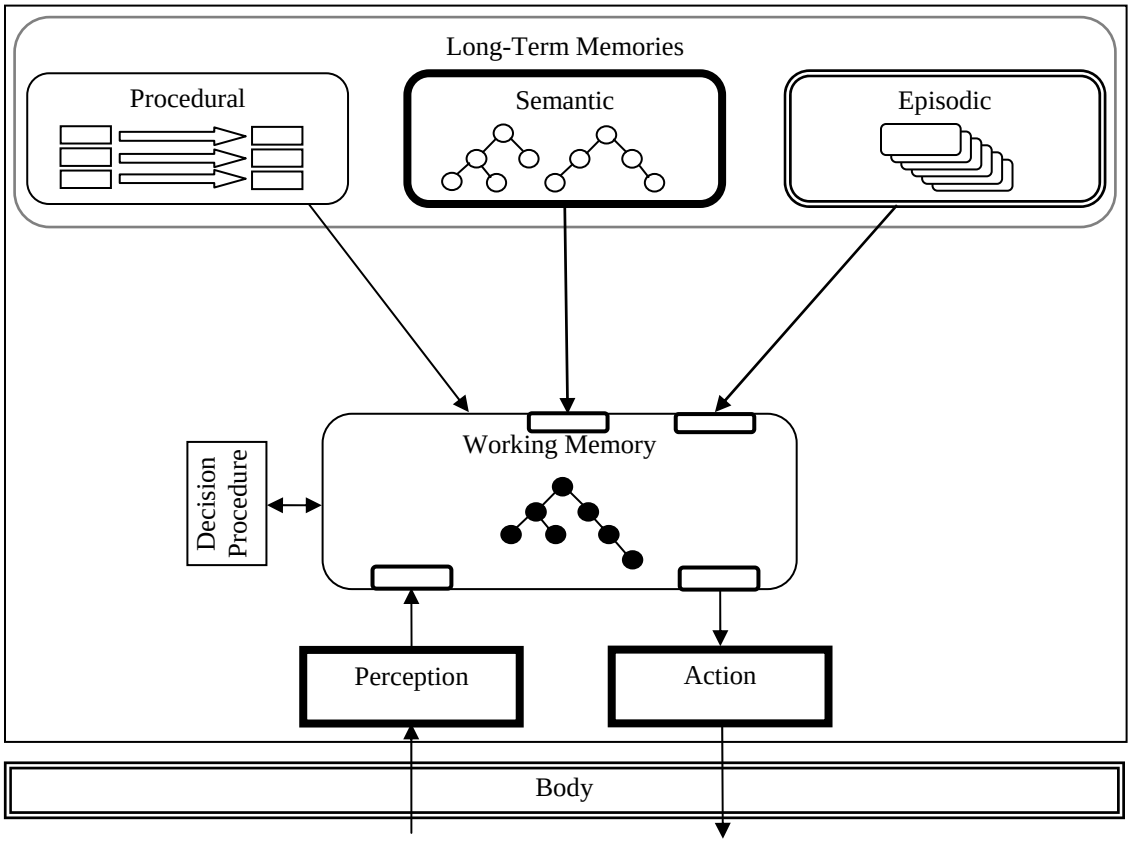

Figure 6: Memory structures in Soar.

We can use Figures 5 and 6 and Table 2 together to make some preliminary observations about the relationship between LTM and WM. First, the "if" portion of each rule tests either a perception or elements of the state (and its substructure of features and values which can include the goal and problem space), or operator. If there is a match between the "if" part of a rule and elements in WM, we say that the rule has been *triggered*. This, in turn, causes the "then" portion to *fire* by either sending a message to the motor system (r6) or suggesting changes to the goal, state, and operator. Thus, each matching rule maps from current goal, state, and operator to changes to those objects.[3]

Second, note that there can be dependencies between the rules. For example, r2 will not match the current goal until r1 has matched and fired, and r3 will not match until r1 and r2 have matched and fired. These dependencies are part of the semantics of the domain of baseball — you don't choose a pitch until you've decided to pitch to the batter and you don't pitch to a batter

[3] We are using the word *map* in its mathematical sense, where it refers to a type of function. A simple function, such as Y = f(X), specifies a single value of Y for each value of X. In contrast, a mapping function can return many values. In this sense, the LTM rules in Soar are mapping functions because, given a particular set of WM features and values, a matching LTM rule can return many new features and values to WM.

if you aren't the pitcher on the mound. Soar doesn't recognize the existence of these dependencies: it simply matches the "if" portions and performs the "then" portions. Because all the elements of the rules are expressed in terms of perceptions, actions, states, and operators, they are processed by the architecture in a completely general, domain-independent way. Put slightly differently: if you can express a domain's content in these terms, the architecture can use that content to produce behavior.

(r1) If I am the pitcher, the other team is at bat, and I perceive that I am at the mound
then suggest a **goal** to get the batter out via pitching (*Pitch*).
(r2) If the **problem space** is to *Pitch* and I perceive a new batter who is left/right handed
then add **batter** *not out*, **balls** *0*, **strikes** *0*, and **batter** *left/right-handed* to the state.
(r3) If the **problem space** is to *Pitch* and the **batter** is *not out*
then suggest the throw-curve-ball operator.
(r4) If the **problem space** is to *Pitch* and the **batter** is *not out* and the **batter** is *left-handed*
then suggest the throw-fast-ball operator.
(r5) If both throw-fast-ball and throw-curve-ball are suggested
then consider throw-curve-ball to be better than throw-fast-ball.
(r6) If the throw-curve-ball operator has been selected
then send throw-curve to the motor system and add **pitch** *thrown* to the state.
(r7) If the **pitch** was *thrown* and I perceive that it was called a ball,
then increment the **balls** count.
(r8) If the **pitch** was *thrown* and I perceive that it was called a strike,
then increment the **strikes** count.
(r9) If the **pitch** was *thrown* and I perceive a hit
then add **pitch** *hit* to the state

Table 2: Some of the structures in the model's long-term memory.

Third, notice that for tasks where a model has sufficient procedural knowledge, it does not require access to semantic or episodic knowledge – it just knows what to do. Thus, models for highly skilled behavior will invariably be heavily weighted to procedural knowledge. Semantic and episodic knowledge will play a bigger role for non-expert behavior, when procedural knowledge is incomplete (Section 7).

Unlike the cognitive processes it represents, a model often begins and ends its processing at arbitrary points in time. A human Joe Rookie does not suddenly find himself on the pitching mound with no context to explain how he got there, but our model of Joe begins with no elements in working memory. Working memory elements in Soar arise in one of two ways: through perception or through retrievals from long-term memory during the decision cycle. The goal of getting the batter out, for example, is established in working memory when rule r1 "fires" because the Soar model of Joe perceives that it is the pitcher and it is on the pitching mound. Modeling perception is a complex topic which we will, for the most part, not address. What is important for our purposes here is that the architecture provides a straightforward facility (the Perception module) for introducing elements into working memory from the external environment via perception, whether that perceptual input comes from a simple simulation program, the output from a video camera, or feedback from a robot arm.

The decision cycle is the processing component that generates behavior out of the content that resides in the long-term and working memories. The purpose of the decision cycle is to select the next operator to apply. Remember the problem space point of view discussed in Section 4: goal-directed behavior corresponds to movement in a problem space from the current state to a new state through the application of an operator to the current state. Concretely, then, the purpose of the decision cycle is to select a new operator. To understand how the decision cycle works, let's focus on how the model chooses and throws the curve ball. Figure 7(a) shows the state of working memory after the decision cycles in which rule r1 has fired, establishing the get-batter-out goal, the *pitch* problem space, and the initial state which contains the substructure for the count (abbreviated as 0/0 for **balls** *0* and **strikes** *0*). The ellipsis in the state represents other working memory elements resulting from perception.

A decision cycle is a fixed processing mechanism in the Soar architecture that does its work in five phases: *input, elaboration, decision, application,* and *output*. During input, working memory elements are created that reflect changes in perception. During elaboration, the contents of working memory are matched against the "if" parts of the rules in long-term memory. All rules that match from procedural memory, fire in parallel, resulting in changes to the features and values of the state in addition to suggestions, or *preferences*, for selecting the current operator. As a result of the working memory changes, more rules may fire. Elaboration continues in parallel waves of rule firings until no more rules fire. Figure 7(b) shows the results of the elaboration phase for our example. First, the appearance of a new batter at home plate causes rule r2 to fire, augmenting the state with information about the batter's presence and handedness (*bnl* in the figure, which is shorthand for **batter** *not out* and **batter** *left-handed*). If we assume Sam Pro is left-handed, then in the next wave of elaborations, rules r3 and r4 fire in parallel, suggesting both a curve ball and a fast ball as pitches (dotted arrows). These suggestions cause r5 to fire in the next elaboration, augmenting working memory with a preference for the curve ball (represented by the greater than sign over the dotted arrows). Because no more of the rules in LTM fire, the elaboration phase is complete. The changes to working memory during elaboration are, as the name suggests, just elaborations. They are simple conclusions or suggestions (in the case of preferences) and not actions in the world or changes to internal structures that should persist beyond the current situation. *Non-monotonic* changes (changes in the current state as opposed to elaborations of the existing state), must be made carefully, as they determine the path of the model through its problem space, and thus must be performed by operators.

At the end of the elaboration phase, two operators, throw-curve-ball and throw-fast-ball have been suggested as candidates for the operator. Why are changes to the operator phrased as suggestions or preferences? To make sure all the knowledge available in the current situation is brought to bear before a decision is made. That is, a decision to change the operator should not be made until all the rules in LTM that match the current situation have been retrieved. Remember the principle of rationality: the model should use its knowledge to achieve its goals. The introduction of preferences allows the architecture to collect all the available evidence for potential changes to the state before actually producing any change. Thus, quiescence in elaboration, i.e., the absence of any further firings of LTM rules, signals the start of the decision phase. This is when the preferences added to working memory are evaluated by a fixed architectural decision procedure. The decision procedure is applied to the vocabulary of preferences, independent of the semantics of the domain. The vocabulary includes symbolic preferences, such as one operator is *better* than another, but it also includes numeric preferences,

such as the expected value of an operator is *X*. In other words, the decision procedure isn't written in terms of throw-curve-ball and throw-fast-ball; it's written in terms of symbolic and numeric preferences among operators. In our example, there are suggestions for two operators and a "better" preference, suggested by rule r5 that orders their desirability. So, as shown in Figure 7(c) the outcome of the decision procedure is to choose the suggested operator that is better: throw-curve-ball.

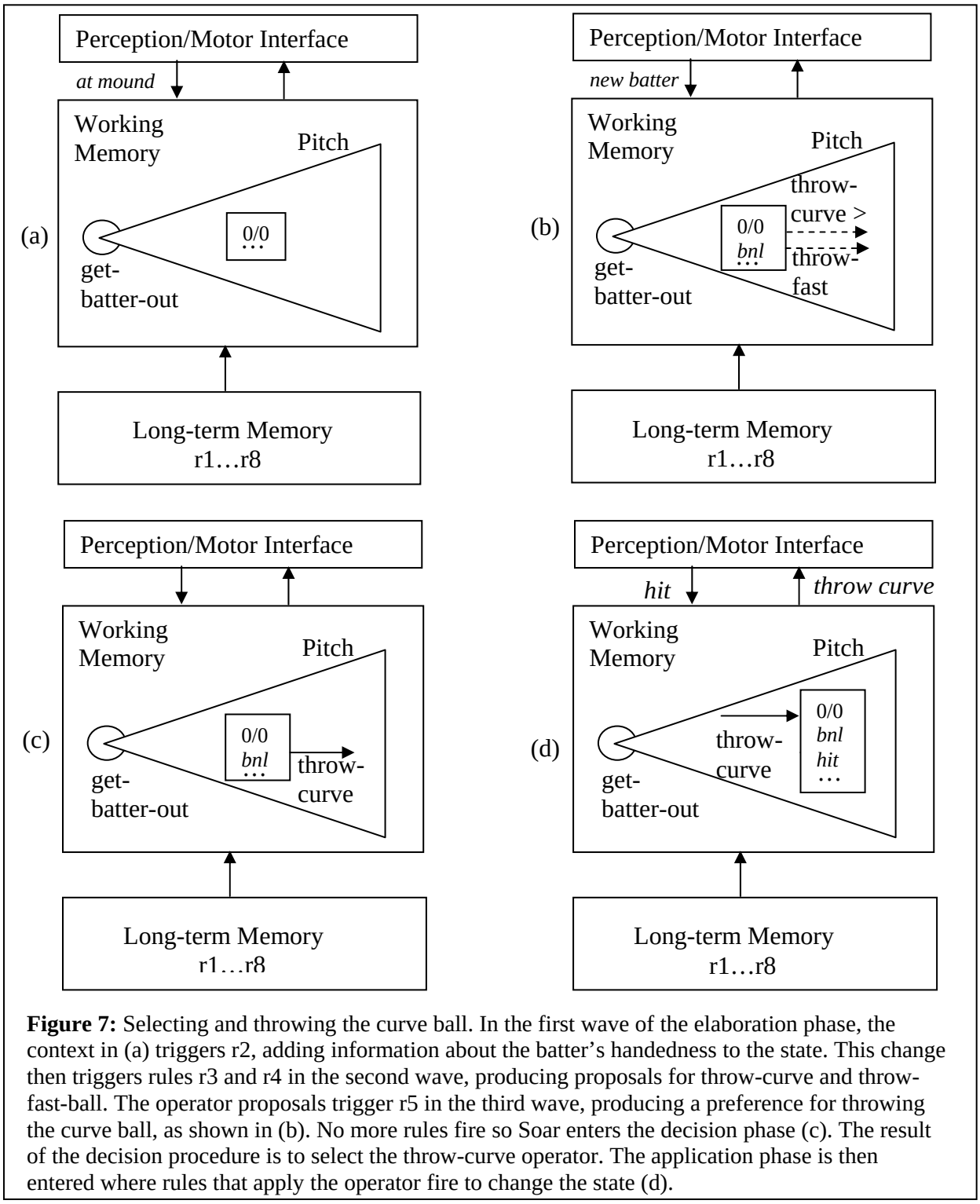

Figure 7: Selecting and throwing the curve ball. In the first wave of the elaboration phase, the context in (a) triggers r2, adding information about the batter's handedness to the state. This change then triggers rules r3 and r4 in the second wave, producing proposals for throw-curve and throw-fast-ball. The operator proposals trigger r5 in the third wave, producing a preference for throwing the curve ball, as shown in (b). No more rules fire so Soar enters the decision phase (c). The result of the decision procedure is to select the throw-curve operator. The application phase is then entered where rules that apply the operator fire to change the state (d).

Of course, selection of the operator isn't enough. The operator must be applied to produce the state transition that completes a move in the problem space. The application of the operator occurs during the application phase, as shown in Figure 7(d). The throw-curve-ball operator in working memory causes rule r6 to fire, producing a motor command. In the final phase, the

motor command is sent to the Perception/Motor Interface to the external environment, leading to the pitch being thrown. In turn, Sam hits the curve ball, an event that is perceived by the model and encoded as an augmentation to the state via rule r9.

In Soar, only a single operator can be selected for a state at a given time. This is the essence of choice – by choosing one action you are committing to the changes that constitute that operator as opposed to some other set of actions. Usually, there is a resource that can be used only in limited ways and a decision must be made how to use it now, for the current situation. Baseball is played with only one ball and Joe can throw only one pitch to one batter in one game at a time. The limits of space and time often make it impossible to do more than one thing at a time. But what about limited parallelism, such as Joe moving many parts of his body in concert to throw a pitch? In Soar, multiple actions can be packaged together as a single operator that is then selected as a unit. The execution of the operator can then be carried out by parallel rule firing (and parallel motor systems). The real restriction is that a set of parallel activities must be represented as a single operator that can be selected and applied as a unit. Thus, Soar supports a limited form of parallelism where the parallel activities are prepackaged as an operator, but it does not support arbitrary parallelism.

Given the knowledge we have put into long-term memory, Figure 7(d) represents the end of the behavior we can elicit from the model. Yet the answers to the questions we asked at the end of Section 5, and the basic outline of further processing, should be clear:

- How should general knowledge be represented? As rules that map one set of working memory elements into a new set of working memory elements. General knowledge can also be represented in semantic memory, which will be described later.
- How should knowledge be represented within a state? As the simple features and values that are the building blocks of the rules in long-term memory. The representations of knowledge in long-term procedural memory and in the state are inextricably linked because the decision cycle is essentially a controlled process for matching elements in the context to the "if" patterns in procedural LTM.
- What are the architectural processes for using knowledge in LTM to create and change the representation of the current situation. The decision cycle with its input, elaboration, decision, application, and output phases. The phases are necessary for the architecture to produce behavior in accordance with the principle of rationality. A result, however, is that the representation of knowledge in procedural long-term memory must be cast in terms of preferences and suggestions. This must occur so that decisions are postponed until the moment when all knowledge has been elicited by the current situation and can be integrated to produce a change to that representation.
- What are the mechanisms for perceiving and acting on the external world? Perception and action go through an interface that is embedded in cognition's decision cycle. Because percepts add new information to working memory, however, perception must ultimately produce working memory elements in the same representation as long-term memory, i.e. from the same vocabulary of features and values. Similarly, the trigger for those actions will be working memory elements.

Considering our questions one at a time leads to answers that are somewhat circular because the basic structures and processes of the architecture must form a coherent whole. The decision cycle

can only serve as an adequate process for ensuring goal-oriented behavior if long-term memory has a certain form. Working memory can only contribute to behavior that is flexible as a function of the environment if the decision cycle is essentially a domain-independent process of retrieving knowledge from LTM into STM. Thus, each piece is motivated at least in part by the existence of the others. Together they serve as a simple, elegant theory of how the knowledge that Joe has can produce the behavior we see.

What we have shown in this section is how the model can be made to produce some of Joe's behavior by casting content in terms the architecture can manipulate. The knowledge that we described in everyday terms in Table 1 was redescribed as rules between contexts in Table 2. Once in this form, the main processing components of the architecture could be applied to the content, with the decision cycle mediating each move through the problem space. In the example, the knowledge in our domain content led directly to the selection of a single operator and then external action at the end of each decision cycle. But what happens if the decision cycle can't decide?

7. DETECTING A LACK OF KNOWLEDGE

Look again at Table 2, and consider what would happen if rule r5 were not part of our domain content. At the end of the elaboration cycle in which r2, r3, and r4 fire, two operators would have been suggested with no preferences to decide between them. The architecture requires that a single operator be selected, but without r5, the knowledge that could be elicited from long-term memory would not be adequate to meet that constraint. In short, processing would reach an *impasse* because of the tie between the two proposed pitch operators. Indeed, in Soar, an impasse is an architectural event that arises whenever the decision procedure cannot resolve the preferences in working memory to select a new operator. An impasse is what happens when the decision cycle can't decide.

Now it might seem that if r5 should be part of our content but isn't, then its absence simply represents an inadequacy of the theory or poor design of the model. But where did the knowledge implicit in that rule come from? Often the reasons for selecting one operator over another are based on deliberate reasoning that is not easy to express in one or two rules. Instead, the reasoning required to select an operator requires decisions and explorations, such as imagining what would happen if each operator were selected and then comparing the results, or attempting to recall a similar situation and using it to guide the selection. To support these types of processing, Soar automatically creates a substate in response to an impasse. A substate is a new state that represents the information relevant to resolving the impasse (including the original state). In the substate, Soar uses exactly the same approach as before – it selects and applies operators to achieve a goal, but in this case the goal is to select between two operators for the original state, and the operators will attempt to retrieve and compare knowledge about those tied operators rather than produce actions, such as pitches, in the external environment.

Figure 8 shows graphically the presence of the impasse (crescent) and the resulting substate. The substate has the goal resolve-tie as a feature. Let us assume that the way Joe would decide between these options is by trying to remember how successful he's been with each of the pitches in the past with this batter. Our model will accomplish this by using cues from the pre-

impasse environment to retrieve a relevant memory. Once such an event has been recalled, it can be used as the basis for evaluating each of the proposed pitches.

As a first step, we assume that Joe has the following fact in episodic memory.

> (r10) The **batter-name** is *Sam Pro,* the **weather** is *windy*, **isa** *homerun*, **event** *116*, **pitch** *throw-fast-ball*, etc.

Although this could potentially have been recalled before the impasse, it is not necessary unless the procedural knowledge is incomplete. Moreover, it seems unlikely that Joe is constantly being reminded about all the games in his past that share features with the current situation, and we don't want our model of Joe to be inundated with memories either. Instead, we want the model to be reminded when there is a reason for the reminding. In other words, the reminding is goal-driven; here, the goal that drives the reminding is the need to resolve the operator tie. In Soar, knowledge encoded in the semantic and episodic memories is usually used in substates when procedural knowledge is inadequate.

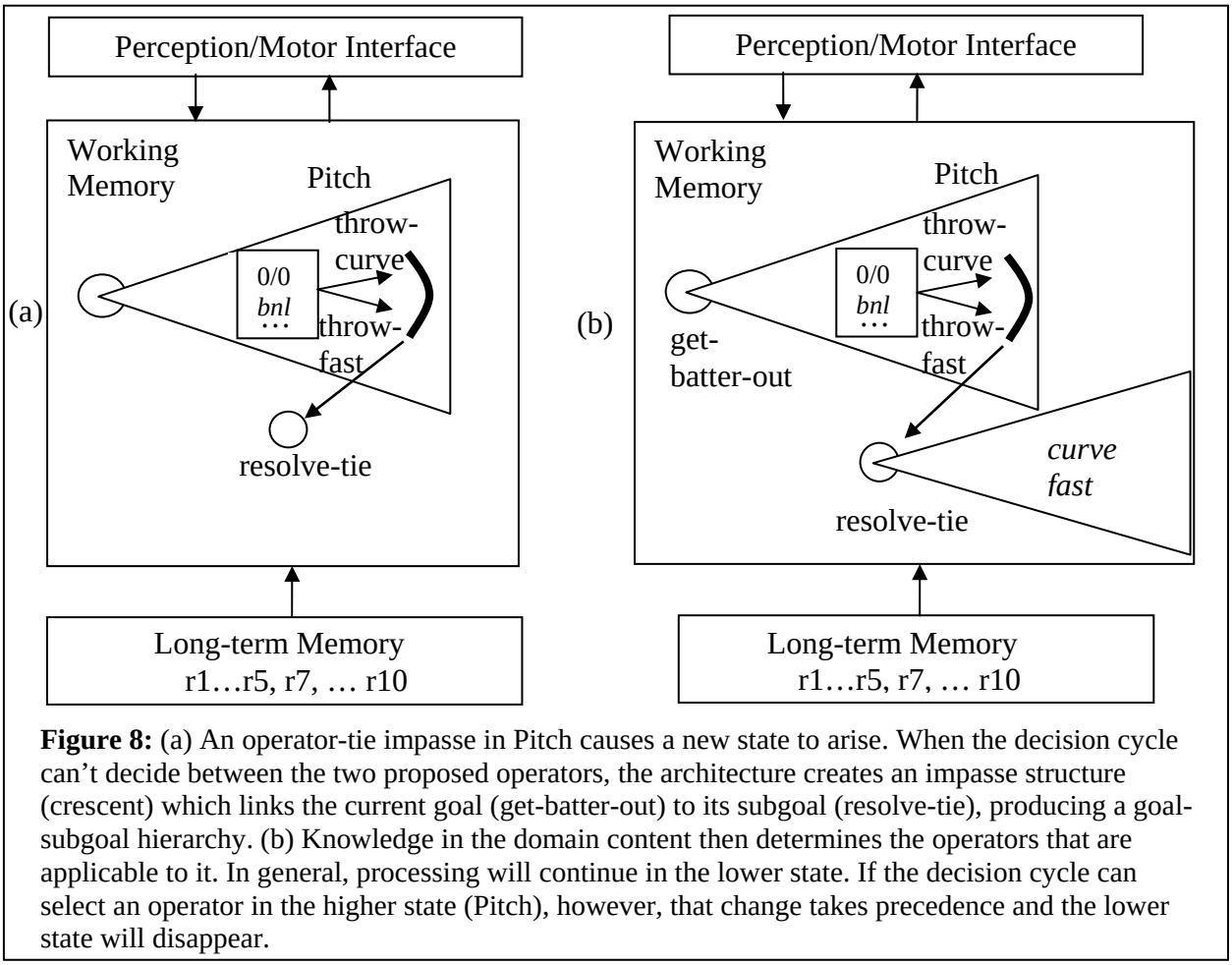

Figure 8: (a) An operator-tie impasse in Pitch causes a new state to arise. When the decision cycle can't decide between the two proposed operators, the architecture creates an impasse structure (crescent) which links the current goal (get-batter-out) to its subgoal (resolve-tie), producing a goal-subgoal hierarchy. (b) Knowledge in the domain content then determines the operators that are applicable to it. In general, processing will continue in the lower state. If the decision cycle can select an operator in the higher state (Pitch), however, that change takes precedence and the lower state will disappear.

How is this knowledge evoked? To control the retrieval of knowledge from episodic memory, a cue must be created that can be used to search the memory. The cue itself is created by an operator that selects structures from the superstate. Thus, we will require additional content knowledge, such as the following:

(r11) If there are two tied pitch operators then suggest a recall operator.
(r12) If applying a recall operator to a state, use the features in the superstate as a cue for retrieval to the episodic memory.

The recall operator creates a cue, and episodic memory delivers a match to the cue. The episode might not match the cue exactly, so for example, the **weather** might not be *windy*, but it will find the episode that is the closest match (which in some cases might not be predictive of the current situation), moving the model into the third state in Figure 9(b). In other words, the model has now been reminded of a previous game that has some elements in common with the current situation. In some cases, no likely match will be returned (especially for a rookie pitcher). The model can modify the cue in an attempt to trigger a memory, possibly by using semantic knowledge such as:

(r13) **Batter-name** *Sam Pro* has **nickname** *Crash*

This could lead to the retrieval of an episode that was created when Joe only knew the batter by his nickname so that the original episode was:

(r10') The **nickname** is *Crash,* the **weather** is *windy*, **isa** *homerun*, **event** *116*, **pitch** *throw-fast-ball*, etc.

Once an episodic memory has been evoked, working memory contains information that can be used to try to resolve the tie. The next three rules define the evaluate operator that creates preferences to resolve the tie. In the language of Table 1, this operator maps knowledge of particular episodes (K2) and knowledge of objectives (K4) into knowledge of when to choose actions or methods (K6):

(r14) If there are two tied operators and one of the operators has an **event** with a **pitch** then suggest an evaluate operator
(r15) Prefer evaluate operators over recall operators
(r16) If applying an evaluate operator to a state that has two tied operators and one of the operators has an **event** that **isa** *homerun* with a **pitch** then prefer the other operator

Rule r14 proposes the evaluate operator on the basis of a relevant event. Rule r15 is included to prefer an evaluation just in case other recall operators are still possible. Rule r16 adds a preference to working memory that results from applying the evaluate operator. In the current situation, r14-r16 simply mean that if Joe remembers a time when someone threw a fast ball that led to a homerun under similar circumstances, then he should prefer to throw the curve. As shown in the right-most state in Figure 9(b), the actual working memory changes produced by rule r16 include both a preference for the throw-curve-ball operator and the delineation of the current state as a goal state in this problem space.

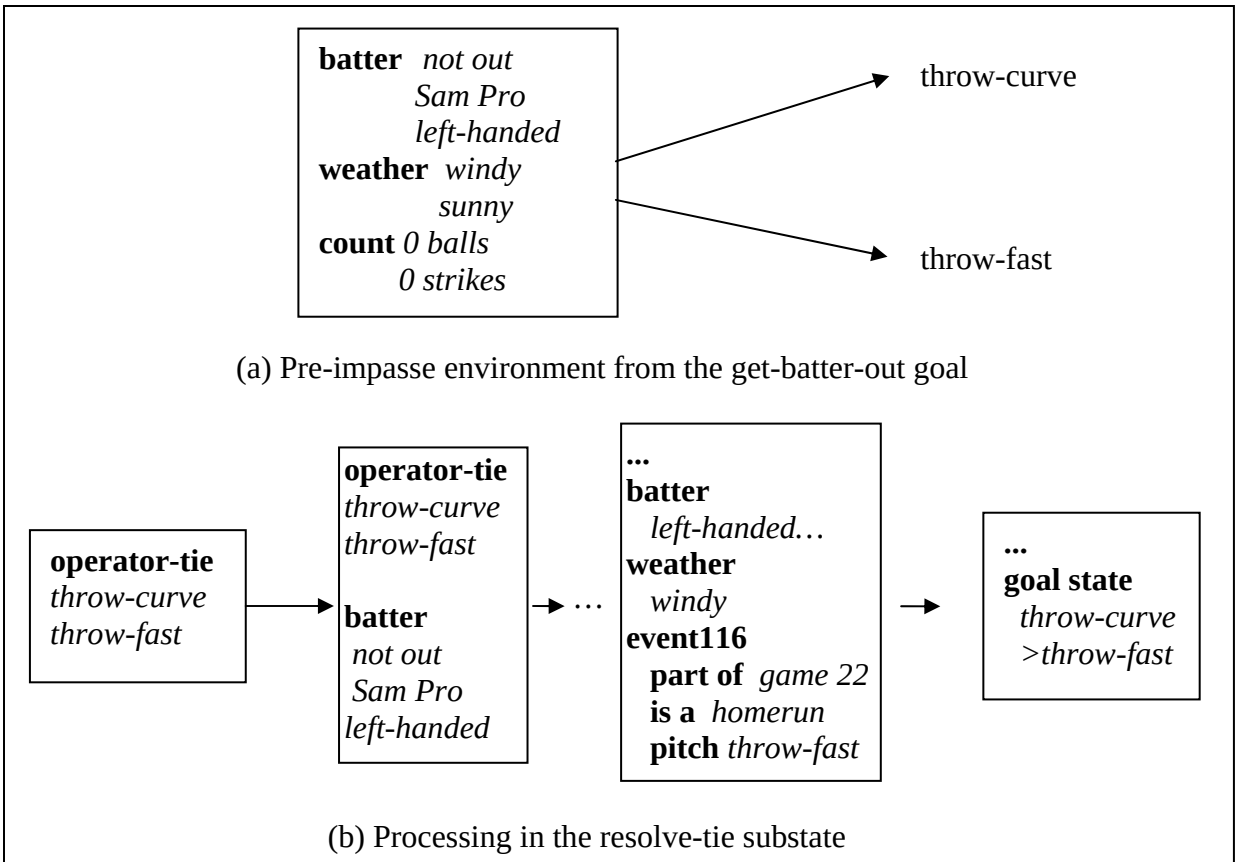

Figure 9: Learning when to prefer the curve ball. Elements in the higher goal context (the pre-impasse environment (a)) are used by operators available in the lower goal context (b) to elicit knowledge that resolves the operator tie. Here, the model elicits a "memory" of a previous game (event116) in which it was windy and throwing the fast ball to a left-handed batter led to a home run. On this basis, throwing the curve becomes preferred.

In general, then, working memory actually consists of a state/substate hierarchy where each substate exists in order to resolve an impasse in the problem space above it.[4] The hierarchy grows as impasses arise in the normal course of processing, and shrinks as impasses are resolved by eliciting the knowledge required to make progress in the higher context. Typically, decision cycles correspond to movement through the problem space that is at the bottom of the hierarchy, but it is not necessary that this be true; the decision cycle still results in only a single operator selection but if multiple changes are suggested in different states, the change to the state that is highest in the hierarchy is the one that occurs. Of course, if a change occurs to a context high up in the hierarchy, then the knowledge needed to overcome the impasse in that context must have become available, so all the substates in the hierarchy below the changed state, which were created because of the impasse, disappear.

[4] In fact, every state in Soar is established by the architecture, a fact we have glossed over in the preceding discussion for expository purposes. The highest level state is created automatically and includes all sensory data; all subsequent states result from impasses.

An operator tie is just one of the ways in which the decision cycle can fail to decide. An operator no-change impasse, for example, occurs when knowledge about when to select an operator is available, but knowledge about how it changes the state is not directly available. Operator no-change impasses often correspond to what are normally thought of as deliberate goals (or subgoals), where a complex problem is decomposed into simpler problems. In Soar, those simpler problems are actually operators that are selected, but which then lead to impasses where their application is pursued. For example, Joe might start by selecting an operator to get the opponent team out for an inning. That operator would lead to an impasse where operators to get each batter out are selected, which in turn lead to impasses where operator for specific pitches are selected and applied.

For our purposes, what is important is not to understand the particulars of the different kinds of impasse, but to understand that the full set of impasses defined in Soar is fixed and domain-independent. As with the vocabulary of preferences, the architecture defines a vocabulary of impasses so that it can process domain content without understanding the meaning of that content. Moreover, impasses can arise for every problem solving function, including proposing, selecting, and applying an operator, so that whenever the directly available knowledge is inadequate, additional knowledge, possibly stored in episodic or semantic memory, can be accessed in the substate.

Conceptually, the resolution of the operator-tie impasse can come about in two ways. Returning to our example, if the catcher signals for a specific pitch, then information coming in through perception might change the top state and elicit preferences for one pitch or the other (or for some other behavior altogether). In this case, there would be sufficient information to select an operator and the impasse would simply disappear. Alternatively, processing could continue in the lower context. Perhaps operators would elicit knowledge about how well Joe has done with each pitch in the past will resolve the impasse, or perhaps further subgoals will arise (if Joe is having difficulty retrieving relevant memories). In any of these cases, we must add more content to our model to have the model behave appropriately. Some of the content will be very specific, such as specific experiences stored in episodic memory or facts about baseball stored in semantic memory. But much of it will be more general and applicable to many situations, such as procedural knowledge about how to make decision (retrieve episodic memories, compare the results, and so on). By now it should be reasonably clear what form that knowledge should take and how to think about the way the architecture will use the knowledge. What is not yet clear is why this same impasse won't arise the next time our model faces Sam Pro. Certainly, if Joe were a seasoned veteran who had faced Sam many times in his career we would expect his choice of pitch to be automatic. But won't our model of Joe always have to plod along, recalling past events and reasoning its way to the same conclusion? Won't the model always reach this impasse? And just where do those memories come from, anyway?

8. LEARNING

There is a very old joke in which a man carrying a violin case stops a woman on the street and asks, "How do you get to Carnegie Hall?" and the woman replies, "Practice." Her advice reflects one of the most interesting regularities in human behavior; when we do something repeatedly we generally get better at it. Better may mean that we make fewer errors, or do the task more quickly, or simply seem to require less effort to achieve some given level of performance.

Somehow the act of doing the task *now* changes our ability to do the task in the future. In short, doing results in learning — the acquisition of knowledge through experience. It is this acquired knowledge that changes our behavior over time.

Three of the fundamental questions that arise when you talk about systems that learn are: What do they learn? What is the source of knowledge for learning? And when do they learn? We can answer a narrow version of the first of these questions from what we know about Soar already: Soar systems learn structures for its long-terms memories: rules, declarative facts, and episodes. To see why this must be so, remember our equation:

BEHAVIOR = ARCHITECTURE + CONTENT

Because learning leads to a change in behavior, either the architecture or the content must be affected by experience. Since the architecture is, by definition, a set of *fixed* structures and mechanisms, it must be the content that changes. And since domain content is captured by rules, facts, and episodes, learning must create those structures.[5] Although the architecture defines the types of structures that can be learned, it does not define either the knowledge that can be represented in those structures or the source of that knowledge. These are more difficult questions, as is the question of when to learn. In Soar, there are four different learning mechanisms, each of which uses a different source of knowledge, and together, they generate all of the different representations of knowledge in Soar.

For most of its existence, Soar has had a single learning mechanism called chunking. Recently, we have expanded Soar to include three additional learning mechanisms: reinforcement learning, semantic learning, and episodic learning. Chunking is the most developed learning mechanism. Running versions of the other learning mechanisms exist, but there are still many unanswered questions about them. Over the coming years, more will be written about the others, although the basics are included here.

8.1 CHUNKING

Initially we will concentrate on learning rules from the processing in substates. Figure 8(a) shows part of the context that led to the impasse in Figure 7 in slightly different form and with the details of the state expanded. We call this the *pre-impasse environment*, as shown in Figure 9(a). When an impasse arises it means that the system does not have rules available in long-term memory that lead to a single next move in the problem space. Expressed a bit more intuitively, an impasse is the architecture's way of signaling a lack of knowledge. A lack of knowledge is an opportunity for learning.

Now let's follow the processing in the resolve-tie substate, as shown in Figure 8(b). The operator-tie impasse arose because the original state did not provide cues for eliciting the LTM knowledge to choose between two pitches. In response to the impasse, a substate to find the knowledge was created by the architecture. Domain content then expanded this substate involving the recall and evaluate operators, which provide the context that could trigger relevant

[5] Strictly speaking, learning could lead to removing those structures as well. However, for now, Soar as a theory assumes that long-term memory only grows. So forgetting, in Soar, is a phenomenon that results when a new knowledge prevents old knowledge from being retrieved.

LTM knowledge. As a result of processing in the substate, the knowledge needed to choose the correct operator became available, and the impasse could be resolved. Moreover, we had an opportunity to learn a new rule that will make that knowledge immediately available under the circumstances that originally led to the impasse. In fact, the architecture automatically forms such a new rule whenever results are generated from an impasse. To distinguish rules that are learned by the model from those created by the modeler, we give them a special name, *chunks*, and call the learning process that creates them *chunking*.

To create a chunk the architecture looks at every part of the original state that existed in the pre-impasse environment (Figure 9(a)) that was used in reaching the result. It determines what was "used" by analyzing all of the LTM retrievals (rule firings, as well as episodic and semantic memory retrievals) that were on the path to generating a result. In our example, the new rule would be:

> (c1) If both the throw-curve-ball and throw-fast-ball operators are suggested and the **batter-name** is *Sam Pro* and the **batter** is *left-handed* and the **weather** is *windy* then prefer the throw-curve-ball operator.

As expected, the "if" portion of c1 includes only elements in the state before the impasse arose, and the "then" portion mirrors the knowledge elicited. With c1 in long-term memory, the model will not encounter an impasse the next time it is in a state like the one in Figure 9(a). Instead, this chunk will fire, the curve ball will be preferred, and processing will continue unabated.

Chunking is essentially a deductive, compositional learning mechanism; that is, the preference for a curve ball represents a kind of deduction from prior knowledge, and the new rule is composed from a "then" part containing the deduction, and an "if" part containing the knowledge that contributed to the deduction. This recombination of existing knowledge makes it accessible in new contexts via new long-term memory rules. Notice that chunking moves knowledge between different situations, overcoming some of the partitioning function of problem spaces and making the knowledge available in a problem space that it was not available in before learning, but that it does so only when experience dictates the need. As a theory, Soar says that chunking happens all the time — it is a ubiquitous mechanism that requires no intention to learn on the part of the agent. Impasses arise automatically when there is a lack of knowledge and chunks arise automatically when knowledge becomes available that helps resolve the impasse.

8.2 Reinforcement Learning

One source of knowledge that is outside of internal reasoning is feedback from the environment, or what is often called a "reward." The reward can come from the "body" in which a model is embedded (which to the model can be considered an external environment), or it can be generated internally when a goal is achieved. The reward can be either positive or negative, such as pleasure or pain, or an emotional reaction to a situation. Feedback can also be treated at a more abstract level, such as a reward for succeeding in a task or punishment for failure. In baseball, one can imagine there being positive reward for winning the game, getting an out, even throwing a strike, with negative reward for losing the game, allowing a run by the other team, etc. Reward can have immediate impact on behavior by informing a model that it should either

avoid the current situation (when there is negative reward) or attempt to maintain it (when there is positive reward). But reward can also be used to impact long term behavior through learning.

To capture this type of knowledge, we have integrated reinforcement learning with Soar (Nason & Laird, 2004). Based on experience, reinforcement learning adjusts predictions of future rewards, which are then used to select actions that maximize future expected rewards. Soar already represents knowledge for action selection as rules that generate preferences to select operators, so it is natural to learn rules that generate preferences based on future expected rewards.

There are two parts to reinforcement learning in Soar. First, Soar must learn rules that test the appropriate features of the states and operators. Second, Soar must learn the appropriate expected rewards for each rule. The first problem is the hardest and is the focus of active research. Our approach is to initially create rules based on the rules that propose operators, and then specialize those rules if they do not include sufficient conditions so that they consistently predict the same value. The second problem is to adjust the value predicted by rules. This is done by comparing the prediction of a rule that contributed to selecting an operator with what happens during the next decision. On the next decision, any reward that occurs is combined with the maximum predicted reward of the operators that are proposed. The resulting value is then used to update the expected reward of the contributing rules so that they better match what actually happened, and the selection of operators should improve over time.

Both chunking and reinforcement learning create new rules, but reinforcement learning creates only operator selection rules and the basis for creating them is statistical regularities in rewards as opposed to the results of internal reasoning. Although chunking could be used to learn rewards, it is difficult and cumbersome to organize problem solving in Soar for chunking to learn from external reward signals – it is much more straightforward and efficient to have a dedicated learning mechanism for expected reward. Within our example, reinforcement learning would play a background role, tuning the selection of pitching operators as experience accumulates. In this way, reinforcement learning directly supports the principle of rationality, leading to preferring operators that lead to greater reward.

8.3 Episodic Memory

Another source of knowledge that is available is the stream of experience. The memory of experiences is commonly called episodic memory (Tulving 1983). Episodic memory records events and history that are embedded in experience. For example, an episodic memory might include a specific baseball game when Joe threw a pitch that resulted in a home run. In contrast, the definition of a home run would be stored in semantic memory. An episode can be used to answer questions about the past (How did I get to the baseball park last time I was in this city?), to predict the outcome of possible courses of action (What happened the last time I pitched to this batter?), or to help keep track of progress on long-term goals (What did my wife ask me to buy at the store?). A set of episodes can improve behavior through other types of learning, such as reinforcement learning or chunking, by making it possible to replay an experience when there is time for more deliberate reflection.

In Soar, episodes are recorded automatically as a problem is solved (Nuxoll & Laird, 2004). An episode consists of a subset of the working memory elements that exist at the time of recording. Soar selects those working memory elements that have been used recently. To retrieve an episode, a cue is created in working memory (by rules). The episode that best matches the cue is found and recreated in working memory. It is tagged as an episode so that there is no confusion between what is a memory (the episode) and what is being experienced (the remainder of working memory). Once the episode is retrieved, it can trigger rule firings, or even be the basis for creating new cues for further searches of episodic memory.

Episodic memory differs from reinforcement learning and chunking in many ways. It is a passive learning mechanism that does not do any generalization. Another difference is that the contents of an episode are determined only indirectly by reasoning. Whatever is in working memory will be stored, and although reasoning determines what is in working memory, episodic learning ignores how the reasoning is done. This contrasts with reinforcement learning and chunking; both of which analyze the reasoning (in terms of rule firings) to determine what is learned. Finally, in episodic learning, there is no distinction between conditions for retrieval and what should be retrieved. There is just the episode, and any part of it can be used in the future for retrieval, and all of it will be retrieved. In contrast, the knowledge learned by chunking and reinforcement learning is asymmetric – there are distinct conditions and actions, and the conditions must exactly match to retrieve the actions (the actions can not be used to retrieve the conditions).

8.4 Semantic Memory

The final source of knowledge is the co-occurrence of structures in working memory. These co-occurrences are more general than episodes, which represent just single occurrences. They are distinct from what can be built into rules by chunking or reinforcement learning because they are about static structure instead of derivation-based rules. These are declarative structures, also called semantic knowledge, which is what you "know" (in contrast to what you "remember" in episodic memory) disassociated from the place and time when they are learned. For Joe Rookie, semantic knowledge would include knowledge about the rules of baseball, how many strikes and balls a batter can get at a turn at bat, what is a home run, etc. A structure from semantic memory is retrieved in the same manner as is an episode – a cue is deliberately created in working memory by rules, the cue is then used to search for the best match in long term memory, and then the complete memory is retrieved. In general, a semantic memory is much smaller and more compact than an episode, representing the relationships between a few elements in working memory instead of the complete memory. Less clear, and a subject of active research is when to store a structure in semantic memory – should every structure that ever occurs in working memory be stored?

9. Putting it all together: a Soar model of Joe Rookie

In our scenario, Joe Rookie chooses to throw a curve ball which is hit by the batter, caught by Joe on a bounce, and thrown to first base. Over the last five sections, we've moved from this narrative description of the activity of a rookie pitcher to a computational description of the architecture and content that produces that activity. Of course, we've only described a fraction of the knowledge involved in deciding to throw a curve ball and an even smaller fraction of the knowledge involved in playing out our original scenario. To expand the model to full

functionality is beyond the scope of this chapter, but the method we would use should be clear. First, we specify the domain knowledge Joe needs; that's our content. We must specify which memories it is stored in. We must then tie the domain knowledge to state structures (including percepts and any other working memory elements that trigger actions), and operators. Next, we must specify the relationships between different levels by the impasses that will arise and the kinds of knowledge that will be missing, and consequently learned.

Figure 10 shows what a fuller model might look like in terms of problem spaces and their relationships. In Soar, the top state is given by the architecture and is the only one that persists across time (other states come and go as the state hierarchy grows and shrinks). All perception and action occurs through the top state so that they are part of the pre-impasse environment for all substates (this ensures that all knowledge from the environment is available to aid decision making for the top state so that input is never hidden in lower levels.). In the figure, operators in the top problem space correspond to tasks Joe has to perform, like getting a batter out, comprehending language or gestures, and so on. Operators in substates implement these high level tasks. Thus, our now-familiar *pitch* substate is actually evoked in response to an impasse in Soar's Top state. Knowledge may be spread out among the substates in various ways, but will be integrated as a result of chunking in response to impasses. The dark arrows connect contexts in the state hierarchy that arise while coming to the decision to throw the curve ball as we discussed it above. The chunk that arises in the Top state under this configuration would look like c1 except that it would check for the task operator of getting the batter out (because these are elements in the pre-impasse environment that are used in arriving at the result).

Another possibility evident from the figure is that while Joe is trying to decide on a pitch, the catcher might just signal the appropriate pitch. Under the configuration shown, the impasse from the Top state to *pitch* would have to be terminated (by rules that prefer comprehension of the catcher's signal to getting the batter out), and the comprehend operator selected. Then an impasse on the comprehend operator would redirect processing along the dotted lines, into the Comprehend substate and downward, in this case, to the Gesture substate. The chunk resulting from the impasse on comprehend might map directly to the choice of a pitch and a motor action in the Top state. Alternatively, it might simply result in an augmentation to the Top state (e.g. with **suggested action** *throw-curve-ball*). In this latter case, the get-batter-out operator would be reselected and impasse again, allowing the catcher's advice to be weighed against evidence from Recall in the substate.

We could go on exploring alternatives to the model shown in Figure 10, elaborating it with more states and operators, moving operators or the connections between states around to explore the effect on learning, and so on. Clearly, the architecture permits many different ways of organizing domain content. It is possible that some of this variation must be there in order to model the natural variations in behavior that occur both within and across people. However, if even after taking such variability into account, the architecture still does not sufficiently constrain the structuring of the domain content, how has its specification helped us in our search for a unified theory of cognition? Before we address this question, let's step back and draw together in one place all the mechanisms and structures of the architecture we have developed above.

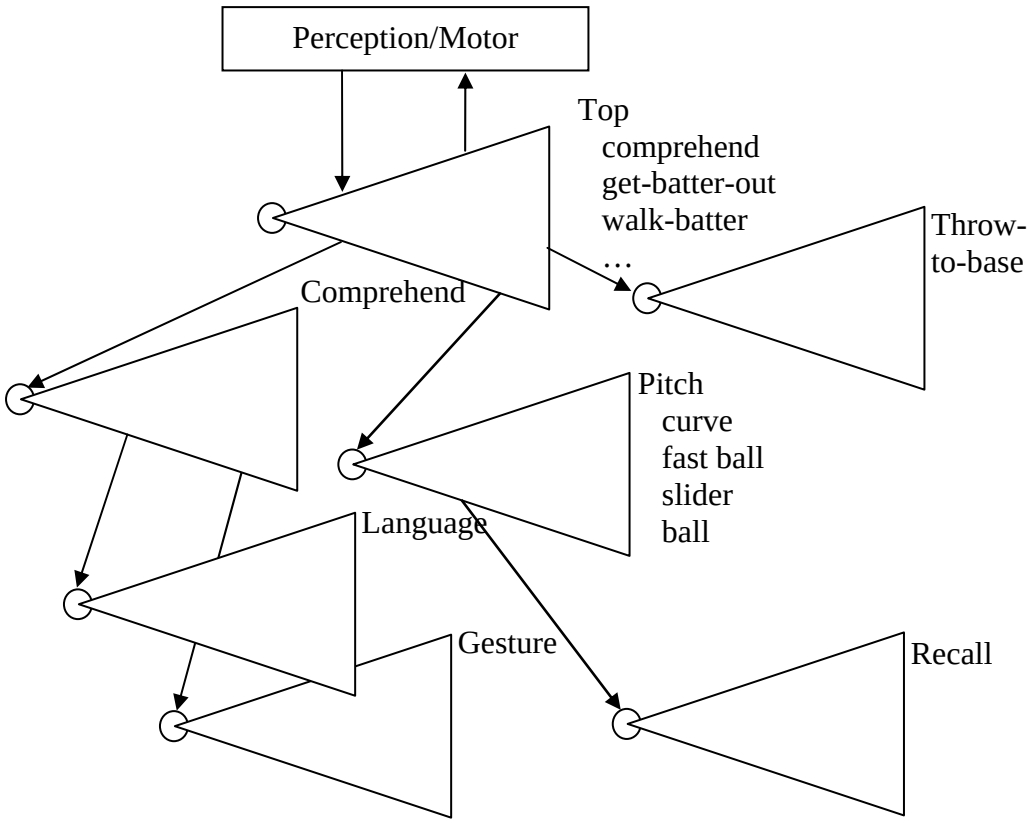

Figure 10: Expanding the model's capabilities. Our model of Joe needs far more knowledge to play out our original scenario than we have given it in the previous sections. Still, increasing the model's functionality can be done within the architectural framework we've described by adding more domain content in the form of LTM knowledge that is used for additional types of states and operators.

10. STEPPING BACK: THE SOAR ARCHITECTURE IN REVIEW

A cognitive architecture is really two things at once. First, it is a fixed set of mechanisms and structures that process content to produce behavior. At the same time, however, it is a theory, or point of view, about what cognitive behaviors have in common. In this chapter we have looked at one particular architecture, Soar, and so at one particular set of mechanisms and one particular point of view. Here is a brief synopsis of the mechanisms and structures we have discussed, along with some of the theoretical commitments that underlie them:

- **States and Operators** are the basic structures supported by the architecture. The states contain all the information about the current situation, including perception (in the Top state) and descriptions of current goals and problem spaces, while operators are the means for taking steps in problem spaces.
- **Working Memory (WM)** contains perceptions and the hierarchy of states and their associated operators. The contents of working memory trigger retrieval from LTM and motor actions.
- **Long-term Memory (LTM)** is the repository for domain content that is processed by the architecture to produce behavior. Soar supports three different representations of LTM: procedural knowledge encoded as rules, semantic knowledge encoded as declarative

structures, and episodic knowledge encoded as episodes. Procedural memory is accessed automatically during the decision cycle, while the semantic and episodic memories are accessed deliberately through the creation of specific cues in working memory. All of Soar's long-term memories are impenetrable. This means that a Soar system cannot examine its LTMs directly; its only window into LTM is through the changes to working memory that arise from memory retrievals.

- **The Perception/Motor Interface** is the mechanism for defining mappings from the external world to the internal representation in working memory, and from the internal representation back out to action in the external world. Through this interface, perception and action can occur in parallel with cognition.
- **The Decision Cycle** is the basic architectural process supporting cognition. At Soar's core is an operator selection and application machine. The decision cycle is composed of three phases. The elaboration phase involves parallel access to LTM to elaborate the state, suggests new operators, and evaluates the operators. Quiescence defines the moment at the end of the elaboration phase and the beginning of the decision phase. During the decision phase, the decision procedure interprets the domain-independent language of operator preferences. The result of the decision procedure is either a change to the selected operator or an impasse if the preferences are incomplete or in conflict. Following decision is the application phase, where rules fire to modify the state. The selection of a single operator per decision cycle imposes a *cognitive bottleneck* in the architecture, that is, a limit on how much work cognition can do at once.
- **Impasses** signal a lack of knowledge, and therefore, an opportunity for learning. An impasse occurs automatically whenever the knowledge elicited by the current state isn't enough for the decision procedure to resolve the preferences in working memory to select an operator. The language of impasses, like the language of preferences, is defined independently of any domain. When an impasse arises, the architecture also automatically begins the creation of a new substate whose goal is to resolve the impasse. In this way, impasses impose a goal/substate hierarchy on the contexts in working memory.
- **Four Learning Mechanisms:** Soar has four architectural learning mechanisms. Chunking automatically creates new rules in LTM whenever results are generated from an impasse. It speeds up performance and moves more deliberate knowledge retrieved in a substate up to a state where it can be used reactively. The new rules map the relevant pre-impasse WM elements into WM changes that prevent that impasse in similar future situations. Reinforcement learning adjusts the values of preferences for operators. Episodic learning stores a history of experiences, while semantic learning captures more abstract declarative statements.

Using these structures and mechanisms we have shown how to model a small piece of cognitive behavior in the Soar architecture. Our purpose in building the model was to ground our discussion in some concrete, familiar behavior so it would be easy to see what was architectural — and, therefore, fixed — and what was not. It is important to realize, however, that the particular examples we modeled are only that, examples. To see how the Soar architecture can be used to help produce some other specific behavior, we would begin by viewing that behavior through the framework introduced in Section 3, describing the relevant goals that drive the behavior and the knowledge required to accomplish them. Once we have a knowledge-level description, we would tie the knowledge to goals, states, and operators. In partitioning the

knowledge among different states, we would attend to the relationships defined by the different types of impasses so that learning results in meaningful new rules. These are the steps to building a Soar model, whether it is a model of a baseball pitcher, a student solving physics problems, or a jet fighter pilot engaging an enemy plane.

We have asked and answered a lot of questions about Soar. What structures and mechanisms are fixed? What structures and mechanisms are left up to the modeler? What processes occur automatically? What processes must be chosen deliberately? By answering these questions we have told you about the architecture we use. By asking them, we have tried to give you a method for analyzing and understanding other architectures.

The power behind the idea of cognitive architecture, any cognitive architecture, is that it lets us explore whether a single set of assumptions, a single theory of what is common among many cognitive behaviors, can in fact support all of cognition. However, defining the architecture, the fixed mechanisms and structures, is only a step toward producing a unified theory of cognition. The architecture, by itself, does not answer all the questions we might want to ask about cognition. Still, given the architecture as a starting point, we know that additional theory must be cast as content that can be processed by that architecture. In short, the architecture is the framework into which the *content theories* must fit. Moreover, the framework provides the opportunity to combine content theories and watch how they interact — to move them from isolation into a unified system.

11. FROM ARCHITECTURE TO UNIFIED THEORIES OF COGNITION

Our model of Joe Rookie is a content theory, although it is hard to imagine what in its content we might want to argue is general across human cognition. As a community of more than 100 researchers world-wide, the Soar project has contributed a number of more interesting content theories to the field. One example is NL-Soar, a theory of human natural language (NL) comprehension and generation (Lehman, Lewis, & Newell, 1991; Rubinoff & Lehman, 1994) that has been used to explain many results from psycholinguistics, including the garden path phenomenon discussed in Section 1 (Lewis, 1993). Research on NL-Soar has continued to include theories of natural language generation and translation (Lonsdale & Melby, 2003). Other examples include work on learning to intermix tasks through experience (Chong, 1998), theories of symbolic concept learning that acquires and retrieves category prediction rules (e.g. if you see something round and small that rolls, predict it is a ball) (Miller, & Laird 1996), and a model of student learning in microworlds (Miller, Lehman, & Koedinger, 1999)

Each of these content theories has, in its own way, been an attempt at unification because each has explained a set of phenomena that had not been explained by a single theory previously. Moreover, because these content theories have been expressed computationally as Soar models, they all share the assumptions in the Soar architecture. What this means is that even though a content theory models aspects of human language (or concept learning, or multi-tasking), it does so within a framework that makes certain assumptions about human problem solving, memory, etc. The resulting model may not, itself, show the full range of human problem solving or memory phenomena, but the theory will, at the very least, be compatible with what is assumed to be architectural in the other content theories. In this weak sense, content theories like the ones above constitute a burgeoning unified theory of cognition (UTC) (Newell, 1990).

Of course, the stronger notion of Soar as a UTC requires that these individual content theories all work together in a single computational model. A number of other Soar projects have focused on variations of that goal, producing models such as:

- NTD-Soar, a computational theory of the perceptual, cognitive, and motor actions performed by the NASA Test Director (NTD) as he utilizes the materials in his surroundings and communicates with others on the Space Shuttle launch team (Nelson, Lehman, & John, 1994). NTD-Soar combined NL-Soar and NOVA (a model of vision) with decision making and problem solving knowledge relevant to the testing and preparation of the Space Shuttle before it is launched.
- Instructo-Soar, a computational theory of how people learn through interactive instruction (Huffman, 1993). Instructo-Soar combined NL-Soar with knowledge about how to use and learn from instructions during problem solving to produce a system that learned new procedures through natural language.
- IMPROV, a computational theory of how to correct knowledge about what actions do in the world (Pearson & Laird, 1995). IMPROV combined SCA (a model of category learning) with knowledge about how to detect when the system's internal knowledge conflicts with what it observes in the world so that the system can automatically improve its knowledge about its interactions with the world.
- TacAir-Soar (Jones et al. 1999) and RWA-Soar (Tambe et al. 1995) were Soar models of human pilots used in training in large-scale distributed simulations. They remain the largest Soar systems built (TacAir-Soar is currently > 8000 rules) and could perform all of the U.S. tactical air missions for fixed wing (jets and propeller planes) and rotary wing (helicopters) aircraft. The aircraft could fly the missions autonomously, dynamically changing the missions, the command structure, all of the while communicating using standard military language. This work included early work on theories of coordination and teamwork that was realized in the STEAM system (Tambe, 1997).
- Soar MOUTBOT (Wray et al. 2004) is similar in spirit to TacAir-Soar but developed for military operations in urban terrain (MOUT). In this system, Soar controlled individual adversaries who would defend a building against attack by military trainees. It required integration of reaction, planning, communication and coordination, and spatial reasoning.

The complex behavior achieved by each of these systems was made possible, in part, because some component content theories were already available. Using Soar as a common framework meant that there already existed a language capability (or visual attention mechanism, or concept acquisition mechanism) cast in the right terms and ready (more or less) for inclusion. From the point of view of the NTD project, for example, the availability of NL-Soar meant that they didn't have to build a theory of human language in addition to a theory of the launch domain. More importantly, incorporating NL-Soar meant that the whole model had to "play by the rules" linguistically. NTD-Soar couldn't be constructed to wait to pay attention to linguistic input indefinitely because NL-Soar says that acoustic information doesn't stay around indefinitely. It couldn't wait until the end of the sentence to produce the meaning because NL-Soar says meaning has to be produced incrementally, word-by-word. In essence, NTD-Soar was constrained by incorporating NL-Soar. But, of course, the constraints hold in both directions. From the point of view of the NL-Soar project, the NASA Test Director's linguistic environment presented many challenges that had not originally shaped the model. Indeed, NL-Soar had been

created as a model of comprehension and generation in isolation. The need to actually comprehend and generate sentences while doing other things changed the structure of NL-Soar for all time.

As these examples show, we are still a long way from creating a full unified theory of cognition. As a candidate UTC, Soar currently insists that theories of regularities in behavior be cast in terms of long-term memory, goals, states, operators, impasses, and a variety of learning mechanisms. Whether these are the right assumptions, whether we can construct on top of these assumptions a set of computationally-realizable mechanisms and structures that can answer all the questions we might want to ask about cognitive behavior, is an empirical question. Even now we are looking beyond what is commonly included in cognition and attempting to integrate a theory of emotion in Soar (Marinier & Laird, 2004). Still, our methodology in working toward that goal is clear: work within the architecture, base content theories on the regularities already available from the contributing disciplines of cognitive science, and combine those content theories to try to explain increasingly complex behaviors. In short, keep searching for the big picture.

12. REFERENCES

1. Anderson, J. R. 1993. *Rules of the Mind*. Hillsdale, NJ: Lawrence Erlbaum Associates.
2. Chong, R. S. 1998. Modeling dual-task performance improvement: Casting executive process knowledge acquisition as strategy refinement. Doctoral Dissertation, Department of Computer Science and Engineering, University of Michigan.
3. The Commissioner of Baseball, Major League Baseball, 1998, http://mlb.mlb.com/NASApp/mlb/mlb/official_info/official_rules/foreword.jsp
4. Fitts, P. M. 1954. The information capacity of the human motor system in controlling the amplitude of movement. *Journal of Experimental Psychology, 47*, 381-391.
5. Gibson, E. 1990. Recency preference and garden-path effects. *Twelfth Annual Conference of the Cognitive Science Society*, 372-379.
6. Jones, R. M., Laird, J. E., Nielsen P. E., Coulter, K., Kenny, P., and Koss, F. 1999. Automated Intelligent Pilots for Combat Flight Simulation, *AI Magazine*, Vol. 20, No. 1, pp. 27-42.
7. Kieras, D.E., Wood, S.D., and Meyer, D.E. 1997. Predictive engineering models based on the EPIC architecture for a multimodal high-performance human-computer interaction task. *ACM Transactions on Computer-Human Interaction*. 4, 230-275.
8. Langley, P., & Laird, J. E. 2002. Cognitive architectures: Research issues and challenges (Technical Report). Institute for the Study of Learning and Expertise, Palo Alto, CA.
9. Lehman, J. Fain, Lewis, R., and Newell, A. 1991. Integrating knowledge sources in language comprehension. *Proceedings of the Thirteenth Annual Conferences of the Cognitive Science Society*, 461-466.
10. Lewis, R. L. 1993. *An Architecturally-based Theory of Human Sentence Comprehension*. Ph.D. diss., Carnegie Mellon University. Also available as Technical Report CMU-CS-93-226.
11. Lonsdale, D. and Melby, A. 2003. Machine Translation, Translation Studies, and Linguistic Theories; *Logos and Language*, 3(1): 39-57; Gunter Narr Verlag, Tübingen.
12. Marinier, R., Laird, J. 2004. Toward a Comprehensive Computational Model of Emotions and Feelings, *International Conference on Cognitive Modeling.*

13. Miller, C. S. 1993. *Modeling Concept Acquisition in the Context of a Unified Theory of Cognition*. Ph.D. diss., The University of Michigan. Also available as Technical Report CSETR- 157-93.
14. Miller, C. S. and Laird, J. E. 1996. Accounting for graded performance within a discrete search framework. *Cognitive Science*, 20, 499-537.
15. Miller, C. S., Lehman, J. F. and Koedinger, K. R. 1999. Goals and Learning in Microworlds. *Cognitive Science*, 23, 305-336.
16. Nason, S. & Laird, J. E. 2004. Soar-RL, Integrating Reinforcement Learning with Soar, *International Conference on Cognitive Modeling.*
17. Nelson, G., Lehman, J. F., John, B. 1994. Integrating cognitive capabilities in a real-time task. *Proceedings of the Sixteenth Annual Conference of the Cognitive Science Society*, 658-663.
18. Newell, A. 1982. The knowledge level. *Artificial Intelligence,* 18, 87-127.
19. Newell, A. 1990. *Unified Theories of Cognition.* Cambridge, Massachusetts: Harvard University Press. Newell, A. & Simon, H. 1972. *Human Problem Solving*. Englewood Cliffs, New Jersey: Prentice-Hall.
20. Newell, A., Rosenbloom, P. S., & Laird, J. E. 1989. Symbolic architectures for cognition. In M. I. Posner (Ed.), *Foundations of Cognitive Science*. Cambridge, MA: Bradford Books/MIT Press.
21. Nilsson, Nils. 1971. *Problem-solving Methods in Artificial Intelligence*. New York, New York: McGraw-Hill.
22. Nuxoll, A., and Laird, J. 2004. A Cognitive Model of Episodic Memory Integrated With a General Cognitive Architecture, *International Conference on Cognitive Modeling*.
23. Pearson, D. J. and Laird, J. E. 1995. Toward Incremental Knowledge Correction for Agents in Complex Environments. In S. Muggleton, D. Michie, and K. Furukawa (Ed.), *Machine Intelligence*. Oxford University Press.
24. Pritchett, B. L. 1988. Garden path phenomena and the grammatical basis of language processing. *Language,* 64, 539-576.
25. Pylyshyn, Z. W. 1984. *Computation and Cognition: Toward a foundation for cognitive science*. Cambridge, MA: Bradford.
26. Rosenbloom, P. S., Laird, J. E., and Newell, A. (Eds.). 1993. *The Soar Papers: Research on Integrated Intelligence*. Cambridge, MA: MIT Press.
27. Rubinoff, R. and Lehman, J. F. 1994. Real-time natural language generation in NL-Soar. *Proceedings of the Seventh International Workshop on Natural Language Generation*, 199-206 .
28. Sternberg, S. 1975. Memory scanning: New findings and current controversies. *Quarterly Journal of Experimental Psychology, 27*, 1-32.
29. Sternberg, S. & Scarborough, D. 1996. (Eds), *Invitation to Cognitive Science*, Volume 4. Cambridge, MA: MIT Press.
30. Tambe, M. 1997. Agent architectures for flexible, practical teamwork. In *Proceedings of the National Conference on Artificial Intelligence*.
31. Tambe, M., Johnson, W. L., Jones, R. M., Koss, F., Laird, J. E., Rosenbloom, P. S., and Schwamb, K. 1995. Intelligent Agents for Interactive Simulation Environments. *AI Magazine,* 16(1).
32. Tulving, E. 1983. *Elements of Episodic Memory*. Oxford: Clarendon Press.

33. Tulving, E. 2002. Episodic Memory: From mind to brain. *Annual Review of Psychology*, 53, 1-25.
34. Wray, R.E., and Laird, J. E. 2003. An architectural approach to consistency in hierarchical execution. *Journal of Artificial Intelligence Research*. 19. 355-398.
35. Wray. R. E., Laird, J. E., Nuxoll, A., Stokes, D., and Kerfoot, A. 2004. Synthetic Adversaries for Urban Combat Training, *Proceedings of the 2004 Innovative Applications of Artificial Intelligence Conference*, San Jose, CA.