The Soar User's Manual

Version 9.3.3

John E. Laird and Clare Bates Congdon
User interface sections by Karen J. Coulter
Soar 9 Modules by Nate Derbinsky and Joseph Xu
Computer Science and Engineering Department
University of Michigan

Draft of: June 12, 2014

Contents

List of Figures

Chapter 1

Introduction

Soar has been developed to be an architecture for constructing general intelligent systems. It has been in use since 1983, and has evolved through many different versions. This manual documents the most current of these: Soar, version 9.3.3.

Our goals for Soar include that it is to be an architecture that can:

- be used to build systems that work on the full range of tasks expected of an intelligent agent, from highly routine to extremely difficult, open-ended problems;
- represent and use appropriate forms of knowledge, such as procedural, declarative, episodic, and possibly iconic;
- employ the full range of problem solving methods;
- interact with the outside world; and
- learn about all aspects of the tasks and its performance on those tasks.

In other words, our intention is for Soar to support all the capabilities required of a general intelligent agent. Below are the major principles that are the cornerstones of Soar's design:

1. The number of distinct architectural mechanisms should be minimized. Classically Soar had a single representation of permanent knowledge (productions), a single representation of temporary knowledge (objects with attributes and values), a single mechanism for generating goals (automatic subgoaling), and a single learning mechanism (chunking). It was only as Soar was applied to diverse tasks in complex environments that we found these mechanisms to be insufficient and have recently added new long-term memories (semantic and episodic) and learning mechanisms (semantic, episodic, and reinforcement learning) to extend Soar agents with crucial new functionalities.

2. All decisions are made through the combination of relevant knowledge at run-time. In Soar, every decision is based on the current interpretation of sensory data and any relevant knowledge retrieved from permanent memory. Decisions are never precompiled into uninterruptible sequences.

1.1   Using this Manual

We expect that novice Soar users will read the manual in the order it is presented:

Chapter 2 and Chapter 3 describe Soar from different perspectives: **Chapter 2** describes the Soar architecture, but avoids issues of syntax, while **Chapter 3** describes the syntax of Soar, including the specific conditions and actions allowed in Soar productions.

Chapter 4 describes chunking, Soar's mechanism to learn new procedural knowledge. Not all users will make use of chunking, but it is important to know that this capability exists.

Chapter 5 describes reinforcement learning (RL), a mechanism by which Soar's procedural knowledge is tuned given task experience. Not all users will make use of RL, but it is important to know that this capability exists.

Chapter 6 and Chapter 7 describe Soar's long-term declarative memory systems, semantic and episodic. Not all users will make use of these mechanisms, but it is important to know that they exist.

Chapter 8 describes the Soar user interface — how the user interacts with Soar. The chapter is a catalog of user-interface commands, grouped by functionality. The most accurate and up-to-date information on the syntax of the Soar User Interface is found online, on the Soar Wiki, at

> http://code.google.com/p/soar/.

Advanced users will refer most often to Chapter 8, flipping back to Chapters 2 and 3 to answer specific questions.

There are several appendices included with this manual:

Appendix A contains an example Soar program for a simple version of the blocks world. This blocks-world program is used as an example throughout the manual.

Appendix B provides a grammar for Soar productions.

Appendix C describes the determination of o-support.

Appendix D provides a detailed explanation of the preference resolution process.

Appendix E provides an explanation of the Goal Dependency Set.

Additional Back Matter

The appendices are followed by an index; the last pages of this manual contain a summary and index of the user-interface functions for quick reference.

Not Described in This Manual

Some of the more advanced features of Soar are not described in this manual, such as how to interface with a simulator, or how to create Soar applications using multiple interacting agents. A discussion of these topics is provided in a separate document, the *SML Quick Start Guide*, which is available at the Soar project website (see link below).

For novice Soar users, try *The Soar 9 Tutorial*, which guides the reader through several example tasks and exercises.

See Section 1.2 for information about obtaining Soar documentation.

1.2   Contacting the Soar Group

Resources on the Internet

The primary website for Soar is:

http://sitemaker.umich.edu/soar/

Look here for the latest downloads, documentation, and Soar-related announcements, as well as links to information about specific Soar research projects and researchers and a FAQ (list of frequently asked questions) about Soar.

Soar kernel development is hosted on Google Code at

http://code.google.com/p/soar/

This site contains the public subversion repository, active documentation wiki, and is also where bugs should be reported.

For questions about Soar, you may write to the Soar e-mail list at:

soar-group@lists.sourceforge.net

If you would like to be on this list yourself, visit:

http://lists.sourceforge.net/lists/listinfo/soar-group/

For Those Without Internet Access

If you cannot reach us on the internet, please write to us at the following address:

> The Soar Group
> Artificial Intelligence Laboratory
> University of Michigan
> 2260 Hayward Street
> Ann Arbor, MI 48109-2121
> USA

1.3   A Note on Different Platforms and Operating Systems

Soar runs on a wide variety of platforms, including Linux, Unix (although not heavily tested), Mac OS X, and Windows 7, Vista, and XP (and probably 2000 and NT).

This manual documents Soar generally, although all references to files and directories use Unix format conventions rather than Windows-style folders.

Chapter 2

The Soar Architecture

This chapter describes the Soar architecture. It covers all aspects of Soar except for the specific syntax of Soar's memories and descriptions of the Soar user-interface commands.

This chapter gives an abstract description of Soar. It starts by giving an overview of Soar and then goes into more detail for each of Soar's main memories (working memory, production memory, and preference memory) and processes (the decision procedure, learning, and input and output).

2.1 An Overview of Soar

The design of Soar is based on the hypothesis that all deliberate *goal*-oriented behavior can be cast as the selection and application of *operators* to a *state*. A state is a representation of the current problem-solving situation; an operator transforms a state (makes changes to the representation); and a goal is a desired outcome of the problem-solving activity.

As Soar runs, it is continually trying to apply the current operator and select the next operator (a state can have only one operator at a time), until the goal has been achieved. The selection and application of operators is illustrated in Figure 2.1.

Soar has separate memories (and different representations) for descriptions of its current

Soar execution

select   apply   select   apply   select   apply   • • •
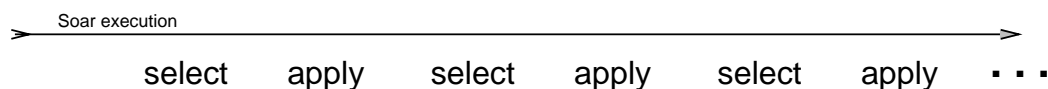
Figure 2.1: Soar is continually trying to select and apply operators.

situation and its long-term knowledge. In Soar, the current situation, including data from sensors, results of intermediate inferences, active goals, and active operators is held in *working memory*. Working memory is organized as *objects*. Objects are described in terms of their *attributes*; the values of the attributes may correspond to sub-objects, so the description of the state can have a hierarchical organization. (This need not be a strict hierarchy; for example, there's nothing to prevent two objects from being "substructure" of each other.)

The long-term knowledge, which specifies how to respond to different situations in working memory, can be thought of as the program for Soar. The Soar architecture cannot solve any problems without the addition of long-term knowledge. (Note the distinction between the "Soar architecture" and the "Soar program": The former refers to the system described in this manual, common to all users, and the latter refers to knowledge added to the architecture.)

A Soar program contains the knowledge to be used for solving a specific task (or set of tasks), including information about how to select and apply operators to transform the states of the problem, and a means of recognizing that the goal has been achieved.

2.1.1   Problem-Solving Functions in Soar

All of Soar's long-term knowledge is organized around the functions of operator selection and operator application, which are organized into four distinct types of knowledge:

Knowledge to select an operator

1. *Operator Proposal:* Knowledge that an operator is appropriate for the current situation.

2. *Operator Comparison:* Knowledge to compare candidate operators.

3. *Operator Selection:* Knowledge to select a single operator, based on the comparisons.

Knowledge to apply an operator

4. *Operator Application:* Knowledge of how a specific operator modifies the state.

In addition, there is a fifth type of knowledge in Soar that is indirectly connected to both operator selection and operator application:

5. Knowledge of monotonic inferences that can be made about the state (*state elaboration*).

State elaborations indirectly affect operator selection and application by creating new descriptions of the current situation that can cue the selection and application of operators.

These problem-solving functions are the primitives for generating behavior in Soar. Four of the functions require retrieving long-term knowledge that is relevant to the current situation: elaborating the state, proposing candidate operators, comparing the candidates, and applying the operator by modifying the state. These functions are driven by the knowledge

encoded in a Soar program. Soar represents that knowledge as *production* rules. Production rules are similar to "if-then" statements in conventional programming languages. (For example, a production might say something like "if there are two blocks on the table, then suggest an operator to move one block ontop of the other block"). The "if" part of the production is called its *conditions* and the "then" part of the production is called its *actions*. When the conditions are met in the current situation as defined by working memory, the production is *matched* and it will *fire*, which means that its actions are executed, making changes to working memory.

The other function, selecting the current operator, involves making a decision once sufficient knowledge has been retrieved. This is performed by Soar's *decision procedure*, which is a fixed procedure that interprets *preferences* that have been created by the retrieval functions. The knowledge-retrieval and decision-making functions combine to form Soar's *decision cycle*.

When the knowledge to perform the problem-solving functions is not directly available in productions, Soar is unable to make progress and reaches an *impasse*. There are three types of possible impasses in Soar:

1. An operator cannot be selected because none are proposed.

2. An operator cannot be selected because multiple operators are proposed and the comparisons are insufficient to determine which one should be selected.

3. An operator has been selected, but there is insufficient knowledge to apply it.

In response to an impasse, the Soar architecture creates a *substate* in which operators can be selected and applied to generate or deliberately retrieve the knowledge that was not directly available; the goal in the substate is to resolve the impasse. For example, in a substate, a Soar program may do a lookahead search to compare candidate operators if comparison knowledge is not directly available. Impasses and substates are described in more detail in Section 2.6.

2.1.2 An Example Task: The Blocks-World

We will use a task called the blocks-world as an example throughout this manual. In the blocks-world task, the initial state has three blocks named `A`, `B`, and `C` on a table; the operators move one block at a time to another location (on top of another block or onto the table); and the goal is to build a tower with `A` on top, `B` in the middle, and `C` on the bottom. The initial state and the goal are illustrated in Figure 2.2.

The Soar code for this task is included in Appendix A. You do not need to look at the code at this point.

The operators in this task move a single block from its current location to a new location; each operator is represented with the following information:

- the name of the block being moved
- the current location of the block (the "thing" it is on top of)
- the destination of the block (the "thing" it will be on top of)
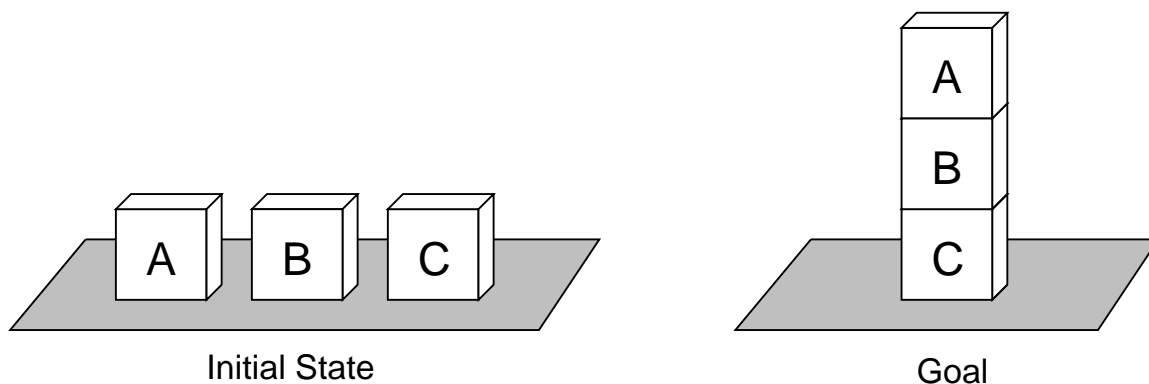
Figure 2.2: The initial state and goal of the "blocks-world" task.

An Abstract View of Working Memory
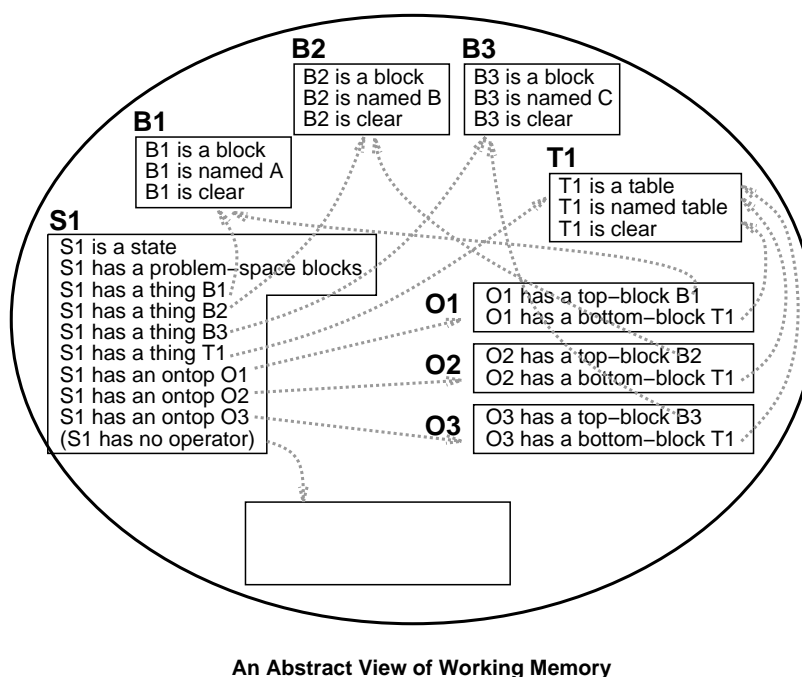
Figure 2.3: An abstract illustration of the initial state of the blocks world as working memory objects. At this stage of problem solving, no operators have been proposed or selected.

The goal in this task is to stack the blocks so that `C` is on the table, with block `B` on block `C`, and block `A` on top of block `B`.

2.1.3   Representation of States, Operators, and Goals

The initial state in our blocks-world task — before any operators have been proposed or selected — is illustrated in Figure 2.3.

A state can have only one operator at a time, and the operator is represented as substructure of the state. A state may also have as substructure a number of *potential* operators that

An Abstract View of Working Memory
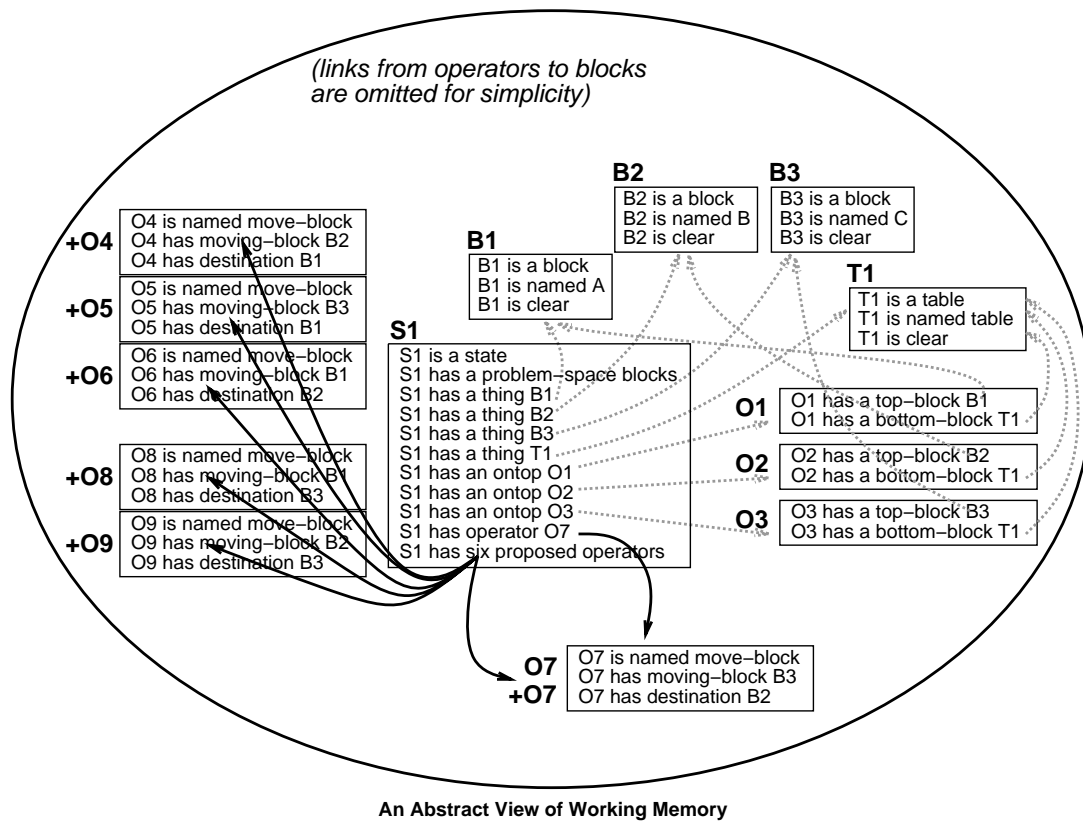
Figure 2.4: An abstract illustration of working memory in the blocks world after the first operator has been selected.

are in consideration; however, these suggested operators should not be confused with the current operator.

Figure 2.4 illustrates working memory after the first operator has been selected. There are six operators proposed, and only one of these is actually selected.

Goals are either represented explicitly as substructure of the state with general rules that recognize when the goal is achieved, or are implicitly represented in the Soar program by goal-specific rules that test the state for specific features and recognize when the goal is achieved. The point is that sometimes a description of the goal will be available in the state for focusing the problem solving, whereas other times it may not. Although representing a goal explicitly has many advantages, some goals are difficult to explicitly represent on the state.

The goal in our blocks-world task is represented implicitly in the Soar program. A single production rule monitors the state for completion of the goal and halts Soar when the goal is achieved.
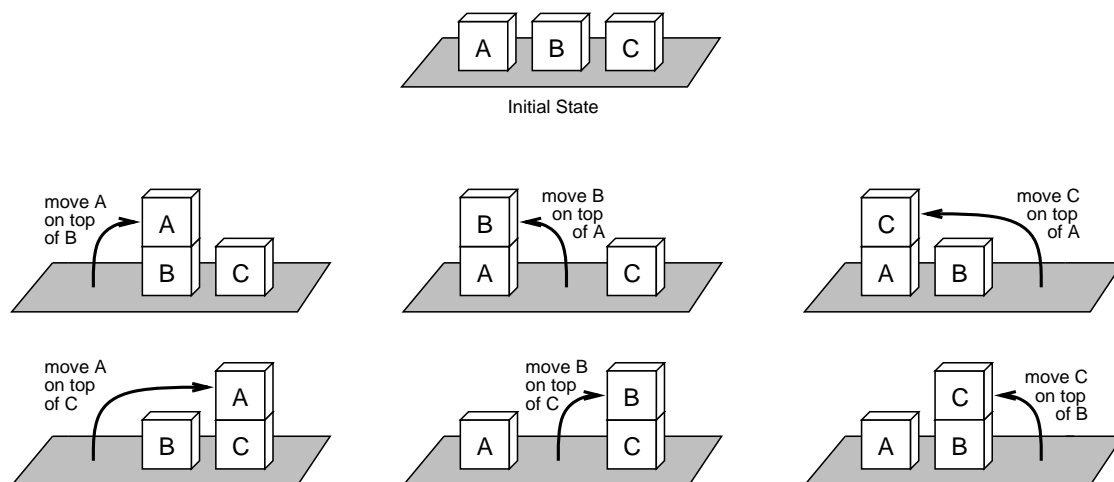
Figure 2.5: The six operators proposed for the initial state of the blocks world each move one block to a new location.

2.1.4   Proposing candidate operators

As a first step in selecting an operator, one or more candidate operators are proposed. Operators are proposed by rules that test features of the current state. When the blocks-world task is run, the Soar program will propose six distinct (but similar) operators for the initial state as illustrated in Figure 2.5. These operators correspond to the six different actions that are possible given the initial state.

2.1.5   Comparing candidate operators: Preferences

The second step Soar takes in selecting an operator is to evaluate or compare the candidate operators. In Soar, this is done via rules that test the proposed operators and the current state, and then create *preferences*. Preferences assert the relative or absolute merits of the candidate operators. For example, a preference may say that operator A is a "better" choice than operator B at this particular time, or a preference may say that operator A is the "best" thing to do at this particular time.

2.1.6   Selecting a single operator

Soar attempts to select a single operator based on the preferences available for the candidate operators. There are four different situations that may arise:

1. The available preferences unambiguously prefer a single operator.

2. The available preferences suggest multiple operators, and prefer a subset that can be selected from randomly.

3. The available preferences suggest multiple operators, but neither case 1 or 2 above hold.

4. The available preferences do not suggest any operators.

In the first case, the preferred operator is selected. In the second case, one of the subset is selected randomly. In the third and fourth cases, Soar has reached an "impasse" in problem solving, and a new substate is created. Impasses are discussed in Section 2.6.

In our blocks-world example, the second case holds, and Soar can select one of the operators randomly.

2.1.7 Applying the operator

An operator applies by making changes to the state; the specific changes that are appropriate depend on the operator and the current state.

There are two primary approaches to modifying the state: indirect and direct. *Indirect* changes are used in Soar programs that interact with an external environment: The Soar program sends motor commands to the external environment and monitors the external environment for changes. The changes are reflected in an updated state description, garnered from sensors. Soar may also make *direct* changes to the state; these correspond to Soar doing problem solving "in its head". Soar programs that do not interact with an external environment can make only direct changes to the state.

Internal and external problem solving should not be viewed as mutually exclusive activities in Soar. Soar programs that interact with an external environment will generally have operators that make direct and indirect changes to the state: The motor command is represented as substructure of the state *and* it is a command to the environment. Also, a Soar program may maintain an internal model of how it expects an external operator will modify the world; if so, the operator must update the internal model (which is substructure of the state).

When Soar is doing internal problem solving, it must know how to modify the state descriptions appropriately when an operator is being applied. If it is solving the problem in an external environment, it must know what possible motor commands it can issue in order to affect its environment.

The example blocks-world task described here does not interact with an external environment. Therefore, the Soar program directly makes changes to the state when operators are applied. There are four changes that may need to be made when a block is moved in our task:

1. The block that is being moved is no longer where it was (it is no longer "on top" of the same thing).

2. The block that is being moved is now in a new location (it is "on top" of a new thing).

3. The place that the block used to be is now clear.

4. The place that the block is moving to is no longer clear — unless it is the table, which is always considered "clear"[1].

---

[1]In this blocks-world task, the table always has room for another block, so it is represented as always being "clear".

The blocks-world task could also be implemented using an external simulator. In this case, the Soar program does not update all the "on top" and "clear" relations; the updated state description comes from the simulator.

2.1.8   Making inferences about the state

Making monotonic inferences about the state is the other role that Soar long-term knowledge may fulfill. Such elaboration knowledge can simplify the encoding of operators because entailments of a set of core features of a state do not have to be explicitly included in application of the operator. In Soar, these inferences will be automatically retracted when the situation changes such that the inference no longer holds.

For instance, our example blocks-world task uses an elaboration to keep track of whether or not a block is "clear". The elaboration tests for the absence of a block that is "on top" of a particular block; if there is no such "on top", the block is "clear". When an operator application creates a new "on top", the corresponding elaboration retracts, and the block is no longer "clear".

2.1.9   Problem Spaces

If we were to construct a Soar system that worked on a large number of different types of problems, we would need to include large numbers of operators in our Soar program. For a specific problem and a particular stage in problem solving, only a subset of all possible operators are actually relevant. For example, if our goal is to *count* the blocks on the table, operators having to do with moving blocks are probably not important, although they may still be "legal". The operators that are relevant to current problem-solving activity define the space of possible states that might be considered in solving a problem, that is, they define the *problem space*.

Soar programs are implicitly organized in terms of problem spaces because the conditions for proposing operators will restrict an operator to be considered only when it is relevant. The complete problem space for the blocks world is show in Figure 2.6. Typically, when Soar solves a problem in this problem space, it does not explicitly generate all of the states, examine them, and then create a path. Instead, Soar is *in* a specific state at a given time (represented in working memory), attempting to select an operator that will move it to a new state. It uses whatever knowledge it has about selecting operators given the current situation, and if its knowledge is sufficient, it will move toward its goal. The same problem could be recast in Soar as a planning problem, where the goal is to develop a plan to solve the problem, instead of just solving the problem. In that case, a state in Soar would consist of a plan, which in turn would have representations of Blocks World states and operators from the original space. The operators would perform editing operations on the plan, such as adding new Blocks World operators, simulating those operators, etc. In both formulations of the problem, Soar is still applying operators to generate new states, it is just that the states and operators have different content.
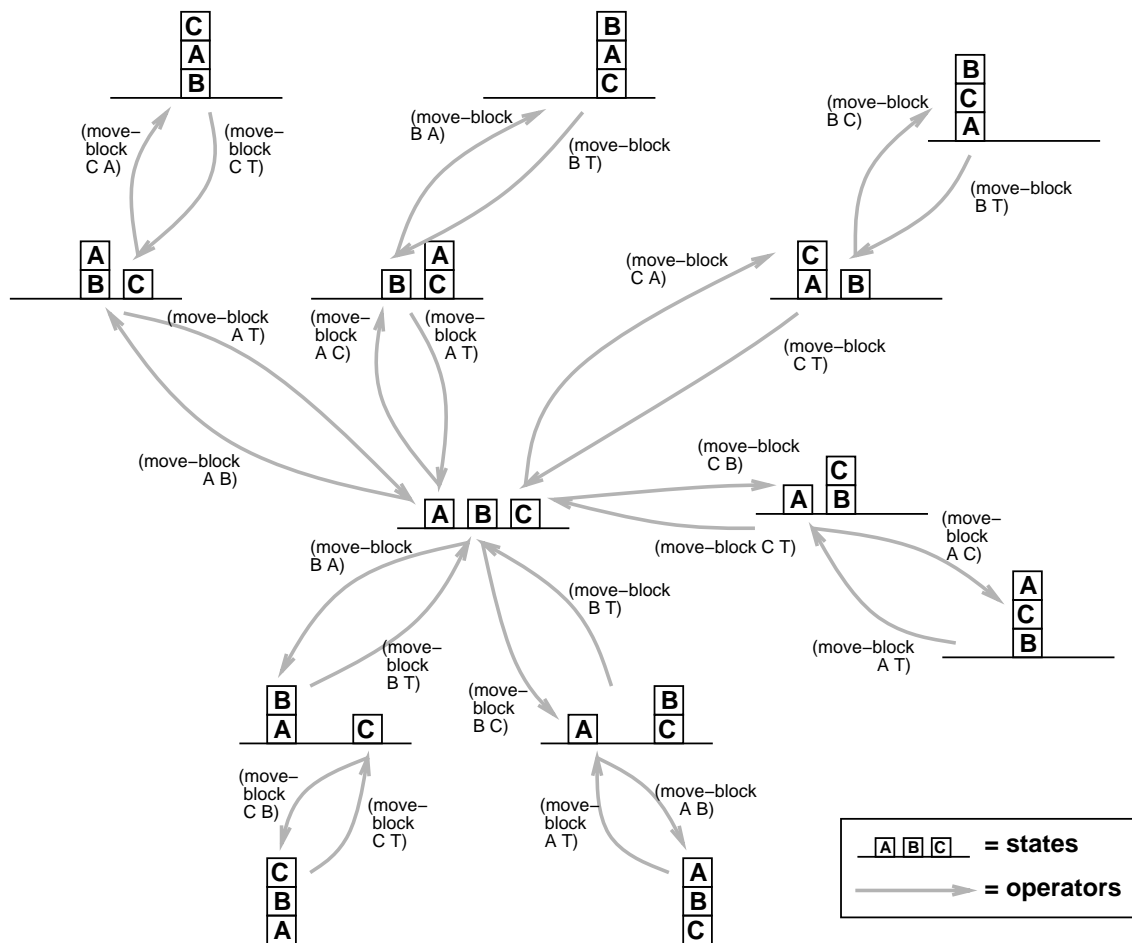
Figure 2.6: The problem space in the blocks-world includes all operators that move blocks from one location to another and all possible configurations of the three blocks.

The remaining sections in this chapter describe the memories and processes of Soar: working memory, production memory, preference memory, Soar's execution cycle (the decision procedure), learning, and how input and output fit in.

2.2 Working memory: The Current Situation

Soar represents the current problem-solving situation in its *working memory*. Thus, working memory holds the current state and operator and is Soar's "short-term" knowledge, reflecting the current knowledge of the world and the status in problem solving.

Working memory contains elements called working memory elements, or WME's for short. Each WME contains a very specific piece of information; for example, a WME might say that "B1 is a block". Several WME's collectively may provide more information about the same *object*, for example, "B1 is a block", "B1 is named A", "B1 is on the table", etc. These WME's are related because they are all contributing to the description of something that is internally known to Soar as "B1". B1 is called an *identifier*; the group of WME's that

share this identifier are referred to as an *object* in working memory. Each WME describes a different *attribute* of the object, for example, its name or type or location; each *attribute* has a *value* associated with it, for example, the name is A, the type is block, and the position is on the table. Therefore, each WME is an identifier-attribute-value triple, and all WME's with the same identifier are part of the same object.

Objects in working memory are *linked* to other objects: The value of one WME may be an identifier of another object. For example, a WME might say that "B1 is ontop of T1", and another collection of WME's might describe the object T1: "T1 is a table", "T1 is brown", and "T1 is ontop of F1". And still another collection of WME's might describe the object F1: "F1 is a floor", etc. All objects in working memory must be linked to a state, either directly or indirectly (through other objects). Objects that are not linked to a state will be automatically removed from working memory by the Soar architecture.

WME's are also often called *augmentations* because they "augment" the object, providing more detail about it. While these two terms are somewhat redundant, WME is a term that is used more often to refer to the contents of working memory, while augmentation is a term that is used more often to refer to the description of an object. Working memory is illustrated at an abstract level in Figure 2.3 on page 8.

The attribute of an augmentation is usually a constant, such as `name` or `type`, because in a sense, the attribute is just a label used to distinguish one link in working memory from another.[2]

The value of an augmentation may be either a constant, such as `red`, or an identifier, such as `06`. When the value is an identifier, it refers to an object in working memory that may have additional substructure. In semantic net terms, if a value is a constant, then it is a terminal node with no links; if it is an identifier it is a nonterminal node.

One key concept of Soar is that working memory is a set, which means that there can never be two elements in working memory at the same time that have the same identifier-attribute-value triple (this is prevented by the architecture). However, it is possible to have multiple working memory elements that have the same identifier and attribute, but that each have different values. When this happens, we say the attribute is a *multi-valued attribute*, which is often shortened to be *multi-attribute*.

An object is defined by its augmentations and *not* by its identifier. An identifier is simply a label or pointer to the object. On subsequent runs of the same Soar program, there may be an object with exactly the same augmentations, but a different identifier, and the program will still reason about the object appropriately. Identifiers are internal markers for Soar; they can appear in working memory, but they never appear in a production.

There is no predefined relationship between objects in working memory and "real objects" in the outside world. Objects in working memory may refer to real objects, such as `block A`; features of an object, such as the color `red` or shape `cube`; a relation between objects, such as `ontop`; classes of objects, such as `blocks`; etc. The actual names of attributes and

---

[2]In order to allow these links to have some substructure, the attribute name may be an identifier, which means that the attribute may itself have attributes and values, as specified by additional working memory elements.

values have no meaning to the Soar architecture (aside from a few WME's created by the architecture itself). For example, Soar doesn't care whether the things in the blocks world are called "blocks" or "cubes" or "chandeliers". It is up to the Soar programmer to pick suitable labels and to use them consistently.

The elements in working memory arise from one of four sources:

1. The actions of productions create most working memory elements.
2. The decision procedure automatically creates some special state augmentations (type, superstate, impasse, ...) whenever a state is created. States are created during initialization (the first state) or because of an impasse (a substate).
3. The decision procedure creates the operator augmentation of the state based on preferences. This records the selection of the current operator.
4. External I/O systems create working memory elements on the input-link for sensory data.

The elements in working memory are removed in six different ways:

1. The decision procedure automatically removes all state augmentations it creates when the impasse that led to their creation is resolved.
2. The decision procedure removes the operator augmentation of the state when that operator is no longer selected as the current operator.
3. Production actions that use `reject` preferences remove working memory elements that were created by other productions.
4. The architecture automatically removes i-supported WMEs when the productions that created them no longer match.
5. The I/O system removes sensory data from the input-link when it is no longer valid.
6. The architecture automatically removes WME's that are no longer linked to a state (because some other WME has been removed).

For the most part, the user is free to use any attributes and values that are appropriate for the task. However, states have special augmentations that cannot be directly created, removed, or modified by rules. These include the augmentations created when a state is created, and the state's operator augmentation that signifies the current operator (and is created based on preferences). The specific attributes that the Soar architecture automatically creates are listed in Section 3.4. Productions may create any other attributes for states.

Preferences are held in a separate *preference memory* where they cannot be tested by productions; however, `acceptable` preferences are held in *both* preference memory and in working memory. By making the `acceptable` preferences available in working memory, the acceptable preferences can be tested for in productions allowing the candidates operators to be compared before they are selected.

An Abstract View of Production Memory
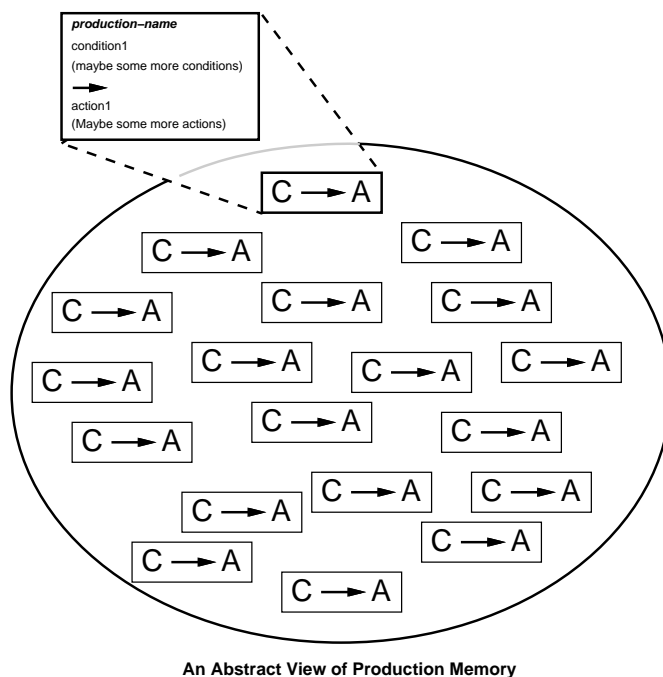
Figure 2.7: An abstract view of production memory. The productions are not related to one another.

2.3 Production Memory: Long-term Knowledge

Soar represents long-term knowledge as *productions* that are stored in *production memory*, illustrated in Figure 2.7. Each production has a set of conditions and a set of actions. If the conditions of a production match working memory, the production *fires*, and the actions are performed.

2.3.1 The structure of a production

In the simplest form of a production, conditions and actions refer directly to the presence (or absence) of objects in working memory. For example, a production might say:

```
CONDITIONS: block A is clear
            block B is clear
ACTIONS:    suggest an operator to move block A ontop of block B
```

This is not the literal syntax of productions, but a simplification. The actual syntax is presented in Chapter 3.

The conditions of a production may also specify the *absence* of patterns in working memory. For example, the conditions could also specify that "block A is not red" or "there are no red blocks on the table". But since these are not needed for our example production, there are no examples of negated conditions for now.

The order of the conditions of a production do not matter to Soar except that the first condition must directly test the state. Internally, Soar will reorder the conditions so that the matching process can be more efficient. This is a mechanical detail that need not concern most users. However, you may print your productions to the screen or save them in a file; if they are not in the order that you expected them to be, it is likely that the conditions have been reordered by Soar.

2.3.1.1   Variables in productions and multiple instantiations

In the example production above, the names of the blocks are "hardcoded", that is, they are named specifically. In Soar productions, variables are used so that a production can apply to a wider range of situations.

The variables are bound to specific symbols in working memory elements by Soar's matching process. A production along with a specific and consistent set of variable bindings is called an *instantiation*. A production instantiation is consistent only if every occurrence of a variable is bound to the same value. Since the same production may match multiple times, each with different variable bindings, several instantiations of the same production may match at the same time and, therefore, fire at the same time. If blocks A and B are clear, the first production (without variables) will suggest one operator. However, if a production was created that used variables to test the names, this second production will be instantiated twice and therefore suggest *two* operators: one operator to move block A ontop of block B and a second operator to move block B ontop of block A.

Because the identifiers of objects are determined at runtime, literal identifiers cannot appear in productions. Since identifiers occur in every working memory element, variables must be used to test for identifiers, and using the same variables across multiple occurrences is what links conditions together.

Just as the elements of working memory must be linked to a state in working memory, so must the objects referred to in a production's conditions. That is, one condition must test a state object *and* all other conditions must test that same state or objects that are linked to that state.

2.3.2   Architectural roles of productions

Soar productions can fulfill four different roles: the three knowledge-retrieval problem-solving functions, and the state elaboration function, all described on page 6:

1. Operator proposal
2. Operator comparison
3. *(Operator selection is not an act of knowledge retrieval)*
4. Operator application
5. State elaboration

A single production should not fulfill more than one of these roles (except for proposing an operator and creating an absolute preference for it). Although productions are not declared to be of one type or the other, Soar examines the structure of each production and classifies the rules automatically based on whether they propose and compare operators, apply operators, or elaborate the state.

2.3.3   Production Actions and Persistence

Generally, actions of a production either create preferences for operator selection, or create/remove working memory elements. For operator proposal and comparison, a production creates preferences for operator selection. These preferences should persist only as long as the production instantiation that created them continues to match. When the production instantiation no longer matches, the situation has changed, making the preference no longer relevant. Soar automatically removes the preferences in such cases. These preferences are said to have *I-support* (for "instantiation support"). Similarly, state elaborations are simple inferences that are valid only so long as the production matches. Working memory elements created as state elaborations also have I-support and remain in working memory only as long as the production instantiation that created them continues to match working memory. For example, the set of relevant operators changes as the state changes, thus the proposal of operators is done with I-supported preferences. This way, the operator proposals will be retracted when they no longer apply to the current situation.

However, the actions of productions that *apply* an operator, either by adding or removing elements from working memory, need to persist even after the operator is no longer selected and operator application production instantiation no longer matches. For example, in placing a block on another block, a condition is that the second block be clear. However, the action of placing the first block removes the fact that the second block is clear, so the condition will no longer be satisfied.

Thus, operator application productions do not retract their actions, even if they no longer match working memory. This is called *O-support* (for "operator support"). Working memory elements that participate in the application of operators are maintained throughout the existence of the state in which the operator is applied, unless explicitly removed (or if they become unlinked). Working memory elements are removed by a *reject* action of a operator-application rule.

Whether a working memory element receives O-support or I-support is determined by the structure of the production instantiation that creates the working memory element. O-support is given only to working memory elements created by operator-application productions.

An operator-application production tests the current operator of a state and modifies the state. Thus, a working memory element receives O-support if it is for an augmentation of the current state or substructure of the state, and the conditions of the instantiation that created it test augmentations of the current operator.

When productions are matched, all productions that have their conditions met fire creating

or removing working memory elements. Also, working memory elements and preferences that lose I-support are removed from working memory. Thus, several new working memory elements and preferences may be created, and several existing working memory elements and preferences may be removed at the same time. (Of course, all this doesn't happen literally at the same time, but the order of firings and retractions is unimportant, and happens in parallel from a functional perspective.)

2.4 Preference memory: Selection Knowledge

The selection of the current operator is determined by the *preferences* in *preference memory*. Preferences are suggestions or imperatives about the current operator, or information about how suggested operators compare to other operators. Preferences refer to operators by using the identifier of a working memory element that stands for the operator. After preferences have been created for a state, the decision procedures evaluates them to select the current operator for that state.

For an operator to be selected, there will be at least one preference for it, specifically, a preference to say that the value is a candidate for the operator attribute of a state (this is done with either an "`acceptable`" or "`require`" preference). There may also be others, for example to say that the value is "best".

The different preferences available and the semantics of preferences are explained in Section 2.4.1. Preferences remain in preference memory until removed for one of the reasons previously discussed in Section 2.3.3.

2.4.1 Preference semantics

This section describes the semantics of each type of preference. More details on the preference resolution process are provided in Appendix D.

Only a single value can be selected as the current operator, that is, all values are mutually exclusive. In addition, there is no implicit transitivity in the semantics of preferences. If A is indifferent to B, and B is indifferent to C, A and C will not be indifferent to one another unless there is a preference that A is indifferent to C (or C and A are both indifferent to all competing values).

Acceptable (+) An `acceptable` preference states that a value is a candidate for selection. All values, except those with `require` preferences, must have an `acceptable` preference in order to be selected. If there is only one value with an `acceptable` preference (and none with a `require` preference), that value will be selected as long as it does not also have a `reject` or a `prohibit` preference.

Reject (−) A `reject` preference states that the value is not a candidate for selection.

Better (>), Worse (<) A `better` or `worse` preference states, for the two values involved, that one value should not be selected if the other value is a candidate. `Better` and

worse allow for the creation of a partial ordering between candidate values. `Better` and `worse` are simple inverses of each other, so that A better than B is equivalent to B worse than A.

Best (>) A `best` preference states that the value may be better than any competing value (unless there are other competing values that are also "best"). If a value is `best` (and not `reject`ed, `prohibit`ed, or `worse` than another), it will be selected over any other value that is not also `best` (or `require`d). If two such values are `best`, then any remaining preferences for those candidates (`worst`, `indifferent`) will be examined to determine the selection. Note that if a value (that is not `reject`ed or `prohibit`ed) is `better` than a `best` value, the `better` value will be selected. (This result is counterintuitive, but allows explicit knowledge about the relative worth of two values to dominate knowledge of only a single value. A `require` preference should be used when a value *must* be selected for the goal to be achieved.)

Worst (<) A `worst` preference states that the value should be selected only if there are no alternatives. It allows for a simple type of default specification. The semantics of the `worst` preference are similar to those for the `best` preference.

Indifferent (=) An `indifferent` preference states that there is positive knowledge that it does not matter which value is selected. This may be a binary preference, to say that two values are mutually indifferent, or a unary preference, to say that a single value is as good or as bad a choice as other expected alternatives.

When `indifferent` preferences are used to signal that it does not matter which operator is selected, by default, Soar chooses randomly from among the alternatives. (The `indifferent-selection` function can be used to change this behavior as described on page 164 in Chapter 8.)

Numeric-Indifferent (= *number***)** A `numeric-indifferent` preference is used to bias the random selection from mutually indifferent values. This preference includes a unary indifferent preference, so an operator with a `numeric-indifferent` preference will not force a tie impasse. When a set of operators are determined to be indifferent based on all other asserted preference types and at least one operator has a numeric-indifferent preference, the decision mechanism will choose an operator based on their numeric-indifferent values and the exploration policy. The available exploration policies and how they calculate selection probability are detailed in the documentation for the `indifferent-selection` command on page 164. When a single operator is given multiple numeric-indifferent preferences, they are either averaged or summed into a single value based on the setting of the `numeric-indifferent-mode` command (see page 173).

Numeric-indifferent preferences that are created by RL rules can be adjusted by the reinforcement learning mechanism. In this way, it's possible for an agent to begin a task with only arbitrarily initialized numeric indifferent preferences and with experience learn to make the optimal decisions. See chapter 5 for more information.

Require (!) A `require` preference states that the value must be selected if the goal is to be achieved.

Prohibit (∼) A `prohibit` preference states that the value cannot be selected if the goal is to be achieved. If a value has a `prohibit` preference, it will not be selected for a value of an augmentation, independent of the other preferences.

If there is an `acceptable` preference for a value of an operator, and there are no other competing values, that operator will be selected. If there are multiple `acceptable` preferences for the same state but with different values, the preferences must be evaluated to determine which candidate is selected.

If the preferences can be evaluated without conflict, the appropriate operator augmentation of the state will be added to working memory. This can happen when they all suggest the same operator or when one operator is preferable to the others that have been suggested. When the preferences conflict, Soar reaches an impasse, as described in Section 2.6.

Preferences can be confusing; for example, there can be two suggested values that are both "best" (which again will lead to an impasse unless additional preferences resolve this conflict); or there may be one preference to say that value `A` is better than value `B` and a second preference to say that value `B` is better than value `A`.

2.5 Soar's Execution Cycle: Without Substates

The execution of a Soar program proceeds through a number of *cycles*. Each cycle has five phases:

1. Input: New sensory data comes into working memory.

2. Proposal: Productions fire (and retract) to interpret new data (state elaboration), propose operators for the current situation (operator proposal), and compare proposed operators (operator comparison). All of the actions of these productions are I-supported. All matched productions fire in parallel (and all retractions occur in parallel), and matching and firing continues until there are no more additional complete matches or retractions of productions (*quiescence*).

3. Decision: A new operator is selected, or an impasse is detected and a new state is created.

4. Application: Productions fire to apply the operator (operator application). The actions of these productions will be O-supported. Because of changes from operator application productions, other productions with I-supported actions may also match or retract. Just as during proposal, productions fire and retract in parallel until quiescence.

5. Output: Output commands are sent to the external environment.

The cycles continue until the halt action is issued from the Soar program (as the action of a production) or until Soar is interrupted by the user.
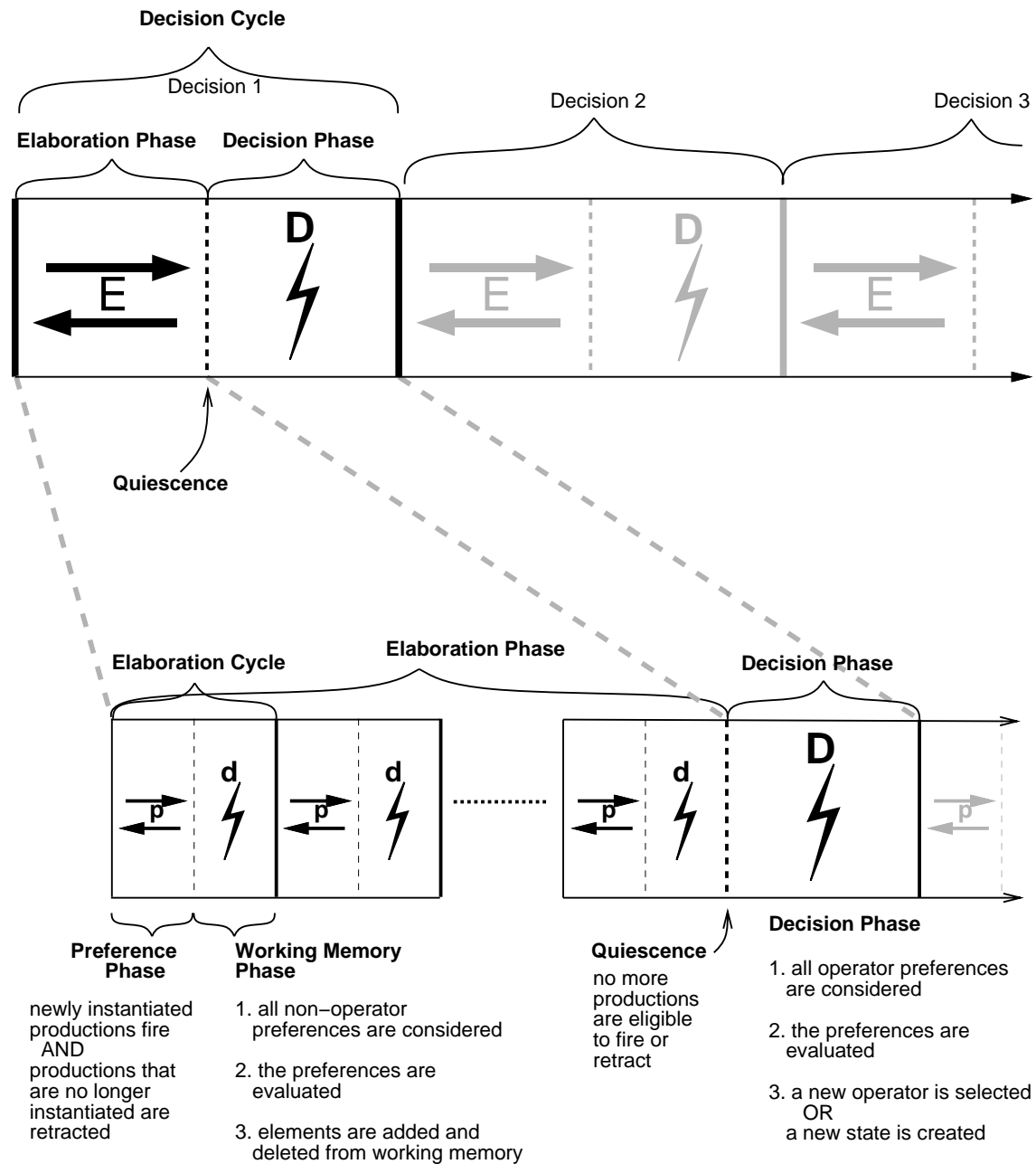
Figure 2.8: A detailed illustration of Soar's decision cycle: out of date

During the processing of these phases, it is possible that the preferences that resulted in the selection of the current operator could change. Whenever operator preferences change, the preferences are re-evaluated and if a different operator selection would be made, then the current operator augmentation of the state is immediately removed. However, a new operator is not selected until the next decision phase, when all knowledge has had a chance to be retrieved.

```
Soar
  while (HALT not true) Cycle;

Cycle
  InputPhase;
  ProposalPhase;
  DecisionPhase;
  ApplicationPhase;
  OutputPhase;


ProposalPhase
  while (some I-supported productions are waiting to fire or retract)
    FireNewlyMatchedProductions;
    RetractNewlyUnmatchedProductions;

DecisionPhase
  for (each state in the stack,
       starting with the top-level state)
  until (a new decision is reached)
    EvaluateOperatorPreferences; /* for the state being considered */
    if (one operator preferred after preference evaluation)
      SelectNewOperator;
    else                   /* could be no operator available or */
      CreateNewSubstate;   /* unable to decide between more than one */

ApplicationPhase
  while (some productions are waiting to fire or retract)
    FireNewlyMatchedProductions;
    RetractNewlyUnmatchedProductions;
```

Figure 2.9: A simplified version of the Soar algorithm.

2.6 Impasses and Substates

When the decision procedure is applied to evaluate preferences and determine the operator augmentation of the state, it is possible that the preferences are either incomplete or inconsistent. The preferences can be incomplete in that no `acceptable` operators are suggested, or that there are insufficient preferences to distinguish among `acceptable` operators. The preferences can be inconsistent if, for instance, operator `A` is preferred to operator `B`, and operator `B` is preferred to operator `A`. Since preferences are generated independently, from different production instantiations, there is no guarantee that they will be consistent.

2.6.1   Impasse Types

There are four types of impasses that can arise from the preference scheme.

Tie impasse — A *tie* impasse arises if the preferences do not distinguish between two or more operators with `acceptable` preferences. If two operators both have `best` or `worst` preferences, they will tie unless additional preferences distinguish between them.

Conflict impasse — A *conflict* impasse arises if at least two values have conflicting better or worse preferences (such as `A` is better than `B` and `B` is better than `A`) for an operator, and neither one is rejected, prohibited, or `required`.

Constraint-failure impasse — A *constraint-failure* impasse arises if there is more than one `required` value for an operator, or if a value has both a `require` and a `prohibit` preference. These preferences represent constraints on the legal selections that can be made for a decision and if they conflict, no progress can be made from the current situation and the impasse cannot be resolved by additional preferences.

No-change impasse — A *no-change* impasse arises if a new operator is not selected during the decision procedure. There are two types of no-change impasses: state no-change and operator no-change:

    State no-change impasse — A state no-change impasse occurs when there are no `acceptable` (or `require`) preferences to suggest operators for the current state (or all the `acceptable` values have also been `reject`ed). The decision procedure cannot select a new operator.

    Operator no-change impasse — An operator no-change impasse occurs when either a new operator is selected for the current state but no additional productions match during the application phase, or a new operator is not selected during the next decision phase.

There can be only one type of impasse at a given level of subgoaling at a time. Given the semantics of the preferences, it is possible to have a tie or conflict impasse and a constraint-failure impasse at the same time. In these cases, Soar detects only the constraint-failure impasse.

The impasse is detected *during* the selection of the operator, but happens *because* one of the other four problem-solving functions was incomplete.

2.6.2   Creating New States

Soar handles these inconsistencies by creating a new state in which the goal of the problem solving is to resolve the impasse. Thus, in the substate, operators will be selected and applied in an attempt either to discover which of the tied operators should be selected, or to apply the selected operator piece by piece. The substate is often called a *subgoal* because it exists to resolve the impasse, but is sometimes called a substate because the representation of the subgoal in Soar is as a state.

The initial state in the subgoal contains a complete description of the cause of the impasse, such as the operators that could not be decided among (or that there were no operators

proposed) and the state that the impasse arose in. From the perspective of the new state, the latter is called the *superstate*. Thus, the superstate is part of the substructure of each state, represented by the Soar architecture using the `superstate` attribute. (The initial state, created in the 0th decision cycle, contains a `superstate` attribute with the value of `nil` — the top-level state has no superstate.)

The knowledge to resolve the impasse may be retrieved by any type of problem solving, from searching to discover the implications of different decisions, to asking an outside agent for advice. There is no *a priori* restriction on the processing, except that it involves applying operators to states.

In the substate, operators can be selected and applied as Soar attempts to solve the subgoal. (The operators proposed for solving the subgoal may be similar to the operators in the superstate, or they may be entirely different.) While problem solving in the subgoal, additional impasses may be encountered, leading to new subgoals. Thus, it is possible for Soar to have a *stack* of subgoals, represented as states: Each state has a single superstate (except the initial state) and each state may have at most one substate. Newly created subgoals are considered to be added to the bottom of the stack; the first state is therefore called the *top-level state*.[3] See Figure 2.10 for a simplified illustrations of a subgoal stack.

Soar continually attempts to retrieve knowledge relevant to all goals in the subgoal stack, although problem-solving activity will tend to focus on the most recently created state. However, problem solving is active at all levels, and productions that match at any level will fire.

2.6.3 Results

In order to resolve impasses, subgoals must generate results that allow the problem solving at higher levels to proceed. The *results* of a subgoal are the working memory elements and preferences that were created in the substate, and that are also linked directly or indirectly to a superstate (*any* superstate in the stack). A preference or working memory element is said to be created in a state if the production that created it tested that state and this is the most recent state that the production tested. Thus, if a production tests multiple states, the preferences and working memory elements in its actions are considered to be created in the most recent of those states (and is not considered to have been created in the other states). The architecture automatically detects if a preference or working memory element created in a substate is also linked to a superstate.

These working memory elements and preferences will not be removed when the impasse is resolved because they are still linked to a superstate, and therefore, they are called the *results of the subgoal*. A result has either I-support or O-support; the determination of support is described below.

A working memory element or preference will be a result if its identifier is already linked to a superstate. A working memory element or preference can also become a result indirectly

---

[3]The original state is the "top" of the stack because as Soar runs, this state (created first), will be at the top of the computer screen, and substates will appear on the screen below the top-level state.

Figure 2.10: A simplified illustration of a subgoal stack.

if, after it is created and it is still in working memory or preference memory, its identifier becomes linked to a superstate through the creation of another result. For example, if the problem solving in a state constructs an operator for a superstate, it may wait until the operator structure is complete before creating an `acceptable` preference for the operator in the superstate. The `acceptable` preference is a result because it was created in the state and is linked to the superstate (and, through the superstate, is linked to the top-level state). The substructures of the operator then become results because the operator's identifier is now linked to the superstate.

Justifications: Determination of support for results

Some results receive I-support, while others receive O-support. The type of support received by a result is determined by the function it plays in the superstate, and not the function it played in the state in which it was created. For example, a result might be created through operator application in the state that created it; however, it might only be a state elaboration in the superstate. The first function would lead to O-support, but the second would lead to I-support.

In order for the architecture to determine whether a result receives I-support or O-support, Soar must first determine the function that the working memory element or preference plays (that is, whether the result should be considered an operator application or not). To do this, Soar creates a temporary production, called a *justification*. The justification summarizes the processing in the substate that led to the result:

The conditions of a justification are those working memory elements that exist in the superstate (and above) that were necessary for producing the result. This is determined by collecting all of the working memory elements tested by the productions that fired in the subgoal that led to the creation of the result, and then removing those conditions that test working memory elements created in the subgoal.

The action of the justification is the result of the subgoal.

Soar determines I-support or O-support for the justification just as it would for any other production, as described in Section 2.3.3. If the justification is an operator application, the result will receive O-support. Otherwise, the result gets I-support from the justification. If such a result loses I-support from the justification, it will be retracted if there is no other support. Justification are not added to production memory, but are otherwise treated as an instantiated productions that have already fired.

Justifications include any negated conditions that were in the original productions that participated in producing the results, and that test for the absence of superstate working memory elements. Negated conditions that test for the absence of working memory elements that are local to the substate are not included, which can lead to overgeneralization in the justification (see Section 4.7 on page 80 for details).

Justifications can also include operator evaluation knowledge that led to the selection of the operator that produced the result. For example, the conditions of any production that

creates a prohibit preference for an operator in the substate that was not selected will be backtraced through and may produce additional conditions in the justification. Moreover, if the add-desirability-prefs learn setting is on, conditions from other preference types (better, best, worse, worst indifferent) can be included as well. For a more complete description of how Soar chooses which desirability preferences to include, see Section 4.3 on page 77.

2.6.4   Removal of Substates: Impasse Resolution

Problem solving in substates is an important part of what Soar *does*, and an operator impasse does not necessarily indicate a problem in the Soar program. They are a way to decompose a complex problem into smaller parts and they provide a context for a program to deliberate about which operator to select. Operator impasses are necessary, for example, for Soar to do any learning about problem solving (as will be discussed in Chapter 4). This section describes how impasses may be resolved during the execution of a Soar program, how they may be eliminated during execution without being resolved, and some tips on how to modify a Soar program to prevent a specific impasse from occurring in the first place.

Resolving Impasses

An impasse is *resolved* when processing in a subgoal creates results that lead to the selection of a new operator for the state where the impasse arose. When an operator impasse is resolved, Soar has an opportunity to learn, and the substate (and all its substructure) is removed from working memory.

Here are possible approaches for resolving specific types of impasses are listed below:

Tie impasse — A tie impasse can be resolved by productions that create preferences that prefer one option (`better`, `best`, `require`), eliminate alternatives (`worse`, `worst`, `reject`, `prohibit`), or make all of the objects indifferent (`indifferent`).

Conflict impasse — A conflict impasse can be resolved by productions that create preferences to `require` one option (`require`), or eliminate the alternatives (reject, prohibit).

Constraint-failure impasse — A constraint-failure impasse cannot be resolved by additional preferences, but may be prevented by changing productions so that they create fewer `require` or `prohibit` preferences.

State no-change impasse — A state no-change impasse can be resolved by productions that create `acceptable` or `require` preferences for operators.

Operator no-change impasse — An operator no-change impasse can be resolved by productions that apply the operator, changing the state so the operator proposal no longer matches or other operators are proposed and preferred.

Eliminating Impasses

An impasse is resolved when results are created that allow progress to be made in the state where the impasse arose. In Soar, an impasse can be *eliminated* (but not resolved)

when a higher level impasse is resolved, eliminated, or regenerated. In these cases, the impasse becomes irrelevant because higher-level processing can proceed. An impasse can also become irrelevant if input from the outside world changes working memory which in turn causes productions to fire that make it possible to select an operator. In all these cases, the impasse is eliminated, but not "resolved", and Soar does not learn in this situation.

Regenerating Impasses

An impasse is *regenerated* when the problem solving in the subgoal becomes *inconsistent* with the current situation. During problem solving in a subgoal, Soar monitors which aspect of the surrounding situation (the working memory elements that exist in superstates) the problem solving in the subgoal has depended upon. If those aspects of the surronding situation change, either because of changes in input or because of results, the problem solving in the subgoal is inconsistent, and the state created in response to the original impasse is removed and a new state is created. Problem solving will now continue from this new state. The impasse is not "resolved", and Soar does not learn in this situation.

The reason for regeneration is to guarantee that the working memory elements and preferences created in a substate are consistent with higher level states. As stated above, inconsistency can arise when a higher level state changes either as a result of changes in what is sensed in the external environment, or from results produced in the subgoal. The problem with inconsistency is that once inconsistency arises, the problem being solved in the subgoal may no longer be the problem that actually needs to be solved. Luckily, not all changes to a superstate lead to inconsistency.

In order to detect inconsistencies, Soar maintains a *dependency set* for every subgoal/substate. The dependency set consists of all working memory elements that were tested in the conditions of productions that created O-supported working memory elements that are directly or indirectly linked to the substate. Thus, whenever such an O-supported working memory element is created, Soar records which working memory elements that exist in a superstate were tested, directly or indirectly in creating that working memory element. dependency-set Whenever any of the working memory elements in the dependency set of a substate change, the substate is regenerated.

Note that the creation of I-supported structures in a subgoal does not increase the dependency set, nor do O-supported results. Thus, only subgoals that involve the creation of internal O-support working memory elements risk regeneration, and then only when the basis for the creation of those elements changes.

Substate Removal

Whenever a substate is removed, all working memory elements and preferences that were created in the substate that are not results are removed from working memory. In Figure 2.10, state `S3` will be removed from working memory when the impasse that created it is resolved, that is, when sufficient preferences have been generated so that one of the operators

for state `S2` can be selected. When state `S3` is removed, operator `O9` will also be removed, as will the acceptable preferences for `O7`, `O8`, and `O9`, and the `impasse`, `attribute`, and `choices` augmentations of state `S3`. These working memory elements are removed because they are no longer linked to the subgoal stack. The acceptable preferences for operators `O4`, `O5`, and `O6` remain in working memory. They were linked to state `S3`, but since they are also linked to state `S2`, they will stay in working memory until `S2` is removed (or until they are retracted or rejected).

2.6.5   Soar's Cycle: With Substates

When there are multiple substates, Soar's cycle remains basically the same but has a few minor changes.

The first change is that during the decision procedure, Soar will detect impasses and create new substates. For example, following the proposal phase, the decision phase will detect if a decision cannot be made given the current preferences. If an impasse arises, a new substate is created and added to working memory.

The second change when there are multiple substates is that at each phase, Soar goes through the substates, from oldest (highest) to newest (lowest), completing any necessary processing at that level for that phase before doing any processing in the next substate. When firing productions for the proposal or application phases, Soar processes the firing (and retraction) of rules, starting from those matching the oldest substate to the newest. Whenever a production fires or retracts, changes are made to working memory and preference memory, possibly changing which productions will match at the lower levels (productions firing within a given level are fired in parallel – simulated). Productions firings at higher levels can resolve impasses and thus eliminate lower states before the productions at the lower level ever fire. Thus, whenever a level in the state stack is reached, all production activity is guaranteed to be consistent with any processing that has occurred at higher levels.

2.7   Learning

When an operator impasse is resolved, it means that Soar has, through problem solving, gained access to knowledge that was not readily available before. Therefore, when an impasse is resolved, Soar has an opportunity to learn, by summarizing and generalizing the processing in the substate.

One of Soar's learning mechanisms is called *chunking*; it attempts to create a new production, called a chunk. The conditions of the chunk are the elements of the state that (through some chain of production firings) allowed the impasse to be resolved; the action of the production is the working memory element or preference that resolved the impasse (the result of the impasse). The conditions and action are variablized so that this new production may match in a similar situation in the future and prevent an impasse from arising.

Chunks are very similar to justifications in that they are both formed via the backtrac-

ing process and both create a result in their actions. However, there are some important distinctions:

1. Chunks are productions and are added to production memory. Justifications do not appear in production memory.
2. Justifications disappear as soon as the working memory element or preference they provide support for is removed.
3. Chunks contain variables so that they may match working memory in other situations; justifications are similar to an instantiated chunk.

2.8 Input and Output

Many Soar users will want their programs to interact with a real or simulated environment. For example, Soar programs may control a robot, receiving sensory inputs and sending command outputs. Soar programs may also interact with simulated environments, such as a flight simulator. Input is viewed as Soar's perception and output is viewed as Soar's motor abilities.

When Soar interacts with an external environment, it must make use of mechanisms that allow it to receive input from that environment and to effect changes in that environment; the mechanisms provided in Soar are called *input functions* and *output functions*.

Input functions add and delete elements from working memory in response to changes in the external environment.

Output functions attempt to effect changes in the external environment.

Input is processed at the beginning of each execution cycle and output occurs at the end of each execution cycle.

For instructions on how to use input and output functions with Soar, refer to the *SML Quick Start Guide*.

Chapter 3

The Syntax of Soar Programs

This chapter describes in detail the syntax of elements in working memory, preference memory, and production memory, and how impasses and I/O are represented in working memory and in productions. Working memory elements and preferences are created as Soar runs, while productions are created by the user or through chunking. The bulk of this chapter explains the syntax for writing productions.

The first section of this chapter describes the structure of working memory elements in Soar; the second section describes the structure of preferences; and the third section describes the structure of productions. The fourth section describes the structure of impasses. An overview of how input and output appear in working memory is presented in the fifth section; the full discussion of Soar I/O can be found in the *SML Quick Start Guide*.

This chapter assumes that you understand the operating principles of Soar, as presented in Chapter 2.

3.1   Working Memory

Working memory contains *working memory elements* (WME's). As described in Section 2.2, WME's can be created by the actions of productions, the evaluation of preferences, the Soar architecture, and via the input/output system.

A WME is a list consisting of three symbols: an *identifier*, an *attribute*, and a *value*, where the entire WME is enclosed in parentheses and the attribute is preceded by an up-arrow (^). A template for a working memory element is:

```
(identifier ^attribute value)
```

The identifier is an internal symbol, generated by the Soar architecture as it runs. The attribute and value can be either identifiers or constants; if they are identifiers, there are other working memory elements that have that identifier in their first position. As the previous sentences demonstrate, identifier is used to refer both to the first position of a working memory element, as well as to the symbols that occupy that position.

3.1.1   Symbols

Soar distinguishes between two types of working memory symbols: *identifiers* and *constants*.

Identifiers: An identifier is a unique symbol, created at runtime when a new object is added to working memory. The names of identifiers are created by Soar, and consist of a single uppercase letter followed by a string of digits, such as `G37` or `O22`.

(The Soar user interface will also allow users to specify identifiers using lowercase letters, for example, when using the `print` command. But internally, they are actually uppercase letters.)

Constants: There are three types of constants: integers, floating-point, and symbolic constants:

- Integer constants (numbers). The range of values depends on the machine and implementation you're using, but it is at least [-2 billion..2 billion].
- Floating-point constants (numbers). The range depends on the machine and implementation you're using.
- Symbolic constants. These are symbols with arbitrary names. A constant can use any combination of letters, digits, or `$%&*+-/:<=>?_` Other characters (such as blank spaces) can be included by surrounding the complete constant name with vertical bars: `|This is a constant|`. (The vertical bars aren't part of the name; they're just notation.) A vertical bar can be included by prefacing it with a backslash inside surrounding vertical bars: `|Odd-symbol\|name|`

Identifiers should not be confused with constants, although they may "look the same"; identifiers are generated (by the Soar architecture) at runtime and will not necessarily be the same for repeated runs of the same program. Constants are specified in the Soar program and will be the same for repeated runs.

Even when a constant "looks like" an identifier, it will not act like an identifier in terms of matching. A constant is printed surrounded by vertical bars whenever there is a possibility of confusing it with an identifier: `|G37|` is a constant while `G37` is an identifier. To avoid possible confusion, you should not use letter-number combinations as constants or for production names.

3.1.2   Objects

Recall from Section 2.2 that all WME's that share an identifier are collectively called an *object* in working memory. The individual working memory elements that make up an object are often called *augmentations*, because they augment the object. A template for an object in working memory is:

```
(identifier ^attribute-1 value-1 ^attribute-2 value-2
            ^attribute-3 value-3... ^attribute-n value-n)
```

For example, if you run Soar with the example blocks-world program described in Appendix A, after one elaboration cycle, you can look at the top-level state by using the `print` command:

```
soar> print s1
(S1 ^io I1 ^ontop O2 ^ontop O3 ^ontop O1 ^problem-space blocks
    ^superstate nil ^thing B3 ^thing T1 ^thing B1 ^thing B2
    ^type state)
```

The attributes of an object are printed in alphabetical order to make it easier to find a specific attribute.

Working memory is a set, so that at any time, there are never duplicate versions of working memory elements. However, it is possible for several working memory elements to share the same identifier and attribute but have different values. Such attributes are called multi-valued attributes or *multi-attributes*. For example, state `S1`, above, has two attributes that are multi-valued: `thing` and `ontop`.

3.1.3 Timetags

When a working memory element is created, Soar assigns it a unique integer *timetag*. The timetag is a part of the working memory element, and therefore, WME's are actually quadruples, rather than triples. However, the timetags are not represented in working memory and cannot be matched by productions. The timetags are used to distinguish between multiple occurrences of the same WME. As preferences change and elements are added and deleted from working memory, it is possible for a WME to be created, removed, and created again. The second creation of the WME — which bears the same identifier, attribute, and value as the first WME — is *different*, and therefore is assigned a different timetag. This is important because a production will fire only once for a given instantiation, and the instantiation is determined by the timetags that match the production and not by the identifier-attribute-value triples.

To look at the timetags of WMEs, the `wmes` command can be used:

```
soar> wmes s1
(3: S1 ^io I1)
(10: S1 ^ontop O2)
(9: S1 ^ontop O3)
(11: S1 ^ontop O1)
(4: S1 ^problem-space blocks)
(2: S1 ^superstate nil)
(6: S1 ^thing B3)
(5: S1 ^thing T1)
(8: S1 ^thing B1)
(7: S1 ^thing B2)
(1: S1 ^type state)
```

This shows all the individual augmentations of `S1`, each is preceded by an integer *timetag*.

3.1.4   Acceptable preferences in working memory

The acceptable preferences for the operator augmentations of states appear in working memory as identifier-attribute-value-preference quadruples. No other preferences appear in working memory. A template for an acceptable preference in working memory is:

```
(identifier ^operator value +)
```

For example, if you run Soar with the example blocks-world program described in Appendix A, after the first operator has been selected, you can again look at the top-level state using the `wmes` command:

```
soar> wmes s1
(3: S1 ^io I1)
(9: S1 ^ontop O3)
(10: S1 ^ontop O2)
(11: S1 ^ontop O1)
(48: S1 ^operator O4 +)
(49: S1 ^operator O5 +)
(50: S1 ^operator O6 +)
(51: S1 ^operator O7 +)
(54: S1 ^operator O7)
(52: S1 ^operator O8 +)
(53: S1 ^operator O9 +)
(4: S1 ^problem-space blocks)
(2: S1 ^superstate nil)
(5: S1 ^thing T1)
(8: S1 ^thing B1)
(6: S1 ^thing B3)
(7: S1 ^thing B2)
(1: S1 ^type state)
```

The state `S1` has six augmentations of acceptable preferences for different operators (`O4` through `O9`). These have plus signs following the value to denote that they are acceptable preferences. The state has exactly one operator, `O7`. This state corresponds to the illustration of working memory in Figure 2.4.

3.1.5   Working Memory as a Graph

Not only is working memory a set, it is also a graph structure where the identifiers are nodes, attributes are links, and constants are terminal nodes. Working memory is not an arbitrary graph, but a graph rooted in the states. Therefore, all WMEs are *linked* either directly or
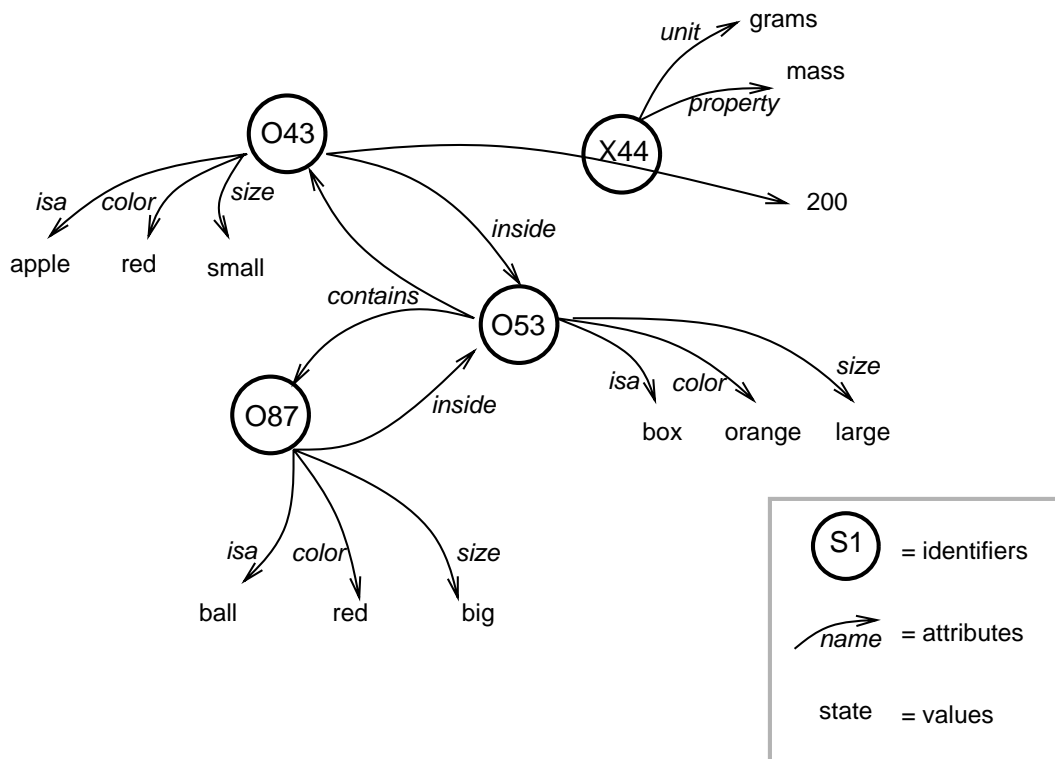
Figure 3.1: A semantic net illustration of four objects in working memory.

indirectly to a state. The impact of this constraint is that all WMEs created by actions are linked to WMEs tested in the conditions. The link is one-way, from the identifier to the value. Less commonly, the attribute of a WME may be an identifier.

Figure 3.1 illustrates four objects in working memory; the object with identifier X44 has been linked to the object with identifier O43, using the attribute as the link, rather than the value. The objects in working memory illustrated by this figure are:

```
(O43 ^isa apple ^color red ^inside O53 ^size small ^X44 200)
(O87 ^isa ball ^color red ^inside O53 ^size big)
(O53 ^isa box ^size large ^color orange ^contains O43 O87)
(X44 ^unit grams ^property mass)
```

In this example, object O43 and object O87 are both linked to object O53 through (O53 ^contains O43) and (O53 ^contains O87), respectively (the contains attribute is a multi-valued attribute). Likewise, object O53 is linked to object O43 through (O43 ^inside O53) and linked to object O87 through (O87 ^inside O53). Object X44 is linked to object O43 through (O43 ^X44 200).

Links are transitive so that X44 is linked to O53 (because O43 is linked to O53 and X44 is linked to O43). However, since links are not symmetric, O53 is not linked to X44.

3.2   Preference Memory

Preferences are created by production firings and express the relative or absolute merits for selecting an operator for a state. When preferences express an absolute rating, they are identifier-attribute-value-preference quadruples; when preferences express relative ratings, they are identifier-attribute-value-preference-value quintuples

For example,

```
(S1 ^operator O3 +)
```

is a preference that asserts that operator O3 is an acceptable operator for state S1, while

```
(S1 ^operator O3 > O4)
```

is a preference that asserts that operator O3 is a better choice for the operator of state S1 than operator O4.

The semantics of preferences and how they are processed were described in Section 2.4, which also described each of the eleven different types of preferences. Multiple production instantiations may create identical preferences. Unlike working memory, preference memory is not a set: Duplicate preferences are allowed in preference memory.

3.3   Production Memory

Production memory contains productions, which can be loaded in by a user (typed in while Soar is running or `source`d from a file) or generated by chunking while Soar is running. Productions (both user-defined productions and chunks) may be examined using the `print` command, described in Section 8.2 on page 135.

Each production has three required components: a name, a set of conditions (also called the left-hand side, or LHS), and a set of actions (also called the right-hand side, or RHS). There are also two optional components: a documentation string and a type.

Syntactically, each production consists of the symbol `sp`, followed by: an opening curly brace, {; the production's name; the documentation string (optional); the production type (optional); comments (optional); the production's conditions; the symbol `-->` (literally: dash-dash-greaterthan); the production's actions; and a closing curly brace, }. Each element of a production is separated by white space. Indentation and linefeeds are used by convention, but are not necessary.

```
sp {production-name
    Documentation string
    :type
    CONDITIONS
    -->
    ACTIONS
    }
```

```
sp {blocks-world*propose*move-block
   (state <s> ^problem-space blocks
              ^thing <thing1> {<> <thing1> <thing2>}
              ^ontop <ontop>)
   (<thing1> ^type block ^clear yes)
   (<thing2> ^clear yes)
   (<ontop> ^top-block <thing1>
            ^bottom-block <> <thing2>)
   -->
   (<s> ^operator <o> +)
   (<o> ^name move-block
        ^moving-block <thing1>
        ^destination <thing2>)}
```

Figure 3.2: An example production from the example blocks-world task.

An example production, named "`blocks-world*propose*move-block`", is shown in Figure 3.2. This production proposes operators named `move-block` that move blocks from one location to another. The details of this production will be described in the following sections.

Conventions for indenting productions

Productions in this manual are formatted using conventions designed to improve their readability. These conventions are not part of the required syntax. First, the name of the production immediately follows the first curly bracket after the `sp`. All conditions are aligned with the first letter after the first curly brace, and attributes of an object are all aligned The arrow is indented to align with the conditions and actions and the closing curly brace follows the last action.

3.3.1 Production Names

The name of the production is an almost arbitrary constant. (See Section 3.1.1 for a description of constants.) By convention, the name describes the role of the production, but functionally, the name is just a label primarily for the use of the programmer.

A production name should never be a single letter followed by numbers, which is the format of identifiers.

The convention for naming productions is to separate important elements with asterisks; the important elements that tend to appear in the name are:

1. The name of the task or goal (e.g., `blocks-world`).
2. The name of the architectural function (e.g., `propose`).
3. The name of the operator (or other object) at issue. (e.g., `move-block`)
4. Any other relevant details.

This name convention enables one to have a good idea of the function of a production just by examining its name. This can help, for example, when you are watching Soar run and looking at the specific productions that are firing and retracting. Since Soar uses white space to delimit components of a production, if whitespace inadvertently occurs in the production name, Soar will complain that an open parenthesis was expected to start the first condition.

3.3.2   Documentation string (optional)

A production may contain an optional documentation string. The syntax for a documentation string is that it is enclosed in double quotes and appears after the name of the production and before the first condition (and may carry over to multiple lines). The documentation string allows the inclusion of internal documentation about the production; it will be printed out when the production is printed using the `print` command.

3.3.3   Production type (optional)

A production may also include an optional *production type*, which may specify that the production should be considered a default production (`:default`) or a chunk (`:chunk`), or may specify that a production should be given O- support (`:o-support`) or I-support (`:i-support`). Users are discouraged from using these types. These types are described in Section 8.1, which begins on Page 123.

There is one additional flag (`:interrupt`) which can be placed at this location in a production. However this flag does not specify a production type, but is a signal that the production should be marked for special debugging capabilities. For more information, see Section 8.1 on Page 123.

3.3.4   Comments (optional)

Productions may contain comments, which are not stored in Soar when the production is loaded, and are therefore not printed out by the `print` command. A comment is begun with a pound sign character `#` and ends at the end of the line. Thus, everything following the `#` is not considered part of the production, and comments that run across multiple lines must each begin with a `#`.

For example:

```
sp {blocks-world*propose*move-block
   (state <s> ^problem-space blocks
             ^thing <thing1> {<> <thing1> <thing2>}
             ^ontop <ontop>)
   (<thing1> ^type block ^clear yes)
   (<thing2> ^clear yes)
#  (<ontop> ^top-block <thing1>
```

```
#            ^bottom-block <> <thing2>)
   -->
   (<s> ^operator <o> +)
   (<o> ^name move-block        # you can also use in-line comments
        ^moving-block <thing1>
        ^destination <thing2>)}
```

When commenting out conditions or actions, be sure that all parentheses remain balanced outside the comment.

External comments

Comments may also appear in a file with Soar productions, outside the curly braces of the `sp` command. Comments must either start a new line with a `#` or start with `;#`. In both cases, the comment runs to the end of the line.

```
# imagine that this is part of a "Soar program" that contains
# Soar productions as well as some other code.

source blocks.soar      ;# this is also a comment
```

3.3.5 The condition side of productions (or LHS)

The condition side of a production, also called the left-hand side (or LHS) of the production, is a pattern for matching one or more WMEs. When all of the conditions of a production match elements in working memory, the production is said to be instantiated, and is ready to perform its action.

The following subsections describe the condition side of a production, including predicates, disjunctions, conjunctions, negations, acceptable preferences for operators, and a few advanced topics.

3.3.5.1 Conditions

The condition side of a production consists of a set of conditions. Each condition tests for the existence or absence (explained later in Section 3.3.5.6) of working memory elements. Each condition consists of a open parenthesis, followed by a test for the identifier, and the tests for augmentations of that identifier, in terms of attributes and values. The condition is terminated with a close parenthesis. Thus, a single condition might test properties of a single working memory element, or properties of multiple working memory elements that constitute an object.

```
(identifier-test ^attribute1-test value1-test
                 ^attribute2-test value2-test
                 ^attribute3-test value3-test
```

```
                    ...)
```

The first condition in a production must match against a state in working memory. Thus, the first condition must begin with the additional symbol "state". All other conditions and actions must be *linked* directly or indirectly to this condition. This linkage may be direct to the state, or it may be indirect, through objects specified in the conditions. If the identifiers of the actions are not linked to the state, a warning is printed when the production is parsed, and the production is not stored in production memory. In the actions of the example production shown in Figure 3.2, the operator preference is directly linked to the state and the remaining actions are linked indirectly via the operator preference.

Although all of the attribute tests in the template above are followed by value tests, it is possible to test for only the existence of an attribute and not test any specific value by just including the attribute and no value. Another exception to the above template is operator preferences, which have the following structure where a plus sign follows the value test.

```
(state-identifier-test ^operator value1-test +
                   ...)
```

In the remainder of this section, we describe the different tests that can be used for identifiers, attributes, and values. The simplest of these is a constant, where the constant specified in the attribute or value must match the same constant in a working memory element.

3.3.5.2   Variables in productions

Variables match against constants in working memory elements in the identifier, attribute, or value positions. Variables can be further constrained by additional tests (described in later sections) or by multiple occurrences in conditions. If a variable occurs more than once in the condition of a production, the production will match only if the variables match the same identifier or constant. However, there is no restriction that prevents different variables from binding to the same identifier or constant.

Because identifiers are generated by Soar at run time, it impossible to include tests for specific identifiers in conditions. Therefore, variables are used in conditions whenever an identifier is to be matched.

Variables also provide a mechanism for passing identifiers and constants which match in conditions to the action side of a rule.

Syntactically, a variable is a symbol that begins with a left angle-bracket (i.e., `<`), ends with a right angle-bracket (i.e., `>`), and contains at least one alphanumeric symbol in between.

In the example production in Figure 3.2, there are seven variables: `<s>`, `<clear1>`, `<clear2>`, `<ontop>`, `<block1>`, `<block2>`, and `<o>`.

The following table gives examples of legal and illegal variable names.

Legal variables	Illegal variables
`<s>`	`<>`
`<1>`	`<1`
`<variable1>`	`variable>`
`<abc1>`	`<a b>`

3.3.5.3 Predicates for values

A test for an identifier, attribute, or value in a condition (whether constant or variable) can be modified by a preceding predicate. There are six predicates that can be used: `<>`, `<=>`, `<`, `<=`, `>=`, `>`.

Predicate	Semantics of Predicate
`<>`	Not equal. Matches anything except the value immediately following it.
`<=>`	Same type. Matches any symbol that is the same type (identifier, integer, floating-point, non-numeric constant) as the value immediately following it.
`<`	Numerically less than the value immediately following it.
`<=`	Numerically less than or equal to the value immediately following it.
`>=`	Numerically greater than or equal to the value immediately following it.
`>`	Numerically greater than the value immediately following it.

The following table shows examples of legal and illegal predicates:

Legal predicates	Illegal predicates
`> <valuex>`	`> > <valuey>`
`< 1`	`1 >`
`<=> <y>`	`= 10`

Example Production

```
sp {propose-operator*to-show-example-predicate
   (state <s> ^car <c>)
   (<c> ^style convertible ^color <> rust)
   -->
   (<s> ^operator <o> +)
   (<o> ^name drive-car ^car <c>) }
```

In this production, there must be a "color" attribute for the working memory object that matches `<c>`, and the value of that attribute must not be "rust".

3.3.5.4 Disjunctions of values

A test for an identifier, attribute, or value may also be for a disjunction of constants. With a disjunction, there will be a match if any one of the constants is found in a working memory element (and the other parts of the working memory element matches). Variables and predicates may not be used within disjunctive tests.

Syntactically, a disjunctive test is specified with double angle brackets (i.e., `<<` and `>>`). There must be spaces separating the brackets from the constants.

The following table provides examples of legal and illegal disjunctions:

Legal disjunctions	Illegal disjunctions
`<< A B C 45 I17 >>`	`<< <A> A >>`
`<< 5 10 >>`	`<< < 5 > 10 >>`
`<< good-morning good-evening >>`	`<<A B C >>`

Example Production

For example, the third condition of the following production contains a disjunction that restricts the color of the table to `red` or `blue`:

```
sp {blocks*example-production-conditions
   (state ^operator <o> + ^table <t>)
   (<o> ^name move-block)
   (<t> ^type table ^color << red blue >> )
   -->
   ... }
```

Note

Disjunctions of complete conditions are not allowed in Soar. Multiple (similar) productions fulfill this role.

3.3.5.5 Conjunctions of values

A test for an identifier, attribute, or value in a condition may include a conjunction of tests, all of which must hold for there to be a match.

Syntactically, conjuncts are contained within curly braces (i.e., { and }). The following table shows some examples of legal and illegal conjunctive tests:

Legal conjunctions	Illegal conjunctions
`{ <= <a> >= <b> }`	`{ <x> < <a> + <b> }`
`{ <x> > <y> }`	`{ > > <b> }`
`{ <> <x> <y> }`	
`{ << A B C >> <x> }`	
`{ <=> <x> > <y> << 1 2 3 4 >> <z> }`	

Because those examples are a bit difficult to interpret, let's go over the legal examples one by one to understand what each is doing.

In the first example, the value must be less than or equal to the value bound to variable `<a>` and greater than or equal to the value bound to variable `<b>`.

In the second example, the value is bound to the variable `<x>`, which must also be greater than the value bound to variable `<y>`.

In the third example, the value must not be equal to the value bound to variable `<x>` and should be bound to variable `<y>`. Note the importance of order when using conjunctions with predicates: in the second example, the predicate modifies `<y>`, but in the third example, the predicate modifies `<x>`.

In the fourth example, the value must be one of `A`, `B`, or `C`, and the second conjunctive test binds the value to variable `<x>`.

In the fifth example, there are four conjunctive tests. First, the value must be the same type as the value bound to variable `<x>`. Second, the value must be greater than the value bound to variable `<y>`. Third, the value must be equal to `1`, `2`, `3`, or `4`. Finally, the value should be bound to variable `<z>`.

In Figure 3.2, a conjunctive test is used for the `thing` attribute in the first condition.

3.3.5.6 Negated conditions

In addition to the positive tests for elements in working memory, conditions can also test for the absence of patterns. A *negated condition* will be matched only if there does not exist a working memory element consistent with its tests and variable bindings. Thus, it is a test for the *absence* of a working memory element.

Syntactically, a negated condition is specified by preceding a condition with a dash (i.e., "`-`").

For example, the following condition tests the absence of a working memory element of the object bound to `<p1>` ^`type father`.

`-(<p1> ^type father)`

A negation can be used within an object with many attribute-value pairs by having it precede a specific attribute:

```
(<p1> ^name john -^type father ^spouse <p2>)
```

In that example, the condition would match if there is a working memory element that matches (<p1> ^name john) and another that matches (<p1> ^spouse <p2>), but is no working memory element that matches (<p1> ^type father) (when p1 is bound to the same identifier).

On the other hand, the condition:

```
-(<p1> ^name john ^type father ^spouse <p2>)
```

would match only if there is no object in working memory that matches all three attribute-value tests.

Example Production

```
sp {default*evaluate-object
   (state <ss> ^operator <so>)
   (<so> ^type evaluation
         ^superproblem-space <p>)
  -(<p> ^default-state-copy no)
   -->
   (<so> ^default-state-copy yes) }
```

Notes

One use of negated conditions to avoid is testing for the absence of the working memory element that a production creates with I-support; this would lead to an "infinite loop" in your Soar program, as Soar would repeatedly fire and retract the production.

3.3.5.7   Negated conjunctions of conditions

Conditions can be grouped into conjunctive sets by surrounding the set of conditions with { and }. The production compiler groups the test in these conditions together. This grouping allows for negated tests of more than one working memory element at a time. In the example below, the state is tested to ensure that it does not have an object on the table.

```
sp {blocks*negated-conjunction-example
   (state <s> ^name top-state)
  -{(<s> ^ontop <on>)
    (<on> ^bottom-object <bo>)
    (<bo> ^type table)}
   -->
   (<s> ^nothing-ontop-table true) }
```

When using negated conjunctions of conditions, the production has nested curly braces. One set of curly braces delimits the production, while the other set delimits the conditions to be conjunctively negated.

If only the last condition, (`<bo>` `^type table`) were negated, the production would match only if the state *had* an ontop relation, and the ontop relation had a bottom-object, but the bottom object wasn't a table. Using the negated conjunction, the production will also match when the state has no ontop augmentation or when it has an ontop augmentation that doesn't have a bottom-object augmentation.

The semantics of negated conjunctions can be thought of in terms of mathematical logic, where the negation of $(A \wedge B \wedge C)$:

$\neg(A \wedge B \wedge C)$

can be rewritten as:

$(\neg A) \vee (\neg B) \vee (\neg C)$

That is, "not (A and B and C)" becomes "(not A) or (not B) or (not C)".

3.3.5.8   Multi-valued attributes

An object in working memory may have multiple augmentations that specify the same attribute with different values; these are called multi-valued attributes, or multi-attributes for short. To shorten the specification of a condition, tests for multi-valued attributes can be shortened so that the value tests are together.

For example, the condition:

```
(<p1> ^type father ^child sally ^child sue)
```

could also be written as:

```
(<p1> ^type father ^child sally sue)
```

Multi-valued attributes and variables

When variables are used with multi-valued attributes, remember that variable bindings are not unique unless explicitly forced to be so. For example, to test that an object has two values for attribute `child`, the variables in the following condition can match to the same value.

```
(<p1> ^type father ^child <c1> <c2>)
```

To do tests for multi-valued attributes with variables correctly, conjunctive tests must be used, as in:

```
(<p1> ^type father ^child <c1> {<> <c1> <c2>})
```

The conjunctive test   {<> <c1> <c2>}   ensures that <c2> will bind to a different value than
<c1> binds to.

Negated conditions and multi-valued attributes

A negation can also precede an attribute with multiple values. In this case it tests for the
absence of the conjunction of the values. For example

```
(<p1> ^name john -^child oprah uma)
```

is the same as

```
(<p1> ^name john)
-{(<p1> ^child oprah)
  (<p1> ^child uma)}
```

and the match is possible if either (<p1> $^\wedge$child oprah) or (<p1> $^\wedge$child uma) cannot be
found in working memory with the binding for <p1> (but not if both are present).

3.3.5.9   Acceptable preferences for operators

The only preferences that can appear in working memory are acceptable preferences for oper-
ators, and therefore, the only preferences that may appear in the conditions of a production
are acceptable preferences for operators.

Acceptable preferences for operators can be matched in a condition by testing for a "+"
following the value. This allows a production to test the existence of a candidate operator
and its properties, and possibly create a preference for it, before it is selected.

In the example below, $^\wedge$operator <o> + matches the acceptable preference for the operator
augmentation of the state. *This does not test that operator* <o> *has been selected as the
current operator.*

```
sp {blocks*example-production-conditions
   (state ^operator <o> + ^table <t>)
   (<o> ^name move-block)
   -->
   ... }
```

In the example below, the production tests the state for acceptable preferences for two
different operators (and also tests that these operators move different blocks):

```
sp {blocks*example-production-conditions
   (state ^operator <o1> + <o2> + ^table <t>)
   (<o1> ^name move-block ^moving-block <m1> ^destination <d1>)
   (<o2> ^name move-block ^moving-block {<m2> <> <m1>}
         ^destination <d2>)
   -->
```

```
   ... }
```

3.3.5.10   Attribute tests

The previous examples applied all of the different test to the values of working memory elements. All of the tests that can be used for values can also be used for attributes and identifiers (except those including constants).

Variables in attributes

Variables may be used with attributes, as in:

```
sp {blocks*example-production-conditions
   (state <s> ^operator <o> +
              ^thing <t> {<> <t> <t2>} )
   (operator <o> ^name group
                 ^by-attribute <a>
                 ^moving-block <t>
                 ^destination <t2>)
   (<t> ^type block ^<a> <x>)
   (<t2> ^type block ^<a> <x>)
   -->
   (<s> ^operator <o> >) }
```

This production tests that there is acceptable operator that is trying to group blocks according to some attribute, `<a>`, and that block `<t>` and `<t2>` both have this attribute (whatever it is), and have the same value for the attribute.

Predicates in attributes

Predicates may be used with attributes, as in:

```
sp {blocks*example-production-conditions
   (state ^operator <o> + ^table <t>)
   (<t> ^<> type table)
   -->
   ... }
```

which tests that the object with its identifier bound to `<t>` must have an attribute whose value is `table`, but the name of this attribute is not `type`.

Disjunctions of attributes

Disjunctions may also be used with attributes, as in:

```
sp {blocks*example-production-conditions
   (state ^operator <o> + ^table <t>)
   (<t> ^<< type name>> table)
   -->
   ... }
```

which tests that the object with its identifier bound to `<t>` must have either an attribute
`type` whose value is `table` or an attribute `name` whose value is `table`.

Conjunctive tests for attributes

Section 3.3.5.5 illustrated the use of conjunctions for the values in conditions. Conjunctive
tests may also be used with attributes, as in:

```
sp {blocks*example-production-conditions
   (state ^operator <o> + ^table <t>)
   (<t> ^{<ta> <> name} table)
   -->
   ... }
```

which tests that the object with its identifier bound to `<t>` must have an attribute whose
value is `table`, and the name of this attribute is not `name`, and the name of this attribute
(whatever it is) is bound to the variable `<ta>`.

When attribute predicates or attribute disjunctions are used with multi-valued attributes, the
production is rewritten internally to use a conjunctive test for the attribute; the conjunctive
test includes a variable used to bind to the attribute name. Thus,

```
(<p1> ^type father ^ <> name sue sally)
```

is interpreted to mean:

```
(<p1> ^type father ^ {<> name <a*1>} sue ^ <a*1> sally)
```

3.3.5.11   Attribute-path notation

Often, variables appear in the conditions of productions only to link the value of one attribute
with the identifier of another attribute. Attribute-path notation provides a shorthand so that
these intermediate variables do not need to be included.

Syntactically, path notation lists a sequence of attributes separated by dots (.), after the ^
in a condition.

For example, using attribute path notation, the production:

```
sp {blocks-world*monitor*move-block
   (state <s> ^operator <o>)
   (<o> ^name move-block
        ^moving-block <block1>
```

```
            ^destination <block2>)
    (<block1> ^name <block1-name>)
    (<block2> ^name <block2-name>)
    -->
    (write (crlf) |Moving Block: | <block1-name>
                  | to: | <block2-name> ) }
```

could be written as:

```
sp {blocks-world*monitor*move-block
    (state <s> ^operator <o>)
    (<o> ^name move-block
         ^moving-block.name <block1-name>
         ^destination.name <block2-name>)
    -->
    (write (crlf) |Moving Block: | <block1-name>
                  | to: | <block2-name> ) }
```

Attribute-path notation yields shorter productions that are easier to write, less prone to errors, and easier to understand.

When attribute-path notation is used, Soar internally expands the conditions into the multiple Soar objects, creating its own variables as needed. Therefore, when you print a production (using the `print` command), the production will not be represented using attribute-path notation.

Negations and attribute path notation

A negation may be used with attribute path notation, in which case it amounts to a negated conjunction. For example, the production:

```
sp {blocks*negated-conjunction-example
    (state <s> ^name top-state)
  -{(<s> ^ontop <on>)
    (<on> ^bottom-object <bo>)
    (<bo> ^type table)}
    -->
    (<s> ^nothing-ontop-table true) }
```

could be rewritten as:

```
sp {blocks*negated-conjunction-example
    (state <s> ^name top-state -^ontop.bottom-object.type table)
    -->
    (<s> ^nothing-ontop-table true) }
```

Multi-valued attributes and attribute path notation

Attribute path notation may also be used with multi-valued attributes, such as:

```
sp {blocks-world*propose*move-block
   (state <s> ^problem-space blocks
              ^clear.block <block1> { <> <block1> <block2> }
              ^ontop <ontop>)
   (<block1> ^type block)
   (<ontop> ^top-block <block1>
            ^bottom-block <> <block2>)
   -->
   (<s> ^operator <o> +)
   (<o> ^name move-block +
        ^moving-block <block1> +
        ^destination <block2> +) }
```

Multi-attributes and attribute-path notation

Note: It would not be advisable to write the production in Figure 3.2 using attribute-path notation as follows:

```
sp {blocks-world*propose*move-block*dont-do-this
   (state <s> ^problem-space blocks
              ^clear.block <block1>
              ^clear.block { <> <block1> <block2> }
              ^ontop.top-block <block1>
              ^ontop.bottom-block <> <block2>)
   (<block1> ^type block)
   -->
   ...
   }
```

This is not advisable because it corresponds to a different set of conditions than those in the original production (the `top-block` and `bottom-block` need not correspond to the same `ontop` relation). To check this, we could print the original production at the Soar prompt:

```
soar> print blocks-world*propose*move-block*dont-do-this
sp {blocks-world*propose*move-block*dont-do-this
    (state <s> ^problem-space blocks ^thing <thing2>
          ^thing { <> <thing2> <thing1> } ^ontop <o*1> ^ontop <o*2>)
   (<thing2> ^clear yes)
   (<thing1> ^clear yes ^type block)
   (<o*1> ^top-block <thing1>)
   (<o*2> ^bottom-block { <> <thing2> <b*1> })
   -->
   (<s> ^operator <o> +)
```

```
    (<o> ^name move-block
         ^moving-block <thing1>
         ^destination <thing2>) }
```

Soar has expanded the production into the longer form, and created two distinctive variables, $<o*1>$ and $<o*2>$ to represent the `ontop` attribute. These two variables will not necessarily bind to the same identifiers in working memory.

Negated multi-valued attributes and attribute-path notation

Negations of multi-valued attributes can be combined with attribute-path notation. However; it is very easy to make mistakes when using negated multi-valued attributes with attribute-path notation. Although it is possible to do it correctly, we strongly discourage its use.

For example,

```
sp {blocks*negated-conjunction-example
   (state <s> ^name top-state -^ontop.bottom-object.name table A)
   -->
   (<s> ^nothing-ontop-A-or-table true) }
```

gets expanded to:

```
sp {blocks*negated-conjunction-example
   (state <s> ^name top-state)
  -{(<s> ^ontop <o*1>)
    (<o*1> ^bottom-object <b*1>)
    (<b*1> ^name A)
    (<b*1> ^name table)}
   -->
   (<s> ^nothing-ontop-A-or-table true) }
```

This example does not refer to two different blocks with different names. It tests that there is not an `ontop` relation with a `bottom-block` that is named `A` and named `table`. Thus, this production probably should have been written as:

```
sp {blocks*negated-conjunction-example
   (state <s> ^name top-state
              -^ontop.bottom-object.name table
              -^ontop.bottom-object.name A)
   -->
   (<s> ^nothing-ontop-A-or-table true) }
```

which expands to:

```
sp {blocks*negated-conjunction-example
   (state <s> ^name top-state)
  -{(<s> ^ontop <o*2>)
```

```
    (<o*2> ^bottom-object <b*2>)
    (<b*2> ^name a)}
 -{(<s> ^ontop <o*1>)
    (<o*1> ^bottom-object <b*1>)
    (<b*1> ^name table)}
   -->
   (<s> ^nothing-ontop-a-or-table true +) }
```

Notes on attribute-path notation

- Attributes specified in attribute-path notation may not start with a digit. For example, if you type ^foo.3.bar, Soar thinks the .3 is a floating-point number. (Attributes that don't appear in path notation can begin with a number.)

- Attribute-path notation may be used to any depth.

- Attribute-path notation may be combined with structured values, described in Section 3.3.5.12.

3.3.5.12   Structured-value notation

Another convenience that eliminates the use of intermediate variables is structured-value notation.

Syntactically, the attributes and values of a condition may be written where a variable would normally be written. The attribute-value structure is delimited by parentheses.

Using structured-value notation, the production in Figure 3.2 (on page 39) may also be written as:

```
sp {blocks-world*propose*move-block
   (state <s> ^problem-space blocks
              ^thing <thing1> {<> <thing1> <thing2>}
              ^ontop (^top-block <thing1>
                      ^bottom-block <> <thing2>))
   (<thing1> ^type block ^clear yes)
   (<thing2> ^clear yes)
-->
   (<s> ^operator <o> +)
   (<o> ^name move-block
        ^moving-block <thing1>
        ^destination <thing2>) }
```

Thus, several conditions may be "collapsed" into a single condition.

Using variables within structured-value notation

Variables are allowed within the parentheses of structured-value notation to specify an identifier to be matched elsewhere in the production. For example, the variable `<ontop>` could be added to the conditions (although it are not referenced again, so this is not helpful in this instance):

```
sp {blocks-world*propose*move-block
   (state <s> ^problem-space blocks
             ^thing <thing1> {<> <thing1> <thing2>}
             ^ontop (<ontop>
                     ^top-block <thing1>
                     ^bottom-block <> <thing2>))
   (<thing1> ^type block ^clear yes)
   (<thing2> ^clear yes)
   -->
   (<s> ^operator <o> +)
   (<o> ^name move-block
        ^moving-block <thing1>
        ^destination <thing2>) }
```

Structured values may be nested to any depth. Thus, it is possible to write our example production using a single condition with multiple structured values:

```
sp {blocks-world*propose*move-block
   (state <s> ^problem-space blocks
             ^thing <thing1>
                    ({<> <thing1> <thing2>}
                     ^clear yes)
             ^ontop (^top-block
                       (<thing1>
                        ^type block
                        ^clear yes)
                     ^bottom-block <> <thing2>) )
   -->
   (<s> ^operator <o> +)
   (<o> ^name move-block
        ^moving-block <thing1>
        ^destination <thing2>) }
```

Notes on structured-value notation

- Attribute-path notation and structured-value notation are orthogonal and can be combined in any way. A structured value can contain an attribute path, or a structure can be given as the value for an attribute path.

- Structured-value notation may also be combined with negations and with multi-attributes.

- Structured-value notation may not be used in the actions of productions.

3.3.6   The action side of productions (or RHS)

The action side of a production, also called the right-hand side (or RHS) of the production, consists of individual actions that can:

- Add new elements to working memory.

- Remove elements from working memory.

- Create preferences.

- Perform other actions

When the conditions of a production match working memory, the production is said to be instantiated, and the production will fire during the next elaboration cycle. Firing the production involves performing the actions *using the same variable bindings* that formed the instantiation.

3.3.6.1   Variables in Actions

Variables can be used in actions. A variable that appeared in the condition side will be replaced with the value that is was bound to in the condition. A variable that appears only in the action side will be bound to a new identifier that begins with the first letter of that variable (e.g., `<o>` might be bound to `o234`). This symbol is guaranteed to be unique and it will be used for all occurrences of the variable in the action side, appearing in all working memory elements and preferences that are created by the production action.

3.3.6.2   Creating Working Memory Elements

An element is created in working memory by specifying it as an action. Multiple augmentations of an object can be combined into a single action, using the same syntax as in conditions, including path notation and multi-valued attributes.

```
-->
(<s> ^block.color red
     ^thing <t1> <t2>) }
```

The action above is expanded to be:

```
-->
(<s> ^block <*b>)
(<*b> ^color red)
(<s> ^thing <t1>)
(<s> ^thing <t2>) }
```

This will add four elements to working memory with the variables replaced with whatever values they were bound to on the condition side.

Since Soar is case sensitive, different combinations of upper- and lowercase letters represent *different* constants. For example, "`red`", "`Red`", and "`RED`" are all distinct symbols in Soar. In many cases, it is prudent to choose one of uppercase or lowercase and write all constants in that case to avoid confusion (and bugs).

The constants that are used for attributes and values have a few restrictions on them:

1. There are a number of architecturally created augmentations for state and impasse objects; see Section 3.4 for a listing of these special augmentations. User-defined productions can not create or remove augmentations of states that use these attribute names.

2. Attribute names should not begin with a number if these attributes will be used in attribute-path notation.

3.3.6.3 Removing Working Memory Elements

A element is explicitly removed from working memory by following the value with a dash: `-`, also called a reject.

```
-->
(<s> ^block <b> -)}
```

If the removal of a working memory element removes the only link between the state and working memory elements that had the value of the removed element as an identifier, those working memory elements will be removed. This is applied recursively, so that all item that become unlinked are removed.

The reject should be used with an action that will be o-supported. If reject is attempted with I-support, the working memory element will reappear if the reject loses I-support and the element still has support.

3.3.6.4 The syntax of preferences

Below are the eleven types of preferences as they can appear in the actions of a production for the selection of operators:

RHS preferences	Semantics
(id $^\wedge$operator value)	acceptable
(id $^\wedge$operator value +)	acceptable
(id $^\wedge$operator value !)	require
(id $^\wedge$operator value $\sim$)	prohibit
(id $^\wedge$operator value -)	reject
(id $^\wedge$operator value > value2)	better
(id $^\wedge$operator value < value2)	worse
(id $^\wedge$operator value >)	best
(id $^\wedge$operator value <)	worst
(id $^\wedge$operator value =)	unary indifferent
(id $^\wedge$operator value = value2)	binary indifferent
(id $^\wedge$operator value = number)	numeric indifferent

The identifier and value will always be variables, such as (<s1> $^\wedge$operator <o1> > <o2>).

The preference notation appears similar to the predicate tests that appear on the left-hand side of productions, but has very different meaning. Predicates cannot be used on the right-hand side of a production and you cannot restrict the bindings of variables on the right-hand side of a production. (Such restrictions can happen only in the conditions.)

Also notice that the + symbol is optional when specifying acceptable preferences in the actions of a production, although using this symbol will make the semantics of your productions clearer in many instances. The + symbol will always appear when you inspect preference memory (with the `preferences` command).

Productions are never needed to delete preferences because preferences will be retracted when the production no longer matches. Preferences should never be created by operator application rules, and they should always be created by rules that will give only I-support to their actions.

3.3.6.5   Shorthand notations for preference creation

There are a few shorthand notations allowed for the creation of operator preferences on the right-hand side of productions.

Acceptable preferences do not need to be specified with a + symbol. (<s> $^\wedge$operator <op1>) is assumed to mean (<s> $^\wedge$operator <op1> +).

Ambiguity can easily arise when using a preference that can be either binary or unary: > < =. The default assumption is that if a value follows the preference, then the preference is binary. It will be unary if a carat (up-arrow), a closing parenthesis, another preference, or a comma follows it.

Below are four examples of legal, although unrealistic, actions that have the same effect.

```
(<s> ^operator <o1> <o2> + <o2> < <o1> <o3> =, <o4>)
(<s> ^operator <o1> + <o2> +
```

```
              <o2> < <o1> <o3> =, <o4> +)
(<s> ^operator <o1> <o2> <o2> < <o1> <o4> <o3> =)
(<s> ^operator <o1> ^operator <o2>
              ^operator <o2> < <o1> ^operator <o4> <o3> =)
```

Any one of those actions could be expanded to the following list of preferences:

```
(<s> ^operator <o1> +)
(<s> ^operator <o2> +)
(<s> ^operator <o2> < <o1>)
(<s> ^operator <o3> =)
(<s> ^operator <o4> +)
```

Note that structured-value notation may not be used in the actions of productions.

3.3.6.6   Righthand-side Functions

The fourth type of action that can occur in productions is called a *righthand-side function*. Righthand-side functions allow productions to create side effects other than changing working memory. The RHS functions are described below, organized by the type of side effect they have.

3.3.6.7   Stopping and pausing Soar

halt — Terminates Soar's execution and returns to the user prompt. A `halt` action irreversibly terminates the running of a Soar program. It should not be used if Soar is to be restarted (see the `interrupt` RHS action below.)

```
sp {
    ...
    -->
    (halt) }
```

interrupt —   Executing this function causes Soar to stop at the end of the current phase, and return to the user prompt. This is similar to `halt`, but does not terminate the run. The run may be continued by issuing a `run` command from the user interface. The `interrupt` RHS function has the same effect as typing `stop-soar` at the prompt, except that there is more control because it takes effect exactly at the end of the phase that fires the production.

```
sp {
    ...
    -->
    (interrupt) }
```

Soar execution may also be stopped immediately before a production fires, using the
`:interrupt` directive. This functionality is called a matchtime interrupt and is very
useful for debugging. See Section 8.1 on Page 123 for more information.

```
sp {production*name
    :interrupt
    ...
    -->
    ...
    }
```

3.3.6.8   Text input and output

The function `write` is provided as a production action to do simple output of text in Soar.
Soar applications that do extensive input and output of text should use Soar Markup Lan-
guage (SML). To learn about SML, read the "SML Quick Start Guide" which should be
located in the "Documentation" folder of your Soar install.

write — This function writes its arguments to the standard output. It does not auto-
matically insert blanks, linefeeds, or carriage returns. For example, if `<o>` is bound to
4, then

```
sp {
    ...
    -->
    (write  <o> <o> <o> | x| <o> | | <o>) }
```

prints

```
444 x4 4
```

crlf — Short for "carriage return, line feed", this function can be called only within
`write`. It forces a new line at its position in the `write` action.

```
sp {
    ...
    -->
    (write <x> (crlf) <y>) }
```

3.3.6.9   Mathematical functions

The expressions described in this section can be nested to any depth. For all of the functions
in this section, missing or non-numeric arguments result in an error.

+, -, *, / — These symbols provide prefix notation mathematical functions. These symbols work similarly to C functions. They will take either integer or real-number arguments. The first three functions return an integer when all arguments are integers and otherwise return a real number, and the last two functions always return a real number. The **-** symbol is also a unary function which, given a single argument, returns the product of the argument and **-1**. The **/** symbol is also a unary function which, given a single argument, returns the reciprocal of the argument (1/x).

```
sp {
    ...
    -->
    (<s> ^sum (+ <x> <y>)
         ^product-sum (* (+ <v> <w>) (+ <x> <y>))
         ^big-sum (+ <x> <y> <z> 402)
         ^negative-x (- <x>))
}
```

div, mod — These symbols provide prefix notation binary mathematical functions (they each take two arguments). These symbols work similarly to C functions: They will take only integer arguments (using reals results in an error) and return an integer: **div** takes two integers and returns their integer quotient; **mod** returns their remainder.

```
sp {
    ...
    -->
    (<s> ^quotient (div <x> <y>)
         ^remainder (mod <x> <y>)) }
```

abs, atan2, sqrt, sin, cos — These symbols provide prefix notation unary mathematical functions (they each take one argument). These symbols work similarly to C functions: They will take either integer or real-number arguments. The first function (**abs**) returns an integer when its argument is an integer and otherwise returns a real number, and the last four functions always return a real number. **atan2** returns as a float in radians, the arctangent of (first_arg / second_arg). **sin** and **cos** take as arguments the angle in radians.

```
sp {
    ...
    -->
    (<s> ^abs-value (abs <x>)
         ^sqrt (sqrt <x>)) }
```

int — Converts a single symbol to an integer constant. This function expects either an integer constant, symbolic constant, or floating point constant. The symbolic constant must be a string which can be interpreted as a single integer. The floating point

constant is truncated to only the integer portion. This function essentially operates as a type casting function.

For example, the expression `2 + sqrt(6)` could be printed as an integer using the following:

```
sp {
    ...
    -->
    (write (+ 2 (int sqrt(6))) ) }
```

float — Converts a single symbol to a floating point constant. This function expects either an integer constant, symbolic constant, or floating point constant. The symbolic constant must be a string which can be interpreted as a single floating point number. This function essentially operates as a type casting function.

For example, if you wanted to print out an integer expression as a floating-point number, you could do the following:

```
sp {
    ...
    -->
    (write (float (+ 2 3))) }
```

ifeq — Conditionally return a symbol. This function takes four arguments. It returns the third argument if the first two are equal and the fourth argument otherwise. Note that symbols of different types will always be considered unequal. For example, 1.0 and 1 will be unequal because the first is a float and the second is an integer.

```
sp {example-rule
    (state <s> ^a <a> ^b <b>)
    ...
    -->
    (write (ifeq <a> <b> equal not-equal)) }
```

3.3.6.10   Generating and manipulating symbols

A new symbol (an identifier) is generated on the right-hand side of a production whenever a previously unbound variable is used. This section describes other ways of generating and manipulating symbols on the right-hand side.

timestamp — This function returns a symbol whose print name is a representation of the current date and time.

For example:

```
sp {
    ...
    -->
    (write (timestamp)) }
```

When this production fires, it will print out a representation of the current date and time, such as:

```
soar> run 1 e
8/1/96-15:22:49
```

make-constant-symbol — This function returns a new constant symbol guaranteed to be different from all symbols currently present in the system. With no arguments, it returns a symbol whose name starts with "`constant`". With one or more arguments, it takes those argument symbols, concatenates them, and uses that as the prefix for the new symbol. (It may also append a number to the resulting symbol, if a symbol with that prefix as its name already exists.)

```
sp {
    ...
    -->
    (<s> ^new-symbol (make-constant-symbol)) }
```

When this production fires, it will create an augmentation in working memory such as:

```
(S1 ^new-symbol constant5)
```

The production:

```
sp {
    ...
    -->
    (<s> ^new-symbol (make-constant-symbol <s> )) }
```

will create an augmentation in working memory such as:

```
(S1 ^new-symbol |S14|)
```

when it fires. The vertical bars denote that the symbol is a constant, rather than an identifier; in this example, the number 4 has been appended to the symbol S1.

This can be particularly useful when used in conjunction with the `timestamp` function; by using `timestamp` as an argument to `make-constant-symbol`, you can get a new symbol that is guaranteed to be unique. For example:

```
sp {
    ...
    -->
    (<s> ^new-symbol (make-constant-symbol (timestamp))) }
```

When this production fires, it will create an augmentation in working memory such as:

```
(S1 ^new-symbol 8/1/96-15:22:49)
```

capitalize-symbol — Given a symbol, this function returns a new symbol with the
first character capitalized. This function is provided primarily for text output, for
example, to allow the first word in a sentence to be capitalized.

```
(capitalize-symbol foo)
```

concat — Given an arbitrary number of symbols, this function concatenates them to-
gether into a single constant symbol. For example,

```
sp {example
    (state <s> ^type state)
    -->
    (<s> ^name (concat foo bar (+ 2 4))) }
```

After this rule fires, the WME (S1 ^name foobar6) will be added.

3.3.6.11   User-defined functions and interface commands as RHS actions

Any function which has a certain function signature may be registered with the Kernel and
called as a RHS function. The function must have the following signature:

```
std::string MyFunction(smlRhsEventId id, void* pUserData, Agent* pAgent,
                char const* pFunctionName, char const* pArgument);
```

The Tcl and Java interfaces have similar function signatures. Any arguments passed to the
function on the RHS of a production are concatenated and passed to the function in the
pArgument argument.

Such a function can be registered with the kernel via the client interface by calling:

```
Kernel::AddRhsFunction(char const* pRhsFunctionName, RhsEventHandler
                handler, void* pUserData);
```

The exec and cmd functions are used to call user-defined functions and interface commands
on the RHS of a production.

exec — Used to call user-defined registered functions. Any arguments are concatenated
without spaces. For example, if <o> is bound to x, then

```
sp {
   ...
   -->
   (exec MakeANote <o> 1) }
```

will call the user-defined `MakeANote` function with the argument "x1".

The return value of the function, if any, may be placed in working memory or passed to another RHS function. For example, the log of a number `<x>` could be printed this way:

```
sp {
   ...
   -->
   (write |The log of | <x> | is: | (exec log(<x>))|) }
```

where "`log`" is a registered user-defined function.

cmd — Used to call built-in Soar commands. Spaces are inserted between concatenated arguments. For example, the production

```
sp {
   ...
   -->
   (write (cmd print --depth 2 <s>)) }
```

will have the effect of printing the object bound to `<s>` to depth 2.

3.3.6.12   Controlling chunking

Chunking is described in Chapter 4.

The following two functions are provided as RHS actions to assist in development of Soar programs; they are not intended to correspond to any theory of learning in Soar. This functionality is provided as a development tool, so that learning may be turned off in specific problem spaces, preventing otherwise buggy behavior.

The `dont-learn` and `force-learn` RHS actions are to be used with specific settings for the `learn` command (see page 166.) Using the `learn` command, learning may be set to one of `on`, `off`, `except`, or `only`; learning must be set to `except` for the `dont-learn` RHS action to have any effect and learning must be set to `only` for the `force-learn` RHS action to have any effect.

dont-learn — When learning is set to `except`, by default chunks can be formed in all states; the `dont-learn` RHS action will cause learning to be turned off for the specified state.

```
sp {turn-learning-off
    (state <s> ^feature 1 ^feature 2 -^feature 3)
     -->
    (dont-learn <s>) }
```

The `dont-learn` RHS action applies when `learn` is set to `-except`, and has no effect when other settings for `learn` are used.

force-learn —   When learning is set to `only`, by default chunks are not formed in any state; the `force-learn` RHS action will cause learning to be turned on for the specified state.

```
sp {turn-learning-on
    (state <s> ^feature 1 ^feature 2 -^feature 3)
     -->
    (force-learn <s>) }
```

The `force-learn` RHS action applies when `learn` is set to `-only`, and has no effect when other settings for `learn` are used.

3.4   Impasses in Working Memory and in Productions

When the preferences in preference memory cannot be resolved unambiguously, Soar reaches an impasse, as described in Section 2.6:

- When Soar is unable to select a new operator (in the decision cycle), it is said to reach an operator impasse.

All impasses appear as states in working memory, where they can be tested by productions. This section describes the structure of state objects in working memory.

3.4.1   Impasses in working memory

There are four types of impasses.

Below is a short description of the four types of impasses. (This was described in more detail in Section 2.6 on page 23.)

1. *tie*: when there is a collection of equally eligible operators competing for the value of a particular attribute;
2. *conflict*: when two or more objects are better than each other, and they are not dominated by a third operator;
3. *constraint-failure*: when there are conflicting necessity preferences;
4. *no-change*: when the proposal phase runs to quiescence without suggesting a new operator.

The list below gives the seven augmentations that the architecture creates on the substate generated when an impasse is reached, and the values that each augmentation can contain:
`^type state`

`^impasse` Contains the impasse type: `tie`, `conflict`, `constraint-failure`, or `no-change`.

`^choices` Either `multiple` (for tie and conflict impasses), `constraint-failure` (for constraint-failure impasses), or `none` (for no-change impasses).

`^superstate` Contains the identifier of the state in which the impasse arose.

`^attribute` For multi-choice and constraint-failure impasses, this contains `operator`. For no-change impasses, this contains the attribute of the last decision with a value (`state` or `operator`).

`^item` For multi-choice and constraint-failure impasses, this contains all values involved in the tie, conflict, or constraint-failure. If the set of items that tie or conflict changes during the impasse, the architecture removes or adds the appropriate item augmentations without terminating the existing impasse.

`^item-count` For multi-choice and constraint-failure impasses, this contains the number of values listed under the item augmentation above.

`^non-numeric` For tie impasses, this contains all operators that do not have numeric indifferent preferences associated with them. If the set of items that tie changes during the impasse, the architecture removes or adds the appropriate non-numeric augmentations without terminating the existing impasse.

`^non-numeric-count` For tie impasses, this contains the number of operators listed under the non-numeric augmentation above.

`^quiescence` States are the only objects with `quiescence t`, which is an explicit statement that quiescence (exhaustion of the elaboration cycle) was reached in the superstate. If problem solving in the subgoal is contingent on quiescence having been reached, the substate should test this flag. The side-effect is that no chunk will be built if it depended on that test. See Section 4.1 on page 73 for details. This attribute can be ignored when learning is turned off.

Knowing the names of these architecturally defined attributes and their possible values will help you to write productions that test for the presence of specific types of impasses so that you can attempt to resolve the impasse in a manner appropriate to your program. Many of the default productions in the `demos/defaults` directory of the Soar distribution provide means for resolving certain types of impasses. You may wish to make use of some of all of these productions or merely use them as guides for writing your own set of productions to respond to impasses.

Examples

The following is an example of a substate that is created for a tie among three operators:

```
(S12 ^type state ^impasse tie ^choices multiple ^attribute operator
     ^superstate S3 ^item O9 O10 O11 ^quiescence t)
```

The following is an example of a substate that is created for a no-change impasse to apply
an operator:

```
(S12 ^type state ^impasse no-change ^choices none ^attribute operator
     ^superstate S3 ^quiescence t)
(S3 ^operator O2)
```

3.4.2   Testing for impasses in productions

Since states appear in working memory, they may also be tested for in the conditions of
productions.

For example, the following production tests for a constraint-failure impasse on the top-level
state.

```
sp {default*top-goal*halt*operator*failure
    "Halt if no operator can be selected for the top goal."
    :default
    (state <ss> ^impasse constraint-failure ^superstate <s>)
    (<s> ^superstate nil)
-->
    (write (crlf) |No operator can be selected for top goal.| )
    (write (crlf) |Soar must halt.| )
    (halt)
}
```

3.5   Soar I/O: Input and Output in Soar

Many Soar users will want their programs to interact with a real or simulated environment.
For example, Soar programs could control a robot, receiving sensory *inputs* and sending
command *outputs*. Soar programs might also interact with simulated environments, such as
a flight simulator. The mechanisms by which Soar receives inputs and sends outputs to an
external process is called *Soar I/O*.

This section describes how input and output are represented in working memory and in
productions. The details of creating and registering the input and output functions for Soar
are beyond the scope of this manual, but they are described in the *SML Quick Start Guide*.
This section is provided for the sake of Soar users who will be making use of a program that
has already been implemented, or for those who would simply like to understand how I/O
is implemented in Soar.

3.5.1 Overview of Soar I/O

When Soar interacts with an external environment, it must make use of mechanisms that allow it to receive input from that environment and to effect changes in that environment. An external environment may be the real world or a simulation; input is usually viewed as Soar's perception and output is viewed as Soar's motor abilities.

Soar I/O is accomplished via *input functions* and *output functions*. Input functions are called at the *start* of every execution cycle, and add elements directly to specific input structures in working memory. These changes to working memory may change the set of productions that will fire or retract. Output functions are called at the *end* of every execution cycle and are processed in response to changes to specific output structures in working memory. An output function is called only if changes have been made to the output-link structures in working memory.

The structures for manipulating input and output in Soar are linked to a predefined attribute of the top-level state, called the `io` attribute. The `io` attribute has substructure to represent sensor inputs from the environment called *input links*; because these are represented in working memory, Soar productions can match against input links to respond to an external situation. Likewise, the `io` attribute has substructure to represent motor commands, called *output links*. Functions that execute motor commands in the environment use the values on the output links to determine when and how they should execute an action. Generally, input functions create and remove elements on the input link to update Soar's perception of the environment. Output functions respond to values of working memory elements that appear on Soar's output link strucure.

3.5.2 Input and output in working memory

All input and output is represented in working memory as substructure of the `io` attribute of the top-level state. By default, the architecture creates an `input-link` attribute of the `io` object and an `output-link` attribute of the io object. The values of the `input-link` and `output-link` attributes are identifiers whose augmentations are the complete set of input and output working memory elements, respectively. Some Soar systems may benefit from having multiple input and output links, or that use names which are more descriptive of the input or output function, such as `vision-input-link`, `text-input-link`, or `motor-output-link`. In addition to providing the default `io` substructure, the architecture allows users to create multiple input and output links via productions and I/O functions. Any identifiers for `io` substructure created by the user will be assigned at run time and are not guaranteed to be the same from run to run. Therefore users should always employ variables when referring to input and output links in productions.

Suppose a blocks-world task is implemented using a robot to move actual blocks around, with a camera creating input to Soar and a robotic arm executing command outputs. The camera image might be analyzed by a separate vision program; this program could have as its output the locations of blocks on an xy plane. The Soar input function could take the output from the vision program and create the following working memory elements on
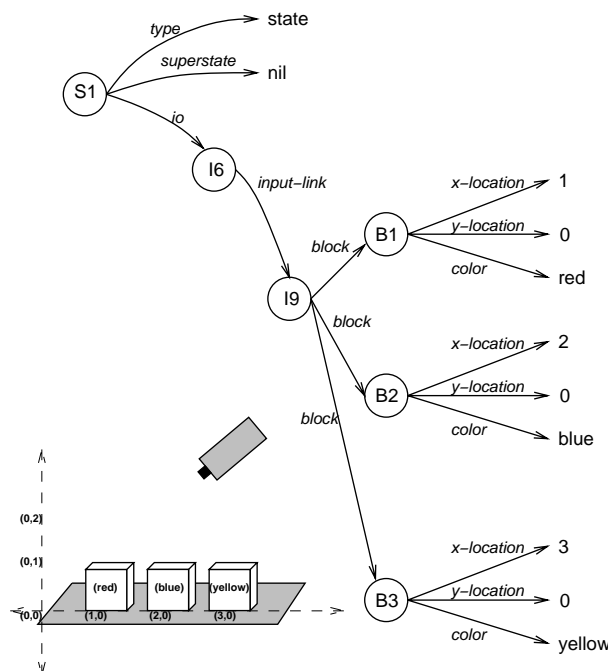
Figure 3.3: An example portion of the input link for the blocks-world task.

the input link (all identifiers are assigned at runtime; this is just an example of possible bindings):

```
(S1 ^io I1)          [A]
(I1 ^input-link I2)  [A]
(I2 ^block B1)
(I2 ^block B2)
(I2 ^block B3)
(B1 ^x-location 1)
(B1 ^y-location 0)
(B1 ^color red)
(B2 ^x-location 2)
(B2 ^y-location 0)
(B2 ^color blue)
(B3 ^x-location 3)
(B3 ^y-location 0)
(B3 ^color yellow)
```

The '[A]' notation in the example is used to indicate the working memory elements that are created by the architecture and not by the input function. This configuration of blocks corresponds to all blocks on the table, as illustrated in the initial state in Figure 2.2.

Then, during the Apply Phase of the execution cycle, Soar productions could respond to an operator, such as "move the red block ontop of the blue block" by creating a structure on the output link, such as:
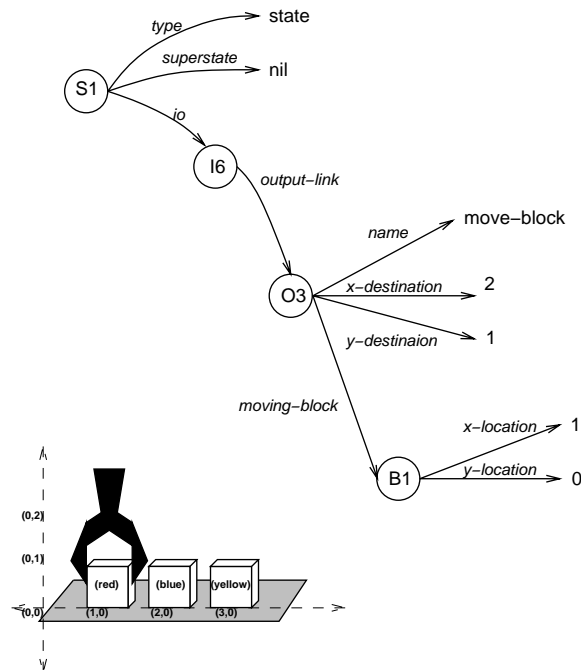
Figure 3.4: An example portion of the output link for the blocks-world task.

```
(S1 ^io I1)          [A]
(I1 ^output-link I3) [A]
(I3 ^name move-block)
(I3 ^moving-block B1)
(I3 ^x-destination 2)
(I3 ^y-destination 1)
(B1 ^x-location 1)
(B1 ^y-location 0)
(B1 ^color red)
```

The '[A]' notation is used to indicate the working memory elements that are created by the architecture and not by productions. An output function would look for specific structure in this output link and translate this into the format required by the external program that controls the robotic arm. Movement by the robotic arm would lead to changes in the vision system, which would later be reported on the input-link.

Input and output are viewed from Soar's perspective. An *input function* adds or deletes augmentations of the `input-link` providing Soar with information about some occurrence external to Soar. An *output function* responds to substructure of the `output-link` produced by production firings, and causes some occurrence external to Soar. Input and output occur through the `io` attribute of the top-level state exclusively.

Structures placed on the input-link by an input function remain there until removed by an input function. During this time, the structure continues to provide support for any production that has matched against it. The structure does *not* cause the production to

rematch and fire again on each cycle as long as it remains in working memory; to get the
production to refire, the structure must be removed and added again.

3.5.3   Input and output in production memory

Productions involved in *input* will test for specific attributes and values on the input-link,
while productions involved in *output* will create preferences for specific attributes and values
on the output link. For example, a simplified production that responds to the vision input
for the blocks task might look like this:

```
sp {blocks-world*elaborate*input
    (state <s> ^io.input-link <in>)
    (<in> ^block <ib1>)
    (<ib1> ^x-location <x1> ^y-location <y1>)
    (<in> ^block {<ib2> <> <ib1>})
    (<ib2> ^x-location <x1> ^y-location {<y2> > <y1>})
    -->
    (<s> ^block <b1>)
    (<s> ^block <b2>)
    (<b1> ^x-location <x1>  ^y-location <y1> ^clear no)
    (<b2> ^x-location <x1>  ^y-location <y2> ^above <b1>)
}
```

This production "copies" two blocks and their locations directly to the top-level state. It
also adds information about the relationship between the two blocks. The variables used for
the blocks on the RHS of the production are deliberately different from the variable name
used for the block on the input-link in the LHS of the production. If the variable were the
same, the production would create a link into the structure of the input-link, rather than
copy the information. The attributes `x-location` and `y-location` are assumed to be values
and not identifiers, so the same variable names may be used to do the copying.

A production that creates wmes on the output-link for the blocks task might look like this:

```
sp {blocks-world*apply*move-block*send-output-command
    (state <s> ^operator <o> ^io.output-link <out>)
    (<o> ^name move-block ^moving-block <b1> ^destination <b2>)
    (<b1> ^x-location <x1> ^y-location <y1>)
    (<b2> ^x-location <x2> ^y-location <y2>)
    -->
    (<out> ^move-block <b1>
           ^x-destination <x2> ^y-destination (+ <y2> 1))
}
```

This production would create substructure on the output-link that the output function could
interpret as being a command to move the block to a new location.

Chapter 4

Chunking

Chunking is Soar's mechanism to learn new procedural knowledge. Chunking creates productions, called *chunks*, that summarize the processing required to produce the results of subgoals. When a chunk is built, it is added to production memory, where it will be matched in similar situations, avoiding the need for the subgoal. Chunks are created only when results are formed in subgoals; since most Soar programs are continuously subgoaling and returning results to higher-level states, chunks are typically created continuously as Soar runs.

This chapter begins with a discussion of when chunks are built (Section 4.1 below), followed by a detailed discussion of how Soar determines a chunk's conditions and actions (Section 4.2). Sections 4.4 through 4.5 examine the construction of chunks in further detail. Section 4.6 explains how and why chunks are prevented from matching with the WME's that led to their creation. Section 4.7 reviews the problem of overgeneral chunks.

4.1 Chunk Creation

Several factors govern when chunks are built. Soar chunks the results of every subgoal, *unless* one of the following conditions is true:

1. Learning is `off`. (See Section 8.4 on page 166 for details of `learn` used to turn learning off.)

   Learning can be set to `on` or `off`. When `learn` is `on` chunks are built. When `learn` is `off`, chunks are not built.

2. Learning is set to `bottom-up` and a chunk has already been built for a subgoal of the state that generated the results. (See Section 8.4 on page 166 for details of `learn` used to set learning to bottom-up.)

   With bottom-up learning, chunks are learned only in states in which no subgoal has yet generated a chunk. In this mode, chunks are learned only for the "bottom" of the subgoal hierarchy and not the intermediate levels. With experience, the subgoals at the

73

bottom will be replaced by the chunks, allowing higher level subgoals to be chunked.[1]

3. The learning flag `through-local-negations` is disabled, and the result is dependent on a test for the negation of a subgoal WME. Testing a local negation can result in an overgeneral chunk (see Section 4.7 on page 80). In this mode, such chunks are not created.

4. The chunk duplicates a production or chunk already in production memory. In some rare cases, a duplicate production will not be detected because the order of the conditions or actions is not the same as an existing production.

5. The augmentation, $^\wedge$`quiescence t`, of the substate that produced the result is backtraced through.

   This mechanism is motivated by the *chunking from exhaustion* problem, where the results of a subgoal are dependent on the exhaustion of alternatives (see Section 4.7 on page 80). If this substate augmentation is encountered when determining the conditions of a chunk, then no chunk will be built for the currently considered action. This is recursive, so that if an un-chunked result is relevant to a second result, no chunk will be built for the second result. This does not prevent the creation of a chunk that would include $^\wedge$`quiescence t` as a condition.

6. Learning has been temporarily turned off via a call to the `dont-learn` production action (described on page 65 in Section 3.3.6.12).

   This capability is provided for debugging and system development, and it is not part of the theory of Soar.

If a result is to be chunked, Soar builds the chunk *as soon as the result is created*, rather than waiting until subgoal termination.

4.2   Determining Conditions and Actions

Chunking is an experience-based learning mechanism that summarizes as productions the problem solving that occurs within a state. In order to maintain a history of the processing to be used for chunking, Soar builds a *trace* of the productions that fire in the subgoals. This section describes how the relevant actions are determined, how information is stored in a trace, and finally, how the trace and the actions together determine the conditions for the chunk.

In order for the chunk to apply at the appropriate time, its conditions must test exactly those working memory elements that were necessary to produce the results of the subgoal. Soar computes a chunk's conditions based on the productions that fire in the subgoal, beginning with the results of the subgoal, and then *backtracing* through the productions that created each result. It recursively backtraces through the working memory elements that matched

---

[1]For some tasks, bottom-up chunking facilitates modelling power-law speedups, although its long-term theoretical status is problematic.

the conditions of the productions, finding the actions that led to the WME's creation, etc., until conditions are found that test elements that are linked to a superstate.

4.2.1 Determining a chunk's actions

A chunk's actions are built from the results of a subgoal. A *result* is any working memory element created in the substate that is linked to a superstate. A working memory element is linked if its identifier is either the value of a superstate WME, or the value of an augmentation for an object that is linked to a superstate.

The results produced by a single production firing are the basis for creating the actions of a chunk. A new result can lead to other results by linking a superstate to a WME in the substate. This WME may in turn link other WMEs in the substate to the superstate, making them results. Therefore, the creation of a single WME that is linked to a superstate can lead to the creation of a large number of results. All of the newly created results become the basis of the chunk's actions.

4.2.2 Tracing the creation and reference of working memory elements

Soar automatically maintains information on the creation of each working memory element in every state. When a production fires, a trace of the production is saved with the appropriate state. A *trace* is a list of the working memory elements matched by the production's conditions, together with the actions created by the production. The appropriate state is the most recently created state (i.e., the state *lowest* in the subgoal hierarchy) that occurs in the production's matched working memory elements.

Recall that when a subgoal is created, the $^\wedge$item augmentation lists all values that lead to the impasse. Chunking is complicated by the fact that the $^\wedge$`item` augmentation of the substate is created by the architecture and not by productions. Backtracing cannot determine the cause of these substate augmentations in the same way as other working memory elements. To overcome this, Soar maps these augmentations onto the acceptable preferences for the operators in the $^\wedge$`item` augmentations.

Negated conditions

Negated conditions are included in a trace in the following way: when a production fires, its negated conditions are fully instantiated with its variables' appropriate values. This instantiation is based on the working memory elements that matched the production's positive conditions. If the variable is not used in any positive conditions, such as in a conjunctive negation, a dummy variable is used that will later become a variable in a chunk.

If the identifier used to instantiate a negated condition's identifier field is linked to the superstate, then the instantiated negated condition is added to the trace as a negated condition.

In all other cases, the negated condition is ignored because the system cannot determine why a working memory element *was not* produced in the subgoal and thus allowed the production to fire. Ignoring these negations of conditions internal to the subgoal may lead to overgeneralization in chunking (see Section 4.7 on page 80).

4.2.3   Determining a chunk's conditions

The conditions of a chunk are determined by a dependency analysis of production traces — a process called *backtracing*. For each instantiated production that creates a subgoal result, backtracing examines the production trace to determine which working memory elements were matched. If a matched working memory element is linked to a superstate, it is included in the chunk's conditions. If it is not linked to a superstate, then backtracing recursively examines the trace of the production that created the working memory element. Thus, backtracing begins with a subgoal result, traces backwards through all working memory elements that were used to produce that result, and collects all of the working memory elements that are linked to a superstate. This method ignores when the working memory elements were created, thus allowing the conditions of one chunk to test the results of a chunk learned earlier in the subgoal. The user can observe the backtracing process by setting setting backtracing on, using the watch command: `watch backtracing -on` (see Section 8.3 on page 148). This prints out a trace of the conditions as they are collected.

Given that results can be created at any point during a subgoal, it is possible for one result to be relevant to another result. Whether or not the first result is included in the chunk for the second result depends on the links that were used to match the first result in the subgoal. If the elements are linked to the superstate, they are included as conditions. If the elements are not linked to the superstate, then the result is traced through. In some cases, there may be more than one set of links, so it is possible for a result to be both backtraced through, and included as a condition.

Some operator evaluations productions, i.e. rules that did not actually produce the result but did lead to the selection of the operator, can also participate in backtracing. These rules produce a set of desirability preferences that we call the context-dependent preference set (CDPS). The rules that produced the preferences in the CDPS are backtraced through to potentially add additional conditions to the chunk.

Not all operator evaluation preferences are included in the CDPS. Traditionally the argument has been that operator evaluation preferences should affect only the *efficiency* and not the *correctness* of problem solving, and therefore are not necessary to produce the results. As a result, desirability preferences, namely `reject`, (`better`, `worse`, `best`, `worst`, or `indifferent`), were not included in the CDPS and hence would not be backtraced through when creating chunks. (The exception to this is that if the desirability or `reject` preference is a *result* of a subgoal, it will be in the chunk's actions.) The philosophy behind this was that desirability and reject preferences should be used only as search control for choosing between legal alternatives and should not be used to guarantee the correctness of the problem solving. In contrast, necessity preferences (`require` or `prohibit`) were used to enforce the correctness of problem solving, and the productions that created these preferences were

always included in backtracing.

This is still the default behavior. Only prohibit and require preference are included in the CDPS. In some case, though, this may lead to over-general chunks (see Section 4.7 on page 80 for details). As of Soar 9.3.3, there is a new learning option (see Section 8.4 on page 166 for details) that will include a broader range of desirability preferences in the CDPS. The rules that determine which desirability preferences are included in the CDPS – it depends on whether a particular desirability preferences played a role in the selection of the operator – are documented in Section 4.3 on page 77.

4.3   The Context-Dependent Preference Set

As described in the beginning of this chapter, chunking summarizes the processing required to produce the results of sub-goals. Traditionally, the philosophy behind how an agent should be designed was that the path of operator selections and applications from an initial state in a sub-state to a result would always have all necessary tests in the operator proposal conditions and any goal test, so only those items would need to be summarized. The idea was that in a properly designed agent, a sub-state's operator evaluation preferences lead to a more efficient search of the space but do not influence the correctness of the result. As a result, the knowledge used by rules that produce such evaluation preferences should not be included in any chunks produced from that sub-state.

In practice, however, a Soar program can be written so that search control does affect the correctness of search. A few examples can be found in Section 4.7 on page 80. Moreover, reinforcement learning rules are often used to direct the search to those states that provide more reward, consequently making the idea of correctness much fuzzier. As a result, there may be cases when it is useful to encode goal-attainment knowledge into operator evaluation rules. Unfortunately, chunks created in such a problem space will be overgeneral because important parts of the superstate that were tested by operator evaluation rules do not appear as conditions in the chunks that summarize the processing in that problem state. The context-dependent preference set is a way to address this issue.

*The context-dependent preference set (CDPS) is the set of **relevant** operator evaluation preferences that led to the selection of an operator in a sub-goal.* Whenever Soar creates either a justification or a chunk, it recursively backtraces through rules to determine what conditions to include in the final chunk or justification. With the CDPS, not only does Soar backtrace through each rule that created a matched working memory element, but it also backtraces through every rule that created a preferences in the CDPS for the selected operator when that rule fired. By backtracing through that additional set of preferences at each step of the backtrace, an agent will create more specific chunks that incorporate the goal-attainment knowledge encoded in the operator evaluation rules.

All necessity preferences, i.e. prohibit and require preferences, are always included in the CDPS since they inherently encode the correctness of whether an operator is applicable in a problem space. In contrast, desirability preferences (rejects, betters, worses, bests, worsts and indifferents) are included depending on the role they play in the selection of the operator

(and whether the add-desirability-prefs learn setting is active).

How Soar determines which of those preferences to include in the CDPS is determined by the preference semantics it uses to choose an operator. During the decision phase, operator preferences are evaluated in a sequence of seven steps or filters, in an effort to select a single operator. Each step handles a specific type of preference. *As the preference semantics are applied at each step to incrementally filter the candidates to a potential selected operator, the CDPS is incrementally built up based on the preferences that were instrumental in applying each filter.*

The following outline describes the logic that happens at each step. For a more detailed description of the various filters (but not the CDPS) see Appendix D on page 227. Note that depending on the set of preferences being processed, impasses may occur at some of these stages, in which case, no operator is selected and the CDPS is emptied. Moreover, if the candidate set is reduced to zero or one, the decision process will exit with a finalized CDPS. For simplicity's sake, this explanation assumes that there are no impasses and the decision process continues.

- Require Filter:

  ⋆ If an operator is selected based on a require preference, that preference is added to the CDPS. The logic behind this step is straightforward, the require preference directly resulted in the selection of the operator.

- Prohibit/Reject Filters:

  ⋆ If there exists at least one prohibit or reject preference, all prohibit and reject preferences for the eliminated candidates are added to the CDPS. The logic behind this stage is that the conditions that led to the exclusion of the prohibited and rejected candidates is what allowed the final operator to be selected from among that particular set of surviving candidates.

- Better/Worse Filter:

  ⋆ For every candidate that is not worse than some other candidate, add all better/worse preferences involving the candidate.

- Best Filter:

  ⋆ Add any best preferences for remaining candidates to the CDPS.

- Worst Filter:

  ⋆ If any remaining candidate has a worst preference which leads to that candidate being removed from consideration, that worst preference is added to the CDPS. Again, the logic is that the conditions that led to that candidate not being selected allowed the final operator to be chosen.

- Indifferent Filter:

    ⋆ This is the final stage, so the operator is now selected based on the agent's exploration policy. How indifferent preferences are added to the CDPS depends on whether any numeric indifferent preferences exist.

        1. If there exists at least one numeric indifferent preference, then every numeric preferences for the winning candidate is added to the CDPS. There can be multiple such preferences. Moreover, all binary indifferent preferences between that winning candidate and candidates without a numeric preference are added.
        2. If all indifferent preferences are non-numeric, then any unary indifferent preferences for the winning candidate are added to the CDPS. Moreover, all binary indifferent preferences between that winning candidate and other candidates are added.

    ⋆ The logic behind adding binary indifferent preferences between the selected operator and the other final candidates is that those binary indifferent preferences prevented a tie impasse and allowed the final candidate to be chosen by the exploration policy from among those mutually indifferent preferences.

Note that there may be cases where two or more rules create the same type of preference for a particular candidate. In those cases, only the first preference encountered is added to the CDPS. Adding all of them can produce over-specific chunks. It may still be possible to learn similar chunks with those other preferences if the agent sub-goals again in a similar context.

As of version 9.3.3, desirability preferences will not be added to the CDPS by default. The setting must be turned on via the learn command's add-desirability-prefs setting. See Section 8.4 on page 166 for more information. Necessity preferences will always be added to the CDPS regardless of setting.

Note that the CDPS also affects the conditions of justifications, so the add-desirability-prefs setting does have an effect on the agent even if learning is turned off.

4.4 Variablizing Identifiers

Chunks are constructed by examining the traces, which include working memory elements and operator preferences. To achieve any useful generality in chunks, identifiers of actual objects must be replaced by variables when the chunk is created; otherwise chunks will only ever fire when the exact same objects are matched. However, a constant value is never variablized; the actual value always appears directly in the chunk.

When a chunk is built, all occurrences of the same identifier are replaced with the same variable. This can lead to an overspecific chunk, when two variables are forced to be the same in the chunk, even though distinct variables in the original productions just happened to match the same identifier.

A chunk's conditions are also constrained by any not-equal (<>) tests for pairs of indentifiers used in the conditions of productions that are included in the chunk. These tests are saved in the production traces and then added in to the chunk.

4.5   Ordering Conditions

Since the efficiency of the Rete matcher depends heavily upon the order of a production's conditions, the chunking mechanism attempts to write the chunk's conditions in the most favorable order. At each stage, the condition-ordering algorithm tries to determine which eligible condition, if placed next, will lead to the fewest number of partial instantiations when the chunk is matched. A condition that matches an object with a multi-valued attribute will lead to multiple partial instantiations, so it is generally more efficient to place these conditions later in the ordering.

This is the same process that internally reorders the conditions in user-defined productions, as mentioned briefly in Section 2.3.1.

4.6   Inhibition of Chunks

When a chunk is built, it may be able to match immediately with the same working memory elements that participated in its creation. If the production's actions include preferences for new operators, the production would immediately fire and create a preference for a new operator, which duplicates the operator preference that was the original result of the subgoal. To prevent this, *inhibition* is used. This means that each production that is built during chunking is considered to have already fired with the instantiation of the exact set of working memory elements used to create it. This does not prevent a newly learned chunk from matching other working memory elements that are present and firing with those values.

4.7   Problems that May Arise with Chunking

One of the weaknesses of Soar is that chunking can create overgeneral productions that apply in inappropriate situations, or overspecific productions that will never fire. These problems arise when chunking cannot accurately summarize the processing that led to the creation of a result. Below is a description of five known problems in chunking.

4.7.1   Using search control to determine correctness

Overgeneral chunks can be created if a result of problem solving in a subgoal is dependent on search-control knowledge. Recall that desirability preferences, such as `better`, `best`, and `worst`, are not included in the traces of problem solving used in chunking (Section 4.2

on page 74). In theory, these preferences do not affect the validity of search. In practice, however, a Soar program can be written so that search control *does* affect the correctness of search. Here are two examples:

1. Some of the tests for correctness of a result are included in productions that prefer operators that will produce correct results. The system will work correctly only when those productions are loaded.

2. An operator is given a worst preference, indicating that it should be used only when all other options have been exhausted. Because of the semantics of worst, this operator will be selected after all other operators; however, if this operator then produces a result that is dependent on the operator occurring after all others, this fact will not be captured in the conditions of the chunk.

In both of these cases, part of the test for producing a result is *implicit* in search control productions. This move allows the explicit state test to be simpler because any state to which the test is applied is guaranteed to satisfy some of the requirements for success. However, chunks created in such a problem space will be overgeneral because the implicit parts of the state test do not appear as conditions.

Solution: To avoid this problem, the context-dependent preference set can be used to include any search-control knowledge that incorporates goal-attainment knowledge in the backtrace used by chunking; this will produce chunks with additional conditions that resulted from the search-control preferences, thereby avoiding overgenerality. By default, all necessity preferences (`require` and `prohibit`) are included in the CDPS. If goal-attainment knowledge is also included in desirability prefs, such as `better`, `worse`, `best`, `worst`, or `indifferent`, the add-desirability-prefs setting of the CDPS (Section 8.4 on 166) can be turned on to incorporate them into the backtrace and hence any resulting chunks. The rules that determine which desirability preferences are included in the CDPS – it depends on whether a particular desirability preferences played an important role in the selection of an operator that ultimately led to the result – are documented in Section 4.3 on 77.

4.7.2   Testing for local negated conditions

Overgeneral chunks can be created when negated conditions test for the absence of a working memory element that, if it existed, would be local to the substate. Chunking has no mechanism for determining *why* a given working memory element does not exist, and thus a condition that occurred in a production in the subgoal is not included in the chunk. For example, if a production tests for the absence of a local flag, and that flag is copied down to the substate from a superstate, then the chunk should include a test that the flag in the superstate does not exist. Unfortunately, it is computationally expensive to determine why a given working memory element does not exist. Chunking only includes negated tests if they test for the absence of superstate working memory elements.

Solution: To avoid using negated conditions for local data, the local data can be made a result by attaching it to the superstate. This increases the number of chunks learned, but a negated condition for the superstate can be used that leads to correct chunks.

Alternatively, Soar's learning mode can be set to reject chunks when the backtrace encounters a local negation, by setting `learn --through-local-negations disable`. There are many cases where local negations are safe to ignore (and hence this mode reduces performance), but it can substantially reduce the number of overgeneral chunks in big agents (and aid in debugging).

4.7.3   Testing for the substate

Overgeneral chunks can be created if a result of a subgoal is dependent on the creation of an impasse within the substate. For example, processing in a subgoal may consist of exhaustively applying all the operators in the problem space. If so, then a convenient way to recognize that all operators have applied and processing is complete is to wait for a state no-change impasse to occur. When the impasse occurs, a production can test for the resulting substate and create a result for the original subgoal. This form of state test builds overgeneral chunks because no pre-existing structure is relevant to the result that terminates the subgoal. The result is dependent only on the existence of the substate within a substate.

Solution: The current solution to this problem is to allow the problem solving to signal the architecture that the test for a substate is being made. The signal used by Soar is a test for the $^\wedge$`quiescence t` augmentation of the subgoal. The chunking mechanism recognizes this test and does not build a chunk when it is found in a backtrace of a subgoal. The history of this test is maintained, so that if the result of the substate is then used to produce further results for a superstate, no higher chunks will be built. However, if the result is used as search control (it is a desirability preference), then it does not prevent the creation of chunks because the original result is not included in the backtrace. If the $^\wedge$`quiescence t` being tested is connected to a superstate, it will not inhibit chunking and it will be included in the conditions of the chunk.

4.7.4   Mapping multiple superstate WMEs to one local WME

An agent may have several rule instantiations that match on different structures in a superstate but create WMEs with the same attribute-value pairs in a substate. For example, there may be a rule that matches several WMEs in a superstate with the same multi-valued attribute and elaborates the local state with a WME indicating that at least one WME with that attribute exists. In these cases, the total effect of those rule firings will be collapsed into creating a single WME in the substate, because working memory is represented as a set. If this WME is then tested to create a result on the superstate, the chunk that is subsequently created will be overgeneral: While the original subgoal processing created only one result, the chunk will create a distinct result for each superstate structure originally tested. This is because the desired behavior cannot be reduced to a single rule.

Solution: If this type of behavior is needed, the single WME should go in the top state, so that the chunks built can similarly map multiple structures to one.

4.7.5 Revising the substructure of a previous result

This can occur when a subgoal creates a local structure, which is then linked to a superstate, becoming a result. A new WME added to this structure is also a result, as as it is linked to the superstate. However, if that WME is created with a rule that matches the local state only (not the superstate), Soar cannot build a chunk for the result, as it is unable to determine how the new WME is linked to the superstate.

For example, assume that an agent builds up a structure consisting of an identifier called ^`thing` attached to a substate, and then adds ^`property foo` as an augmentation to `thing`. If the agent now matches `thing` on the substate, and creates a WME on a superstate linked to the same identifier, that identifier, along with its augmentation ^`property foo`, becomes a result, and a chunk is formed. Now, if a rule in the subgoal adds another augmentation to the `thing` identifier, (^`property bar`, say), that augmentation will also be a result, as it is linked to an identifier which is linked to a superstate. However, if that rule matches the identifier through the substate, the chunking process cannot determine how it is linked to the superstate, and a chunk cannot be created.

Solution: If the substructure of a result must be revised, the rules that modify it should match the result through the superstate, not through the local state.

Chapter 5

Reinforcement Learning

Soar has a reinforcement learning (RL) mechanism that tunes operator selection knowledge based on a given reward function. This chapter describes the RL mechanism and how it is integrated with production memory, the decision cycle, and the state stack. We assume that the reader is familiar with basic reinforcement learning concepts and notation. If not, we recommend first reading *Reinforcement Learning: An Introduction* (1998) by Richard S. Sutton and Andrew G. Barto. The detailed behavior of the RL mechanism is determined by numerous parameters that can be controlled and configured via the **rl** command. Please refer to the documentation for that command in section 8.4 on page 175.

5.1 RL Rules

Soar's RL mechanism learns Q-values for state-operator[1] pairs. Q-values are stored as numeric indifferent preferences created by specially formulated productions called *RL rules*. RL rules are identified by syntax. A production is a RL rule if and only if its left hand side tests for a proposed operator, its right hand side creates a single numeric indifferent preference, and it is not a template rule (see 5.4.2). These constraints ease the technical requirements of identifying/updating RL rules and makes it easy for the agent programmer to add/maintain RL capabilities within an agent. We define an *RL operator* as an operator with numeric indifferent preferences created by RL rules.

The following is an RL rule:

```
sp {rl*3*12*left
   (state <s> ^name task-name
              ^x 3
              ^y 12
           ^operator <o> +)
```

---

[1] In this context, the term "state" refers to the state of the task or environment, not a state identifier. For the rest of this chapter, bold capital letter names such as **S1** will refer to identifiers and italic lowercase names such as s_1 will refer to task states.

```
   (<o> ^name move
     ^direction left)
-->
   (<s> ^operator <o> = 1.5)
}
```

Note that the LHS of the rule can test for anything as long as it contains a test for a proposed operator. The RHS is constrained to exactly one action: creating a numeric indifferent preference for the proposed operator.

The following are not RL rules:

```
sp {multiple*preferences
   (state <s> ^operator <o> +)
-->
   (<s> ^operator <o> = 5, >)
}
```

```
sp {variable*binding
    (state <s> ^operator <o> +
               ^value <v>)
-->
    (<s> ^operator <o> = <v>)
}
```

The first rule proposes multiple preferences for the proposed operator and thus does not comply with the rule format. The second rule does not comply because it does not provide a *constant* for the numeric indifferent preference value.

In the typical RL use case, the user intends for the agent to learn the best operator in each possible state of the environment. The most straightforward way to achieve this is to give the agent a set of RL rules, each matching exactly one possible state-operator pair. This approach is equivalent to a table-based RL algorithm, where the Q-value of each state-operator pair corresponds to the numeric indifferent preference created by exactly one RL rule.

In the more general case, multiple RL rules can match a single state-operator pair, and a single RL rule can match multiple state-operator pairs. all numeric indifferent preferences for an operator are summed when calculating the operator's Q-value[2]. In this context, RL rules can be interpreted more generally as binary features in a linear approximator of each state-operator pair's Q-value, and their numeric indifferent preference values their weights. In other words,

$$Q(s,a) = w_1\phi_2(s,a) + w_2\phi_2(s,a) + \ldots + w_n\phi_n(s,a)$$

where all RL rules in production memory are numbered $1 \ldots n$, $Q(s,a)$ is the Q-value of the state-operator pair (s,a), w_i is the numeric indifferent preference value of RL rule i,

---

[2] This is assuming the value of **numeric-indifferent-mode** is set to **sum**. In general, the RL mechanism only works correctly when this is the case, and we assume this case in the rest of the chapter. See page 173 for more information about this parameter.

$\phi_i(s, a) = 0$ if RL rule i does not match (s, a), and $\phi_i(s, a) = 1$ if it does. This interpretation allows RL rules to simulate a number of popular function approximation schemes used in RL such as tile coding and sparse coding.

5.2 Reward Representation

RL updates are driven by reward signals. In Soar, these reward signals are given to the RL mechanism through a working memory link called the **reward-link**. Each state in Soar's state stack is automatically populated with a **reward-link** structure upon creation. Soar will check this structure for a numeric reward signal for the last operator executed in the associated state at the beginning of every decision phase. Reward is also collected when the agent is halted or a state is retracted.

In order to be recognized, the reward signal must follow this pattern:

```
(<r1> ^reward <r2>)
(<r2> ^value [val])
```

where `<r1>` is the **reward-link** identifier, `<r2>` is some intermediate identifier, and `[val]` is any constant numeric value. Any structure that does not match this pattern is ignored. If there are multiple valid reward signals, their values are summed into a single reward signal. As an example, consider the following state:

```
(S1 ^reward-link R1)
  (R1 ^reward R2)
    (R2 ^value 1.0)
  (R1 ^reward R3)
    (R3 ^value -0.2)
```

In this state, there are two reward signals with values 1.0 and -0.2. They will be summed together for a total reward of 0.8 and this will be the value given to the RL update algorithm.

There are two reasons for requiring the intermediate identifier. The first is so that multiple reward signals with the same value can exist simultaneously. Since working memory is a set, multiple WMEs with identical values in all three positions (identifier, attribute, value) cannot exist simultaneously. Without an intermediate identifier, specifying two rewards with the same value would require a WME structure such as

```
(S1 ^reward-link R1)
  (R1 ^reward 1.0)
  (R1 ^reward 1.0)
```

which is invalid. With the intermediate identifier, the rewards would be specified as

```
(S1 ^reward-link R1)
  (R1 ^reward R2)
    (R2 ^value 1.0)
  (R1 ^reward R3)
```

```
    (R3 ^value 1.0)
```

which is valid. The second reason for requiring an intermediate identifier in the reward signal is so that the rewards can be augmented with additional information, such as their source or how long they have existed. Although this information will be ignored by the RL mechanism, it can be useful to the agent or programmer. For example:

```
(S1 ^reward-link R1)
  (R1 ^reward R2)
    (R2 ^value 1.0)
    (R2 ^source environment)
  (R1 ^reward R3)
    (R3 ^value -0.2)
    (R3 ^source intrinsic)
    (R3 ^duration 5)
```

The `(R2 ^source environment)`, `(R3 ^source intrinsic)`, and `(R3 ^duration 5)` WMEs are arbitrary and ignored by RL, but were added by the agent to keep track of where the rewards came from and for how long.

Note that the **reward-link** is not part of the **io** structure and is not modified directly by the environment. Reward information from the environment should be copied, via rules, from the **input-link** to the **reward-link**. Also note that when collecting rewards, Soar simply scans the **reward-link** and sums the values of all valid reward WMEs. The WMEs are not modified and no bookkeeping is done to keep track of previously seen WMEs. This means that reward WMEs that exist for multiple decision cycles will be collected multiple times.

5.3   Updating RL Rule Values

Soar's RL mechanism is integrated naturally with the decision cycle and performs online updates of RL rules. Whenever an RL operator is selected, the values of the corresponding RL rules will be updated. The update can be on-policy (Sarsa) or off-policy (Q-Learning), as controlled by the **learning-policy** parameter of the **rl** command. Let δ_t be the amount the Q-value of an RL operator changes in an update. For Sarsa, we have

$$\delta_t = \alpha \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right]$$

where

- $Q(s_t, a_t)$ is the Q-value of the state and chosen operator in decision cycle t.

- $Q(s_{t+1}, a_{t+1})$ is the Q-value of the state and chosen RL operator in the next decision cycle.

- r_{t+1} is the total reward collected in the next decision cycle.

- α and γ are the settings of the **learning-rate** and **discount-rate** parameters of the **rl** command, respectively.

Note that since δ_t depends on $Q(s_{t+1}, a_{t+1})$, the update for the operator selected in decision cycle t is not applied until the next RL operator is chosen. For Q-Learning, we have

$$\delta_t = \alpha \left[r_{t+1} + \gamma \max_{a \in A_{t+1}} Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$

where A_{t+1} is the set of RL operators proposed in the next decision cycle.

Finally, δ_t is divided by the number of RL rules comprising the Q-value for the operator and the numeric indifferent values for each RL rule is updated by that amount.

An example walkthrough of a Sarsa update with $\alpha = 0.3$ and $\gamma = 0.9$ (the default settings in Soar) follows.

1. In decision cycle t, an operator **O1** is proposed, and RL rules **rl-1** and **rl-2** create the following numeric indifferent preferences for it:

   ```
   rl-1: (S1 ^operator O1 = 2.3)
   rl-2: (S1 ^operator O1 =  -1)
   ```

   The Q-value for **O1** is $Q(s_t, \mathbf{O1}) = 2.3 - 1 = 1.3$.

2. **O1** is selected and executed, so $Q(s_t, a_t) = Q(s_t, \mathbf{O1}) = 1.3$.

3. In decision cycle $t + 1$, a total reward of 1.0 is collected on the **reward-link**, an operator **O2** is proposed, and another RL rule **rl-3** creates the following numeric indifferent preference for it:

   ```
   rl-3: (S1 ^operator O2 = 0.5)
   ```

   So $Q(s_{t+1}, \mathbf{O2}) = 0.5$.

4. **O2** is selected, so $Q(s_{t+1}, a_{t+1}) = Q(s_{t+1}, \mathbf{O2}) = 0.5$ Therefore,

   $$\delta_t = \alpha \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right] = 0.3 \times [1.0 + 0.9 \times 0.5 - 1.3] = 0.045$$

   Since **rl-1** and **rl-2** both contributed to the Q-value of **O1**, δ_t is evenly divided amongst them, resulting in updated values of

   ```
   rl-1: (<s> ^operator <o> = 2.3225)
   rl-2: (<s> ^operator <o> = -0.9775)
   ```

5. **rl-3** will be updated when the next RL operator is selected.

5.3.1 Gaps in Rule Coverage

Call an operator with numeric indifferent preferences an RL operator. The previous description had assumed that RL operators were selected in both decision cycles t and $t+1$. If the operator selected in $t+1$ is not an RL operator, then $Q(s_{t+1}, a_{t+1})$ would not be defined, and an update for the RL operator selected at time t will be undefined. We will call a sequence of one or more decision cycles in which RL operators are not selected between two decision cycles in which RL operators are selected a *gap*. Conceptually, it is desirable to use the temporal difference information from the RL operator after the gap to update the Q-value of the RL operator before the gap. There are no intermediate storage locations for these updates. Requiring that RL rules support operators at every decision can be difficult for agent programmers, particularly for operators that do not represent steps in a task, but instead perform generic maintenance functions, such as cleaning processed output-link structures.

To address this issue, Soar's RL mechanism supports automatic propagation of updates over gaps. For a gap of length n, the Sarsa update is

$$\delta_t = \alpha \left[\sum_{i=t}^{t+n} \gamma^{i-t} r_i + \gamma^{n+1} Q(s_{t+n+1}, a_{t+n+1}) - Q(s_t, a_t) \right]$$

and the Q-Learning update is

$$\delta_t = \alpha \left[\sum_{i=t}^{t+n} \gamma^{i-t} r_i + \gamma^{n+1} \max_{a \in A_{t+n+1}} Q(s_{t+n+1}, a) - Q(s_t, a_t) \right]$$

Note that rewards will still be collected during the gap, but they are discounted based on the number of decisions they are removed from the initial RL operator.

Gap propagation can be disabled by setting the **temporal-extension** parameter of the **rl** command to **off**. When gap propagation is disabled, the RL rules preceding a gap are updated using $Q(s_{t+1}, a_{t+1}) = 0$. The **rl** setting of the **watch** command (see Section 8.3 on page 148) is useful in identifying gaps.

5.3.2 RL and Substates

When an agent has multiple states in its state stack, the RL mechanism will treat each substate independently. As mentioned previously, each state has its own **reward-link**. When an RL operator is selected in a state **S**, the RL updates for that operator are only affected by the rewards collected on the **reward-link** for **S** and the Q-values of subsequent RL operators selected in **S**.

The only exception to this independence is when a selected RL operator forces an operator-no-change impasse. When this occurs, the number of decision cycles the RL operator at the superstate remains selected is dependent upon the processing in the impasse state. Consider the operator trace in Figure 5.1.
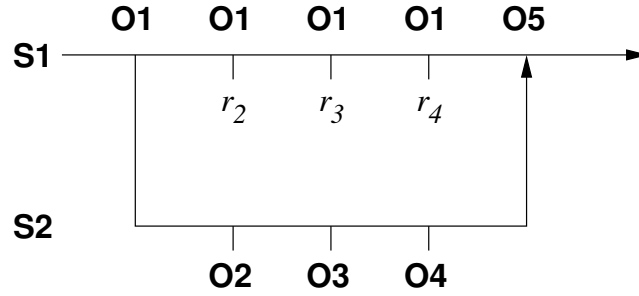
Figure 5.1: Example Soar substate operator trace.

- At decision cycle 1, RL operator **O1** is selected in **S1** and causes an operator-no-change impass for three decision cycles.

- In the substate **S2**, operators **O2**, **O3**, and **O4** are selected and applied sequentially.

- Meanwhile in **S1**, reward values r_2, r_3, and r_4 are put on the **reward-link** sequentially.

- Finally, the impasse is resolved by **O4**, the proposal for **O1** is retracted, and RL operator **O5** is selected in **S1**.

In this scenario, only the RL update for $Q(s_1, \mathbf{O1})$ will be different from the ordinary case. Its value depends on the setting of the **hrl-discount** parameter of the **rl** command. When this parameter is set to the default value **on**, the rewards on **S1** and the Q-value of **O5** are discounted by the number of decision cycles they are removed from the selection of **O1**. In this case the update for $Q(s_1, \mathbf{O1})$ is

$$\delta_1 = \alpha \left[r_2 + \gamma r_3 + \gamma^2 r_4 + \gamma^3 Q(s_5, \mathbf{O5}) - Q(s_1, \mathbf{O1}) \right]$$

which is equivalent to having a three decision gap separating **O1** and **O5**.

When **hrl-discount** is set to **off**, the number of cycles **O1** has been impassed will be ignored. Thus the update would be

$$\delta_1 = \alpha \left[r_2 + r_3 + r_4 + \gamma Q(s_5, \mathbf{O5}) - Q(s_1, \mathbf{O1}) \right]$$

For impasses other than operator no-change, RL acts as if the impasse hadn't occurred. If **O1** is the last RL operator selected before the impasse, r_2 the reward received in the decision cycle immediately following, and **O$_n$**, the first operator selected after the impasse, then **O1** is updated with

$$\delta_1 = \alpha \left[r_2 + \gamma Q(s_n, \mathbf{O_n}) - Q(s_1, \mathbf{O1}) \right]$$

If an RL operator is selected in a substate immediately prior to the state's retraction, the RL rules will be updated based only on the reward signals present and not on the Q-values of future operators. This point is not covered in traditional RL theory. The retraction of a substate corresponds to a suspension of the RL task in that state rather than its termination, so the last update assumes the lack of information about future rewards rather than the

discontinuation of future rewards. To handle this case, the numeric indifferent preference value of each RL rule is stored as two separate values, the expected current reward (ECR) and expected future reward (EFR). The ECR is an estimate of the expected immediate reward signal for executing the corresponding RL operator. The EFR is an estimate of the time discounted Q-value of the next RL operator. Normal updates correspond to traditional RL theory (showing the Sarsa case for simplicity):

$$\delta_{ECR} = \alpha \left[r_t - ECR(s_t, a_t) \right]$$
$$\delta_{EFR} = \alpha \left[\gamma Q(s_{t+1}, a_{t+1}) - EFR(s_t, a_t) \right]$$
$$\delta_t = \delta_{ECR} + \delta_{EFR}$$
$$= \alpha \left[r_t + \gamma Q(s_{t+1}, a_{t+1}) - (ECR(s_t, a_t) + EFR(s_t, a_t)) \right]$$
$$= \alpha \left[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right]$$

During substate retraction, only the ECR is updated based on the reward signals present at the time of retraction, and the EFR is unchanged.

Soar's automatic subgoaling and RL mechanisms can be combined to naturally implement hierarchical reinforcement learning algorithms such as MAXQ and options.

5.3.3   Eligibility Traces

The RL mechanism supports eligibility traces, which can improve the speed of learning by updating RL rules across multiple sequential steps. The **eligibility-trace-decay-rate** and **eligibility-trace-tolerance** parameters control this mechanism. By setting **eligibility-trace-decay-rate** to **0** (default), eligibility traces are in effect disabled. When eligibility traces are enabled, the particular algorithm used is dependent upon the learning policy. For Sarsa, the eligibility trace implementation is *Sarsa(λ)*. For Q-Learning, the eligibility trace implementation is *Watkin's Q(λ)*.

5.3.3.1   Exploration

The **indifferent-selection** command (page 164) determines how operators are selected based on their numeric indifferent preferences. Although all the indifferent selection settings are valid regardless of how the numeric indifferent preferences were arrived at, the **epsilon-greedy** and **boltzmann** settings are specifically designed for use with RL and correspond to the two most common exploration strategies. In an effort to maintain backwards compatibility, the default exploration policy is **softmax**. As a result, one should change to **epsilon-greedy** or **boltzmann** when the reinforcement learning mechanism is enabled.

5.4   Automatic Generation of RL Rules

The number of RL rules required for an agent to accurately approximate operator Q-values is usually infeasibly large to write by hand, even for small domains. Therefore, several methods exist to automate this.

5.4.1   The gp Command

The `gp` command can be used to generate productions based on simple patterns. This is useful if the states and operators of the environment can be distinguished by a fixed number of dimensions with finite domains. An example is a grid world where the states are described by integer row/column coordinates, and the available operators are to move north, south, east, or west. In this case, a single `gp` command will generate all necessary RL rules:

```
gp {gen*rl*rules
   (state <s> ^name gridworld
              ^operator <o> +
              ^row [ 1 2 3 4 ]
              ^col [ 1 2 3 4 ])
   (<o> ^name move
        ^direction [ north south east west ])
-->
   (<s> ^operator <o> = 0.0)
}
```

For more information see the documentation for this command on page 118.

5.4.2   Rule Templates

Rule templates allow Soar to dynamically generate new RL rules based on a predefined pattern as the agent encounters novel states. This is useful when either the domains of environment dimensions are not known ahead of time, or when the enumerable state space of the environment is too large to capture in its entirety using `gp`, but the agent will only encounter a small fraction of that space during its execution. For example, consider the grid world example with 1000 rows and columns. Attempting to generate RL rules for each grid cell and action a priori will result in $1000 \times 1000 \times 4 = 4 \times 10^6$ productions. However, if most of those cells are unreachable due to walls, then the agent will never fire or update most of those productions. Templates give the programmer the convenience of the `gp` command without filling production memory with unnecessary rules.

Rule templates have variables that are filled in to generate RL rules as the agent encounters novel combinations of variable values. A rule template is valid if and only if it is marked with the **:template** flag and, in all other respects, adheres to the format of an RL rule. However, whereas an RL rule may only use constants as the numeric indifference preference value, a rule template may use a variable. Consider the following rule template:

```
sp {sample*rule*template
    :template
    (state <s> ^operator <o> +
              ^value <v>)
-->
    (<s> ^operator <o> = <v>)
}
```

During agent execution, this rule template will match working memory and create new productions by substituting all variables in the rule template that matched against constant values with the values themselves. Suppose that the LHS of the rule template matched against the state

```
(S1 ^value 3.2)
(S1 ^operator O1 +)
```

Then the following production will be added to production memory:

```
sp {rl*sample*rule*template*1
    (state <s> ^operator <o> +
              ^value 3.2)
-->
    (<s> ^operator <o> = 3.2)
}
```

The variable `<v>` is replaced by `3.2` on both the LHS and the RHS, but `<s>` and `<o>` are not replaced because they matches against identifiers (`S1` and `O1`). As with other RL rules, the value of `3.2` on the RHS of this rule may be updated later by reinforcement learning, whereas the value of `3.2` on the LHS will remain unchanged. If `<v>` had matched against a non-numeric constant, it will be replaced by that constant on the LHS, but the RHS numeric indifference preference value will be set to zero to make the new rule valid.

The new production's name adheres to the following pattern: **rl*template−name*id**, where **template−name** is the name of the originating rule template and **id** is monotonically increasing integer that guarantees the uniqueness of the name.

If an identical production already exists in production memory, then the newly generate production is discarded. It should be noted that the current process of identifying unique template match instances can become quite expensive in long agent runs. Therefore, it is recommended to generate all necessary RL rules using the `gp` command or via custom scripting when possible.

5.4.3   Chunking

Since RL rules are regular productions, they can be learned by chunking just like any other production. This method is more general than using the `gp` command or rule templates, and is useful if the environment state consists of arbitrarily complex relational structures that cannot be enumerated.

Chapter 6

Semantic Memory

Soar's semantic memory is a repository for long-term declarative knowledge, supplementing what is contained in short-term working memory (and production memory). Episodic memory, which contains memories of the agent's experiences, is described in Chapter 7. The knowledge encoded in episodic memory is organized temporally, and specific information is embedded within the context of when it was experienced, whereas knowledge in semantic memory is independent of any specific context, representing more general facts about the world.

This chapter is organized as follows: semantic memory structures in working memory (6.1); representation of knowledge in semantic memory (6.2); storing semantic knowledge (6.3); retrieving semantic knowledge (6.4); and a discussion of performance (6.5). The detailed behavior of semantic memory is determined by numerous parameters that can be controlled and configured via the **smem** command. Please refer to the documentation for that command in Section 8.4 on page 182.

6.1 Working Memory Structure

Upon creation of a new state in working memory (see Section 2.6.1 on page 24; Section 3.4 on page 66), the architecture creates the following augmentations to facilitate agent interaction with semantic memory:

```
(<s> ^smem <smem>)
  (<smem> ^command <smem-c>)
  (<smem> ^result <smem-r>)
```

As rules augment the *command* structure in order to access/change semantic knowledge (6.3, 6.4), semantic memory augments the *result* structure in response. Production actions should not remove augmentations of the *result* structure directly, as semantic memory will maintain these WMEs.
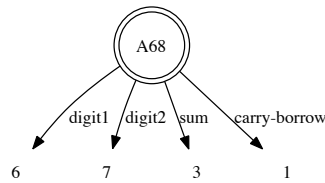
Figure 6.1: Example long-term identifier with four augmentations.

6.2   Knowledge Representation

The representation of knowledge in semantic memory is similar to that in working memory (see Section 2.2 on page 13) – both include graph structures that are composed of symbolic elements consisting of an identifier, an attribute, and a value. It is important to note, however, key differences:

- Currently semantic memory only supports attributes that are symbolic constants (string, integer, or decimal), but *not* attributes that are identifiers

- Whereas working memory is a single, connected, directed graph, semantic memory can be disconnected, consisting of multiple directed, connected sub-graphs

Long-term identifiers (LTIs) are defined as identifiers that exist in semantic memory. The specific letter-number combination that labels an LTI (e.g. S5 or C7) is permanently associated with that long-term identifier: any retrievals of the long-term identifier are guaranteed to return the associated letter-number pair. For clarity, when printed, a long-term identifier is prefaced with the @ symbol (e.g. @S5 or @C7). Also, when presented in a figure, long-term identifiers will be indicated by a double-circle. For instance, Figure 6.1 depicts the long-term identifier @A68, with four augmentations, representing the addition fact of $6 + 7 = 13$ (or, rather, 3, carry 1, in context of multi-column arithmetic).

6.2.1   Integrating Long-Term Identifiers with Soar

Integrating long-term identifiers in Soar presents a number of theoretical and implementation challenges. This section discusses the state of integration with each of Soar's memories/learning mechanisms.

6.2.1.1   Working Memory

Long-term identifiers exist as peers with short-term identifiers in Working Memory.

6.2.1.2   Procedural Memory

Soar's production parser (i.e. the **sp** command) has been modified to allow specification of long-term identifiers (prefaced with an @ symbol) in any context where a variable is valid. If a rule contains a long-term identifier that is not currently in semantic memory, a fatal error will be raised and Soar will quit. Once added to the rete, the long-term identifier is treated as a constant for matching purposes. If specified as the value of a WME in an action, a long-term identifier will be added to working memory if it does not already exist. There is also preliminary support for chunking over long-term identifiers.

It is currently possible to create production actions wherein the identifier of a new WME is a long-term identifier that exists neither in the production conditions, nor as the attribute or value of a prior action. Such rules will wreak havoc within Soar and are not supported. They will be detected and disallowed in future versions of semantic memory.

6.2.1.3   Episodic Memory

Episodic memory (see Section 7 on page 105) faithfully captures short- vs. long-term identifiers, including the episode of transition. Cues are handled in much the same way as cue-based retrievals, with respect to the differences in semantics of a short- vs. long-term identifier.

6.3   Storing Semantic Knowledge

An agent stores a long-term identifier to semantic memory by creating a *store* command: this is a WME whose identifier is the *command* link of a state's *smem* structure, the attribute is *store*, and the value is an identifier (short or long).

```
<s> ^smem.command.store <identifier>
```

Semantic memory will encode and store all WMEs whose identifier is the value of the store command. Storing deeper levels of working memory is achieved through multiple store commands.

Multiple store commands can be issued in parallel. Storage commands are processed on every state at the end of every phase of every decision cycle. Storage is guaranteed to succeed and a status WME will be created, where the identifier is the *result* link of the *smem* structure of that state, the attribute is *success*, and the value is the value of the store command above.

```
<s> ^smem.result.success <identifier>
```

Any short-term identifiers that compose the stored WMEs will be converted to long-term identifiers. If a long-term identifier is the value of a store command, the stored WMEs replace those associated with the LTI in semantic memory. It should be noted that between issuing store commands, it is possible that the augmentations of a long-term identifier in working memory are inconsistent with those in semantic memory.

6.3.1   User-Initiated Storage

Semantic memory provides agent designers the ability to store semantic knowledge via the **add** switch of the **smem** command (see Section 8.4 on page 182). The format of the command is nearly identical to the working memory manipulation components of the RHS of a production (i.e. no RHS-functions; see Section 3.3.6 on page 56). For instance:

```
smem --add {
   (<arithmetic> ^add10-facts <a01> <a02> <a03>)
   (<a01> ^digit1 1 ^digit-10 11)
   (<a02> ^digit1 2 ^digit-10 12)
   (<a03> ^digit1 3 ^digit-10 13)
}
```

Unlike agent storage, declarative storage is automatically recursive. Thus, this command instance will add a new long-term identifier (represented by the temporary 'arithmetic' variable) with three augmentations. The value of each augmentation will each become an LTI with two constant attribute/value pairs. Manual storage can be arbitrarily complex and use standard dot-notation.

6.3.2   Storage Location

Semantic memory uses SQLite to facilitate efficient and standardized storage and querying of knowledge. The semantic store can be maintained in memory or on disk (per the **database** and **path** parameters). If the store is located on disk, users can use any standard SQLite programs/components to access/query its contents. However, using a disk-based semantic store is very costly (performance is discussed in greater detail in Section 6.5 on page 101), and running in memory is recommended for most runs.

The **lazy-commit** parameter is a performance optimization. If set to **on** (default), disk databases will not reflect semantic memory changes until the Soar kernel shuts down. This improves performance by avoiding disk writes. The **optimization** parameter (see Section 6.5 on page 101) will have an affect on whether databases on disk can be opened while the Soar kernel is running.

6.4   Retrieving Semantic Knowledge

An agent retrieves knowledge from semantic memory by creating an appropriate command (we detail the types of commands below) on the *command* link of a state's *smem* structure. At the end of the output of each decision, semantic memory processes each state's *smem* command structure. Results, meta-data, and errors are added to the *result* structure of that state's *smem* structure.

Only one type of retrieval command (which may include optional modifiers) can be issued per state in a single decision cycle. Malformed commands (including attempts at multiple

retrieval types) will result in an error:

```
<s> ^smem.result.bad-cmd <smem-c>
```

Where the **smem-c** variable refers to the *command* structure of the state.

After a command has been processed, semantic memory will ignore it until some aspect of the command structure changes (via addition/removal of WMEs). When this occurs, the result structure is cleared and the new command (if one exists) is processed.

6.4.1 Non-Cue-Based Retrievals

A non-cue-based retrieval is a request by the agent to reflect in working memory the current augmentations of a long-term identifier in semantic memory. The command WME has a *retrieve* attribute and a long-term identifier value:

```
<s> ^smem.command.retrieve <lti>
```

If the value of the command is not a long-term identifier, an error will result:

```
<s> ^smem.result.failure <lti>
```

Otherwise, two new WMEs will be placed on the result structure:

```
<s> ^smem.result.success <lti>
<s> ^smem.result.retrieved <lti>
```

All augmentations of the long-term identifier in semantic memory will be created as new WMEs in working memory.

6.4.2 Cue-Based Retrievals

A cue-based retrieval performs a search for a long-term identifier in semantic memory whose augmentations exactly match an agent-supplied cue, as well as optional cue modifiers.

A cue is composed of WMEs that describe the augmentations of a long-term identifier. A cue WME with a constant value denotes an exact match of both attribute and value. A cue WME with a long-term identifier as its value denotes an exact match as well. A cue WME with a short-term identifier as its value denotes an exact match of attribute, but with any value (constant or identifier).

A cue-based retrieval command has a *query* attribute and an identifier value, the cue:

```
<s> ^smem.command.query <cue>
```

For instance, consider the following rule that creates a cue-based retrieval command:

```
sp {smem*sample*query
    (state <s> ^smem.command <sc>
               ^lti <lti>
```

```
                ^input-link.foo <bar>)
-->
    (<sc> ^query <q>)
    (<q> ^name <any-name>
         ^foo <bar>
         ^associate <lti>
         ^age 25)
}
```

In this example, assume that the `<lti>` variable will match a long-term identifier and the `<bar>` variable will match a constant. Thus, the query requests retrieval of a long-term identifier from semantic memory with augmentations that satisfy ALL of the following requirements:

- Attribute `name` and ANY value

- Attribute `foo` and value equal to the value of variable `<bar>` at the time this rule fires

- Attribute `associate` and value equal to the long-term identifier `<lti>` at the time this rule fires

- Attribute `age` and integer value `25`

If no long-term identifier satisfies ALL of these requirements, an error is returned:

```
<s> ^smem.result.failure <cue>
```

Otherwise, two WMEs are added:

```
<s> ^smem.result.success <cue>
<s> ^smem.result.retrieved <retrieved-lti>
```

During a cue-based retrieval it is possible that the retrieved long-term identifier is not in working memory. If this is the case, semantic memory will add the long-term identifier to working memory with letter-number pair as was originally stored.

As with non-cue-based retrievals all of the augmentations of the long-term identifier in semantic memory are added as new WMEs to working memory.

It is possible that multiple long-term identifiers match the cue equally well. In this case, semantic memory will retrieve the long-term identifier that was most recently stored/retrieved.

The cue-based retrieval process can be further tempered using optional modifiers:

- The *prohibit* command requires that the retrieved long-term identifier is not equal to a supplied long-term identifier:

  ```
  <s> ^smem.command.prohibit <bad-lti>
  ```

  Multiple prohibit command WMEs may be issued as modifiers to a single cue-based retrieval. This method can be used to iterate over all matching long-term identifiers.

- The *neg-query* command requires that the retrieved long-term identifier does NOT contain a set of attributes/attribute-value pairs:

```
<s> ^smem.command.neg-query <cue>
```

  The syntax of this command is identical to that of regular/positive *query* command.

- The *math-query* command requires that the retrieved long term identifier contains an attribute value pair that meets a specified mathematical condition. This condition can either be a conditional query or a superlative query. Conditional queries are of the format:

```
<s> ^smem.command.math-query.<cue-attribute>.<condition-name> <cue-value>
```

  Superlative queries do not use a value argument and are of the format:

```
<s> ^smem.command.math-query.<cue-attribute>.<condition-name>
```

  Values used in math queries must be integer or float type values. Currently supported condition names are:

  less A value less than the given argument

  greater A value greater than the given argument

  less-or-equal A value less than or equal to the given argument

  greater-or-equal A value greater than or equal to the given argument

  max The maximum value for the attribute

  min The minimum value for the attribute

6.5 Performance

Initial empirical results with toy agents show that semantic memory queries carry up to a 40% overhead as compared to comparable rete matching. However, the retrieval mechanism implements some basic query optimization: statistics are maintained about all stored knowledge. When a query is issued, semantic memory re-orders the cue such as to minimize expected query time. Because only perfect matches are acceptable, and there is no symbol variablization, semantic memory retrievals do not contend with the same combinatorial search space as the rete. Preliminary empirical study shows that semantic memory maintains sub-millisecond retrieval time for a large class of queries, even in very large stores (millions of nodes/edges).

Once the number of long-term identifiers overcomes initial overhead (about 1000 WMEs), initial empirical study shows that semantic storage requires far less than 1KB per stored WME.

6.5.1　Math queries

There are some additional performance considerations when using math queries during retrieval. Initial testing indicates that conditional queries show the same time growth with respect to the number of memories as similar non-math restricted queries, however the actual time for retrieval may be slightly longer. Superelative queries will often show a worse result than similar non-superelative queries, because the current implementation of semantic memory requires them to iterate over any memory that matches all other involved cues.

6.5.2　Performance Tweaking

When using a database stored to disk, several parameters become crucial to performance. The first is **lazy-commit**, which controls when database changes are written to disk. The default setting (**on**) will keep all writes in memory and only commit to disk upon re-initialization (quitting the agent or issuing the **init** command). The **off** setting will write each change to disk and thus incurs massive I/O delay.

The next parameter is **thresh**. This has to do with the locality of storing/updating activation information with semantic augmentations. By default, all WME augmentations are incrementally sorted by activation, such that cue-based retrievals need not sort large number of candidate long-term identifiers on demand, and thus retrieval time is independent of cue selectivity. However, each activation update (such as after a retrieval) incurs an update cost linear in the number of augmentations. If the number of augmentations for a long-term identifier is large, this cost can dominate. Thus, the **thresh** parameter sets the upper bound of augmentations, after which activation is stored with the long-term identifier. This allows the user to establish a balance between cost of updating augmentation activation and the number of long-term identifiers that must be pre-sorted during a cue-based retrieval. As long as the threshold is greater than the number of augmentations of most long-term identifiers, performance should be fine (as it will bound the effects of selectivity).

The next two parameters deal with the SQLite cache, which is a memory store used to speed operations like queries by keeping in memory structures like levels of index B+-trees. The first parameter, **page-size**, indicates the size, in bytes, of each cache page. The second parameter, **cache-size**, suggests to SQLite how many pages are available for the cache. Total cache size is the product of these two parameter settings. The cache memory is not pre-allocated, so short/small runs will not necessarily make use of this space. Generally speaking, a greater number of cache pages will benefit query time, as SQLite can keep necessary metadata in memory. However, some documented situations have shown improved performance from decreasing cache pages to increase memory locality. This is of greater concern when dealing with file-based databases, versus in-memory. The size of each page, however, may be important whether databases are disk- or memory-based. This setting can have far-reaching consequences, such as index B+-tree depth. While this setting can be dependent upon a particular situation, a good heuristic is that short, simple runs should use small values of the page size (**1k**, **2k**, **4k**), whereas longer, more complicated runs will benefit from larger values (**8k**, **16k**, **32k**, **64k**). The episodic memory chapter (see Section 7.4 on page 111) has

some further empirical evidence to assist in setting these parameters for very large stores.

The next parameter is **optimization**. The **safety** parameter setting will use SQLite default settings. If data integrity is of importance, this setting is ideal. The **performance** setting will make use of lesser data consistency guarantees for significantly greater performance. First, writes are no longer synchronous with the OS (synchronous pragma), thus semantic memory won't wait for writes to complete before continuing execution. Second, transaction journaling is turned off (journal_mode pragma), thus groups of modifications to the semantic store are not atomic (and thus interruptions due to application/os/hardware failure could lead to inconsistent database state). Finally, upon initialization, semantic memory maintains a continuous exclusive lock to the database (locking_mode pragma), thus other applications/agents cannot make simultaneous read/write calls to the database (thereby reducing the need for potentially expensive system calls to secure/release file locks).

Finally, maintaining accurate operation timers can be relatively expensive in Soar. Thus, these should be enabled with caution and understanding of their limitations. First, they will affect performance, depending on the level (set via the **timers** parameter). A level of **three**, for instance, times every modification to long-term identifier recency statistics. Furthermore, because these iterations are relatively cheap (typically a single step in the linked-list of a b+-tree), timer values are typically unreliable (depending upon the system, resolution is 1 microsecond or more).

Chapter 7

Episodic Memory

Episodic memory is a record of an agent's stream of experience. The episodic storage mechanism will automatically record episodes as a Soar agent executes. The agent can later deliberately retrieve episodic knowledge to extract information and regularities that may not have been noticed during the original experience and combine them with current knowledge such as to improve performance on future tasks.

This chapter is organized as follows: episodic memory structures in working memory (7.1); episodic storage (7.2); retrieving episodes (7.3); and a discussion of performance (7.4). The detailed behavior of episodic memory is determined by numerous parameters that can be controlled and configured via the **epmem** command. Please refer to the documentation for that command in Section 8.4 on page 155.

7.1   Working Memory Structure

Upon creation of a new state in working memory (see Section 2.6.1 on page 24; Section 3.4 on page 66), the architecture creates the following augmentations to facilitate agent interaction with episodic memory:

```
(<s> ^epmem <e>)
  (<e> ^command <e-c>)
  (<e> ^result <e-r>)
  (<e> ^present-id #)
```

As rules augment the *command* structure in order to retrieve episodes (7.3), episodic memory augments the *result* structure in response. Production actions should not remove augmentations of the *result* structure directly, as episodic memory will maintain these WMEs.

The value of the *present-id* augmentation is an integer and will update to expose to the agent the current episode number. This information is identical to what is available via the *time* statistic (see Section 8.4 on page 155) and the *present-id* retrieval meta-data (7.3.4).

7.2   Episodic Storage

Episodic memory records new episodes without deliberate action/consideration by the agent. The timing and frequency of recording new episodes is controlled by the **phase** and **trigger** parameters. The **phase** parameter sets the phase in the decision cycle (default: end of Output phase) during which episodic memory stores episodes and processes commands. The value of the **trigger** parameter indicates to the architecture the event that concludes an episode: adding a new augmentation to the output-link (default) or each decision cycle.

For debugging purposes, the **force** parameter allows the user to manually request that an episode be recorded (or not) during the current decision cycle. Behavior is as follows:

- The value of the **force** parameter is initialized to **off** every decision cycle.

- During the **phase** of episodic storage, episodic memory tests the value of the **force** parameter; if it has a value other than of off, episodic memory follows the *forced* policy irrespective of the value of the **trigger** parameter.

7.2.1   Episode Contents

When episodic memory stores a new episode, it captures the entire top-state of working memory. There are currently two exceptions to this policy:

- Episodic memory only supports WMEs whose attribute is a constant. Behavior is currently undefined when attempting to store a WME that has an attribute that is an identifier.

- The **exclusions** parameter allows the user to specify a set of attributes for which Soar will not store WMEs. The storage process currently walks the top-state of working memory in a breadth-first manner, and any WME that is not reachable other than via an excluded WME will not be stored. By default, episodic memory excludes the *epmem* and *smem* structures, to prevent encoding of potentially large and/or frequently changing memory retrievals.

7.2.2   Storage Location

Episodic memory uses SQLite to facilitate efficient and standardized storage and querying of episodes. The episodic store can be maintained in memory or on disk (per the **database** and **path** parameters). If the store is located on disk, users can use any standard SQLite programs/components to access/query its contents. See the later discussion on performance (7.4) for additional parameters dealing with databases on disk.

7.3 Retrieving Episodes

An agent retrieves episodes by creating an appropriate command (we detail the types of commands below) on the *command* link of a state's *epmem* structure. At the end of the **phase** of each decision, after episodic storage, episodic memory processes each state's *epmem* command structure. Results, meta-data, and errors are placed on the *result* structure of that state's *epmem* structure.

Only one type of retrieval command (which may include optional modifiers) can be issued per state in a single decision cycle. Malformed commands (including attempts at multiple retrieval types) will result in an error:

```
<s> ^epmem.result.status bad-cmd
```

After a command has been processed, episodic memory will ignore it until some aspect of the command structure changes (via addition/removal of WMEs). When this occurs, the result structure is cleared and the new command (if one exists) is processed.

7.3.1 Cue-Based Retrievals

Cue-based retrieval commands are used to search for an episode in the store that best matches an agent-supplied cue, while adhering to optional modifiers. A cue is composed of WMEs that partially describe a top-state of working memory in the retrieved episode. All cue-based retrieval requests must contain a single *query* cue and, optionally, a single *neg-query* cue.

```
<s> ^epmem.command.query <required-cue>
<s> ^epmem.command.neg-query <optional-negative-cue>
```

A *query* cue describes structures desired in the retrieved episode, whereas a *neg-query* cue describes non-desired structures. For example, the following Soar production creates a *query* cue consisting of a particular state name and a copy of a current value on the *input-link* structure:

```
sp {epmem*sample*query
    (state <s> ^epmem.command <ec>
               ^io.input-link.foo <bar>)
-->
    (<ec> ^query <q>)
    (<q> ^name my-state-name
         ^io.input-link.foo <bar>)
}
```

As detailed below, multiple prior episodes may equally match the structure and contents of an agent's cue. Nuxoll has produced initial evidence that in some tasks, retrieval quality improves when using *activation* of cue WMEs as a form of feature weighting. Thus, episodic memory supports integration with working memory activation (see Section 8.4 on page 189). For a theoretical discussion of the Soar implementation of working memory activation, con-

sider reading *Comprehensive Working Memory Activation in Soar* (Nuxoll, A., Laird, J., James, M., ICCM 2004).

The cue-based retrieval process can be thought of conceptually as a nearest-neighbor search. First, all candidate episodes, defined as episodes containing at least one leaf WME (a cue WME with no sub-structure) in at least one cue, are identified. Two quantities are calculated for each candidate episode, with respect to the supplied cue(s): the cardinality of the match (defined as the number of matching leaf WMEs) and the activation of the match (defined as the sum of the activation values of each matching leaf WME). Note that each of these values is negated when applied to a negative query. To compute each candidate episode's match score, these quantities are combined with respect to the **balance** parameter as follows:

$$(balance) * (cardinality) + (1 - balance) * (activation)$$

Performing a graph match on each candidate episode, with respect to the structure of the cue, could be very computationally expensive, so episodic memory implements a two-stage matching process. An episode with perfect cardinality is considered a perfect *surface* match and, per the **graph-match** parameter, is subjected to further *structural* matching. Whereas surface matching efficiently determines if all paths to leaf WMEs exist in a candidate episode, graph matching indicates whether or not the cue can be structurally unified with the candidate episode (paying special regard to the structural constraints imposed by shared identifiers). Cue-based matching will return the most recent structural match, or the most recent candidate episode with the greatest match score.

A special note should be made with respect to how short- vs. long-term identifiers (see Section 6.2 on page 96) are interpreted in a cue. Short-term identifiers are processed much as they are in working memory – transient structures. Cue matching will try to find any identifier in an episode (with respect to WME path from state) that can apply. Long-term identifiers, however, are treated as constants. Thus, when analyzing the cue, episodic memory will not consider long-term identifier augmentations, and will only match with the same long-term identifier (in the same context) in an episode.

The case-based retrieval process can be further controlled using optional modifiers:

- The *before* command requires that the retrieved episode come relatively before a supplied time:

  ```
  <s> ^epmem.command.before time
  ```

- The *after* command requires that the retrieved episode come relatively after a supplied time:

  ```
  <s> ^epmem.command.after time
  ```

- The *prohibit* command requires that the time of the retrieved episode is not equal to a supplied time:

```
<s> ^epmem.command.prohibit time
```

Multiple prohibit command WMEs may be issued as modifiers to a single CB retrieval.

If no episode satisfies the cue(s) and optional modifiers an error is returned:

```
<s> ^epmem.result.failure <query> <optional-neg-query>
```

If an episode is returned, there is additional meta-data supplied (7.3.4).

7.3.2 Absolute Non-Cue-Based Retrieval

At time of storage, each episode is attributed a unique *time*. This is the current value of **time** statistic and is provided as the *memory-id* meta-data item of retrieved episodes (7.3.4). An absolute non-cue-based retrieval is one that requests an episode by time. An agent issues an absolute non-cue-based retrieval by creating a WME on the *command* structure with attribute *retrieve* and value equal to the desired time:

```
<s> ^epmem.command.retrieve time
```

Supplying an invalid value for the *retrieve* command will result in an error.

The time of the first episode in an episodic store will have value 1 and each subsequent episode's time will increase by 1. Thus the desired time may be the mathematical result of operations performed on a known episode's time.

The current episodic memory implementation does not implement any episodic store dynamics, such as forgetting. Thus any integer time greater than 0 and less than the current value of the **time** statistic will be valid. However, if forgetting is implemented in future versions, no such guarantee will be made.

7.3.3 Relative Non-Cue-Based Retrieval

Episodic memory supports the ability for an agent to "play forward" episodes using relative non-cue-based retrievals.

Episodic memory stores the time of the last successful retrieval (non-cue-based or cue-based). Agents can indirectly make use of this information by issuing *next* or *previous* commands. Episodic memory executes these commands by attempting to retrieve the episode immediately proceeding/preceding the last successful retrieval (respectively). To issue one of these commands, the agent must create a new WME on the *command* link with the appropriate attribute (*next* or *previous*) and value of an arbitrary identifier:

```
<s> ^epmem.command.next <n>
<s> ^epmem.command.previous <p>
```

If no such episode exists then an error is returned.

Currently, if the time of the last successfully retrieved episode is known to the agent (as could be the case by accessing result meta-data), these commands are identical to performing an absolute non-cue-based retrieval after adding/subtracting 1 to the last time (respectively). However, if an episodic store dynamic like forgetting is implemented, these relative commands are guaranteed to return the next/previous valid episode (assuming one exists).

7.3.4   Retrieval Meta-Data

The following list details the WMEs that episodic memory creates in the *result* link of the *epmem* structure wherein a command was issued:

- **retrieved <retrieval-root>**

  If episodic memory retrieves an episode, that memory is placed here. This WME is an identifier that is treated as the root of the state that was used to create the episodic memory. If the *retrieve* command was issued with an invalid time, the value of this WME will be *no-memory*.

- **success <query> <optional-neg-query>**

  If the cue-based retrieval was successful, the WME will have the status as the attribute and the value of the identifier of the query (and neg-query, if applicable).

- **match-score**

  This WME is created whenever an episode is successfully retrieved from a cue-based retrieval command. The WME value is a decimal indicating the raw match score for that episode with respect to the cue(s).

- **cue-size**

  This WME is created whenever an episode is successfully retrieved from a cue-based retrieval command. The WME value is an integer indicating the number of leaf WMEs in the cue(s).

- **normalized-match-score**

  This WME is created whenever an episode is successfully retrieved from a cue-based retrieval command. The WME value is the decimal result of dividing the raw match score by the cue size. It can hypothetically be used as a measure of episodic memory's relative confidence in the retrieval.

- **match-cardinality**

  This WME is created whenever an episode is successfully retrieved from a cue-based retrieval command. The WME value is an integer indicating the number of leaf WMEs matched in the *query* cue minus those matched in the *neg-query* cue.

- **memory-id**

This WME is created whenever an episode is successfully retrieved from a cue-based retrieval command. The WME value is an integer indicating the time of the retrieved episode.

- **present-id**

  This WME is created whenever an episode is successfully retrieved from a cue-based retrieval command. The WME value is an integer indicating the current time, such as to provide a sense of "now" in episodic memory terms. By comparing this value to the *memory-id* value, the agent can gain a sense of the relative time that has passed since the retrieved episode was recorded.

- **graph-match**

  This WME is created whenever an episode is successfully retrieved from a cue-based retrieval command and the **graph-match** parameter was **on**. The value is an integer with value 1 if graph matching was executed successfully and 0 otherwise.

- **mapping <mapping-root>**

  This WME is created whenever an episode is successfully retrieved from a cue-based retrieval command, the **graph-match** parameter was **on**, and structural match was successful on the retrieved episode. This WME provides a mapping between identifiers in the cue and in the retrieved episode. For each identifier in the cue, there is a *node* WME as an augmentation to the *mapping* identifier. The node has a *cue* augmentation, whose value is an identifier in the cue, and a *retrieved* augmentation, whose value is an identifier in the retrieved episode. In a graph match it is possible to have multiple identifier mappings – this map represents the "first" unified mapping (with respect to episodic memory algorithms).

7.4 Performance

There are currently two sources of "unbounded" computation: graph matching and cue-based queries. Graph matching is combinatorial in the worst case. Thus, if an episode presents a perfect surface match, but imperfect structural match (i.e. there is no way to unify the cue with the candidate episode), there is the potential for exhaustive search. Each identifier in the cue can be assigned one of any historically consistent identifiers (with respect to the sequence of attributes that leads to the identifier from the root), termed a literal. If the identifier is a multi-valued attribute, there will be more than one candidate literals and this situation can lead to a very expensive search process. Currently there are no heuristics in place to attempt to combat the expensive backtracking. Worst-case performance will be combinatorial in the total number of literals for each cue identifier (with respect to cue structure).

The cue-based query algorithm begins with the most recent candidate episode and will stop search as soon as a match is found (since this episode must be the most recent). Given this procedure, it is trivial to create a two-WME cue that forces a linear search of the episodic

store. Episodic memory combats linear scan by only searching candidate episodes, i.e. only those that contain a change in at least one of the cue WMEs. However, a cue that has no match and contains WMEs relevant to all episodes will force inspection of all episodes. Thus, worst-case performance will be linear in the number of episodes.

7.4.1   Performance Tweaking

When using a database stored to disk, several parameters become crucial to performance. The first is **commit**, which controls the number of episodes that occur between writes to disk. If the total number of episodes (or a range) is known ahead of time, setting this value to a greater number will result in greatest performance (due to decreased I/O).

The next two parameters deal with the SQLite cache, which is a memory store used to speed operations like queries by keeping in memory structures like levels of index B+-trees. The first parameter, **page-size**, indicates the size, in bytes, of each cache page. The second parameter, **cache-size**, suggests to SQLite how many pages are available for the cache. Total cache size is the product of these two parameter settings. The cache memory is not pre-allocated, so short/small runs will not necessarily make use of this space. Generally speaking, a greater number of cache pages will benefit query time, as SQLite can keep necessary meta-data in memory. However, some documented situations have shown improved performance from decreasing cache pages to increase memory locality. This is of greater concern when dealing with file-based databases, versus in-memory. The size of each page, however, may be important whether databases are disk- or memory-based. This setting can have far-reaching consequences, such as index B+-tree depth. While this setting can be dependent upon a particular situation, a good heuristic is that short, simple runs should use small values of the page size (**1k**, **2k**, **4k**), whereas longer, more complicated runs will benefit from larger values (**8k**, **16k**, **32k**, **64k**). One known situation of concern is that as indexed tables accumulate many rows (millions), insertion time of new rows can suffer an infrequent, but linearly increasing burst of computation. In episodic memory, this situation will typically arise with many episodes and/or many working memory changes. Increasing the page size will reduce the intensity of the spikes at the cost of increasing disk I/O and average/total time for episode storage. Thus, the settings of page size for long, complicated runs establishes the desired balance of reactivity (i.e. max computation) and average speed. To ground this discussion, the Figure 7.1 depicts maximum and average episodic storage time (the value of the epmem_storage timer, converted to milliseconds) with different page sizes after 10 million decisions (1 episode/decision) of a very basic agent (i.e. very few working memory changes per episode) running on a 2.8GHz Core i7 with Mac OS X 10.6.5. While only a single use case, the cross-point of these data forms the basis for the decision to default the parameter at 8192 bytes.

The next parameter is **optimization**. The **safety** parameter setting will use SQLite default settings. If data integrity is of importance, this setting is ideal. The performance setting will make use of lesser data consistency guarantees for significantly greater performance. First, writes are no longer synchronous with the OS (synchronous pragma), thus episodic memory won't wait for writes to complete before continuing execution. Second, transaction
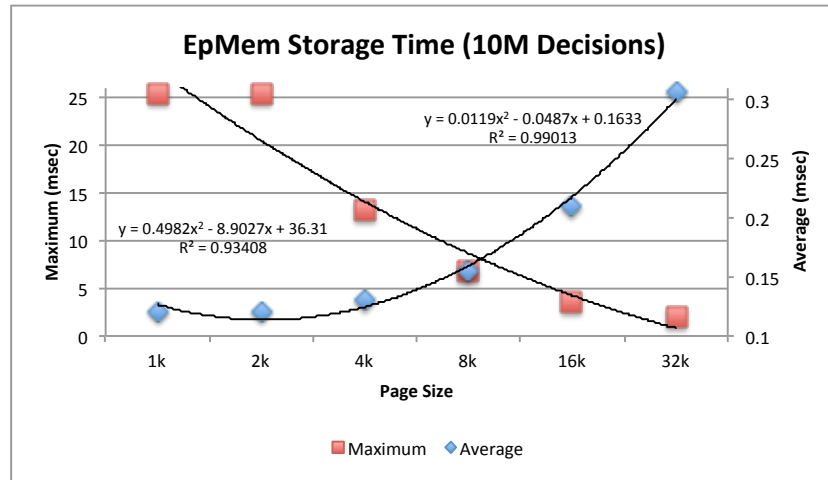
Figure 7.1: Example episodic memory cache setting data.

journaling is turned off (journal_mode pragma), thus groups of modifications to the episodic store are not atomic (and thus interruptions due to application/os/hardware failure could lead to inconsistent database state). Finally, upon initialization, episodic memory maintains a continuous exclusive lock to the database (locking_mode pragma), thus other applications/agents cannot make simultaneous read/write calls to the database (thereby reducing the need for potentially expensive system calls to secure/release file locks).

Finally, maintaining accurate operation timers can be relatively expensive in Soar. Thus, these should be enabled with caution and understanding of their limitations. First, they will affect performance, depending on the level (set via the **timers** parameter). A level of **three**, for instance, times every step in the cue-based retrieval candidate episode search. Furthermore, because these iterations are relatively cheap (typically a single step in the linked-list of a b+-tree), timer values are typically unreliable (depending upon the system, resolution is 1 microsecond or more).

Chapter 8

The Soar User Interface

This chapter describes the set of user interface commands for Soar. All commands and examples are presented as if they are being entered at the Soar command prompt.

This chapter is organized into 7 sections:

1. Basic Commands for Running Soar

2. Examining Memory

3. Configuring Trace Information and Debugging

4. Configuring Soar's Run-Time Parameters

5. File System I/O Commands

6. Soar I/O commands

7. Miscellaneous Commands

Each section begins with a summary description of the commands covered in that section, including the role of the command and its importance to the user. Command syntax and usage are then described fully, in alphabetical order.

The following pages were automatically generated from the wiki version at

> http://code.google.com/p/soar/wiki/CommandIndex

on the date listed on the title page of this manual. Please consult the wiki directly for the most accurate and up-to-date information.

For a concise overview of the Soar interface functions, see the Function Summary and Index on page 245. This index is intended to be a quick reference into the commands described in this chapter.

Notation

The notation used to denote the syntax for each user-interface command follows some general conventions:

- The command name itself is given in a **bold** font.
- Optional command arguments are enclosed within square brackets, [and].
- A vertical bar, |, separates alternatives.
- Curly braces, {}, are used to group arguments when at least one argument from the set is required.

- The commandline prompt that is printed by Soar, is normally the agent name, followed by '>'. In the examples in this manual, we use "soar>".

- Comments in the examples are preceded by a '#', and in-line comments are preceded by ';#'.

For many commands, there is some flexibility in the order in which the arguments may be given. (See the online help for each command for more information.) We have not incorporated this flexible ordering into the syntax specified for each command because doing so complicates the specification of the command. When the order of arguments will affect the output produced by a command, the reader will be alerted.

8.1   Basic Commands for Running Soar

This section describes the commands used to start, run and stop a Soar program; to invoke on-line help information; and to create and delete Soar productions. The specific commands described in this section are:

Summary

> **excise** - Delete Soar productions from production memory.
>
> **gp** - Define a pattern used to generate and source a set of Soar productions.
>
> **gp-max** - Set the upper-limit to the number of productions generated by the gp command.
>
> **help** - Provide formatted, on-line information about Soar commands.
>
> **init-soar** - Reinitialize Soar so a program can be rerun from scratch.
>
> **run** - Begin Soar's execution cycle.
>
> **sp** - Create a production and add it to production memory.
>
> **stop-soar** - Interrupt a running Soar program.

These commands are all frequently used anytime Soar is run.

excise

Delete Soar productions from production memory.

Synopsis

```
excise production_name
excise options
```

Default Aliases

ex	excise

Options

-a, --all	Remove all productions from memory **and** perform an init-soar command
-c, --chunks	Remove all chunks (learned productions) and justifications from memory
-d, --default	Remove all default productions (:default) from memory
-r, --rl	Excise Soar-RL rules
-t, --task	Remove chunks, justifications, and user productions from memory
-T, --templates	Excise Soar-RL templates
-u, --user	Remove all user productions (but not chunks or default rules) from memory
production_name	Remove the specific production with this name.

Description

This command removes productions from Soar's memory. The command must be called with either a specific production name or with a flag that indicates a particular group of productions to be removed. Using the flag `-a` or `--all` also causes an init-soar.

Examples

This command removes the production `my*first*production` and all chunks:

```
excise my*first*production --chunks
```

This removes all productions and does an [cmd_init_soar init-soar]:

```
excise --all
```

See Also

init-soar

gp

Generate productions according to a specified pattern.

Synopsis

```
gp { production_body }
```

Description

The gp command defines a pattern used to generate and source a set of Soar productions. production_body is a single argument that looks almost identical to a standard Soar rule that would be used with the sp command. Indeed, any syntax that is allowed in sp is also allowed in gp.

Patterns in gp are specified with sets of whitespace-seprated values in square brackets. Every combination of values across all square-bracketed value lists will be generated. Values with whitespaces can be used if wrapped in pipes. Characters can also be escaped with a backslash (so string literals with embedded pipes and spaces outside of string literals are both possible).

gp is primarily intended as an alternative to :template rules for reinforcement learning. :template rules generate new rules as patterns occur at run time. Unfortunately, this incurs a high run time cost. If all possible values are known in advance, then the rules can be generated using gp at source time, thus allowing code to run faster. gp is not appropriate when all possible values are not known or if the total number of possible rules is very large (and the system is likely to encounter only a small subset at run time). It is also possible to combine gp and :template (e.g., if some of the values are known and not others). This should reduce the run time cost of :template.

There is nothing that actually restricts gp to being used for RL, although for non-RL rules, a disjunction list (using `<<` and `>>`) is better where it can be used. More esoteric uses may include multiple bracketed value lists inside a disjunction list, or even variables in bracketed value lists.

Each rule generated by gp has `*integer` appended to its name (where `integer` is some incrementing number).

Examples

Template version of rule:

```
sp {water-jug*fill
   :template
   (state <s1> ^name water-jug ^operator <op> +
               ^jug <j1> <j2>)
   (<op> ^name fill ^fill-jug.volume <fvol>)
   (<j1> ^volume 3 ^contents <c1>)
   (<j2> ^volume 5 ^contents <c2>)
-->
   (<s1> ^operator <op> = 0)
}
```

gp version of rule (generates 144 rules):

```
gp {water-jug*fill
   (state <s1> ^name water-jug ^operator <op> +
               ^jug <j1> <j2>)
   (<op> ^name fill ^fill-jug.volume [3 5])
   (<j1> ^volume 3 ^contents [0 1 2 3])
   (<j2> ^volume 5 ^contents [0 1 2 3 4 5])
-->
   (<s1> ^operator <op> = 0)
}
```

Esoteric example (generates 24 rules):

```
gp {strange-example
   (state <s1> ^<< [att1 att2] [att3 att4] >> [ val |another val| |strange val\|| ])
-->
   (<s1> ^foo [bar <bar>])
}
```

testgp.soar contains many more examples.

See Also

sp

gp-max

Set the upper-limit to the number of productions generated by the gp command.

Synopsis

```
gp-max [value]
```

Options

value	Maximum number of productions to produce

Description

gp-max is used to limit the number of productions produced by a gp production. It is easy to write a gp production that has a combinatorial explosion and hangs for a long time while those productions are added to memory. The gp-max command bounds this. Use without an argument to query the current value.

Examples

```
 gp-max 1000
```

gp productions that produce more than 1000 productions will stop producing them when 1000 are made and return an error.

See Also

sp gp

help

Provide formatted usage information about Soar commands.

Synopsis

```
 help [command_name]
```

Default Aliases

?	help
h	help
man	help

Description

This command prints formatted help for the given command name. Issue alone to see what topics have help available.

init-soar

Empties working memory and resets run-time statistics.

Synopsis

```
init-soar
```

Default Aliases

init	init-soar
is	init-soar

Description

The init-soar command initializes Soar. It removes all elements from working memory, wiping out the goal stack, and resets all runtime statistics. The firing counts for all productions are reset to zero. The init-soar command allows a Soar program that has been halted to be reset and start its execution from the beginning.

init-soar does not remove any productions from production memory; to do this, use the excise command. Note however, that all justifications will be removed because they will no longer be supported.

See Also

excise

run

Begin Soar's execution cycle.

Synopsis

```
run -[d|e|o|p][g][u|n][s] [count] [-i e|p|d|o]
```

Default Aliases

d	run -d 1
e	run -e 1
step	run 1

Options

`-d, --decision`	Run Soar for count decision cycles.
`-e, --elaboration`	Run Soar for count elaboration cycles.
`-o, --output`	Run Soar until the nth time output is generated by the agent. Limited by the value of max-nil-output-cycles.
`-p, --phase`	Run Soar by phases. A phase is either an input phase, proposal phase, decision phase, apply phase, or output phase.
`-s, --self`	If other agents exist within the kernel, do not run them at this time.
`-u, --update`	Sets a flag in the update event callback requesting that an environment updates. This is the default if `--self` is not specified.
`-n, --noupdate`	Sets a flag in the update event callback requesting that an environment does not update. This is the default if `--self` is specified.
`count`	A single integer which specifies the number of cycles to run Soar.
`-i, --interleave`	Support round robin execution across agents at a finer grain than the run-size parameter. `e` = elaboration, `p` = phase, `d` = decision, `o` = output
`-g, --goal`	Run agent until a goal retracts

Deprecated Options These may be reimplemented in the future.

`--operator`	Run Soar until the nth time an operator is selected.
`--state`	Run Soar until the nth time a state is selected.

Description

The run command starts the Soar execution cycle or continues any execution that was temporarily stopped. The default behavior of run, with no arguments, is to cause Soar to execute until it is halted or interrupted by an action of a production, or until an external interrupt is issued by the user. The run command can also specify that Soar should run only for a specific number of Soar cycles or phases (which may also be prematurely stopped by a production action or the stop-soar command). This is helpful for debugging sessions, where users may want to pay careful attention to the specific productions that are firing and retracting.

The run command takes optional arguments: an integer, `count`, which specifies how many units to run; and a `units` flag indicating what steps or increments to use. If `count` is specified, but no `units` are specified, then Soar is run by decision cycles. If `units` are specified, but `count` is unpecified, then `count` defaults to '1'. If both are unspecified, Soar will run until either a `halt` is executed, an interrupt is received, or max stack depth is reached.

If there are multiple Soar agents that exist in the same Soar process, then issuing a run command in any agent will cause all agents to run with the same set of parameters, unless the flag `--self` is specified, in which case only that agent will execute.

If an environment is registered for the kernel's update event, then when the event it triggered, the environment will get information about how the run was executed. If a run was executed with the `--update` option, then then event sends a flag requesting that the environment actually update itself. If a run was executed with the –noupdate option, then the event sends a flag requesting that the environment not update itself. The `--update` option is the default when run is specified without the `--self` option is not specified. If the `--self` option is specified, then the `--noupdate` option is on by default. It is up to the environment to check for these flags and honor them.

Some use cases include:

`run --self`	runs one agent but not the environment
`run --self --update`	runs one agent and the environment
`run`	runs all agents and the environment
`run --noupdate`	runs all agents but not the environment

Setting an interleave size

When there are multiple agents running within the same process, it may be useful to keep agents more closely aligned in their execution cycle than the run increment (`--elaboration, --phases, --decisions, --output`) specifies. For instance, it may be necessary to keep agents in ¨lock step¨ at the phase level, even though the run command issued is for 5 decisions. Some use cases include:

`run -d 5 -i p`	run the agent one phase and then move to the next agent, looping over agents until they have run for 5 decision cycles
`run -o 3 -i d`	run the agent one decision cycle and then move to the next agent. When an agent generates output for the 3rd time, it no longer runs even if other agents continue.

The `interleave` parameter must always be equal to or smaller than the specified run parameter.

Note If Soar has been stopped due to a `halt` action, an init-soar command must be issued before Soar can be restarted with the run command.

sp

Define a Soar production.

Synopsis

```
sp {production_body}
```

Options

`production_body`	A Soar production.

Description

The sp command creates a new production and loads it into production memory. `production_body` is a single argument parsed by the Soar kernel, so it should be enclosed in curly braces to avoid being parsed by other scripting languages that might be in the same process. The overall syntax of a rule is as follows:

```
name
    ["documentation-string"]
    [FLAG*]
    LHS
    -->
    RHS
```

The first element of a rule is its name. Conventions for names are given in the Soar Users Manual. If given, the documentation-string must be enclosed in double quotes. Optional flags define the type of rule and the form of support its right-hand side assertions will receive. The specific flags are listed in a separate section below. The LHS defines the left-hand side of the production and specifies the conditions under which the rule can be fired. Its syntax is given in detail in a subsequent section. The –> symbol serves to separate the LHS and RHS portions. The RHS defines the right-hand side of the production and specifies the assertions to be made and the actions to be performed when the rule fires. The syntax of the allowable right-hand side actions are given in a later section. The Soar Users Manual gives an elaborate discussion of the design and coding of productions. Please see that reference for tutorial information about productions.

If the name of the new production is the same as an existing one, the old production will be overwritten (excised).

Rules matching the following requirement are flagged upon being created/sourced: a rule is a Soar-RL rule if and only if its right hand side (RHS) consists of a single numeric preference and it is not a template rule (see FLAGs below). This format exists to ease technical requirements of identifying/updating Soar-RL rules, as well as to make it easy for the agent programmer to add/maintain RL capabilities within an agent. See the Soar-RL Manual for further details.

RULE FLAGS   The optional FLAGs are given below. Note that these switches are preceeded by a colon instead of a dash – this is a Soar parser convention.

```
 :o-support      specifies that all the RHS actions are to be given
                 o-support when the production fires
```

```
:no-support        specifies that all the RHS actions are only to be given
                   i-support when the production fires


:default           specifies that this production is a default production
                   (this matters for excise -task and watch task)


:chunk             specifies that this production is a chunk
                   (this matters for learn trace)


:interrupt         specifies that Soar should stop running when this
                   production matches but before it fires
                   (this is a useful debugging tool)


:template          specifies that this production should be used to generate
                   new reinforcement learning rules by filling in those
                   variables that match constants in working memory
```

Multiple flags may be used, but not both of `o-support` and `no-support`.

Although you could force your productions to provide O-support or I-support by using these commands — regardless of the structure of the conditions and actions of the production — this is not proper coding style. The `o-support` and `no-support` flags are included to help with debugging, but should not be used in a standard Soar program.

Examples

```
sp {blocks*create-problem-space
     "This creates the top-level space"
     (state <s1> ^superstate nil)
     -->
     (<s1> ^name solve-blocks-world ^problem-space <p1>)
     (<p1> ^name blocks-world)
}
```


See Also

excise learn watch

stop-soar

Pause Soar.

Synopsis

```
stop-soar [-s] [reason string]
```

Default Aliases

interrupt	stop-soar
ss	stop-soar
stop	stop-soar

Options

-s, --self	Stop only the soar agent where the command is issued. All other agents continue running as previously specified.
reason_string	An optional string which will be printed when Soar is stopped, to indicate why it was stopped. If left blank, no message will be printed when Soar is stopped.

Description

The stop-soar command stops any running Soar agents. It sets a flag in the Soar kernel so that Soar will stop running at a ¨safe¨ point and return control to the user. This command is usually not issued at the command line prompt - a more common use of this command would be, for instance, as a side-effect of pressing a button on a Graphical User Interface (GUI).

See Also

run

Warnings

If the graphical interface doesn't periodically do an ¨update¨ of flush the pending I/O, then it may not be possible to interrupt a Soar agent from the command line.

8.2   Examining Memory

This section describes the commands used to inspect production memory, working memory, and preference memory; to see what productions will match and fire in the next Propose or Apply phase; and to examine the goal dependency set. These commands are particularly

useful when running or debugging Soar, as they let users see what Soar is "thinking." The specific commands described in this section are:

Summary

> **default-wme-depth** - Set the level of detail used to print WMEs.
>
> **gds-print** - Print the WMEs in the goal dependency set for each goal.
>
> **internal-symbols** - Print information about the Soar symbol table.
>
> **matches** - Print information about the match set and partial matches.
>
> **memories** - Print memory usage for production matches.
>
> **preferences** - Examine items in preference memory.
>
> **print** - Print items in working memory or production memory.
>
> **production-find** - Find productions that contain a given pattern.

Of these commands, **print** is the most often used (and the most complex) followed by **matches** and **memories**. **preferences** is used to examine which candidate operators have been proposed. **production-find** is especially useful when the number of productions loaded is high. **gds-print** is useful for examining the goal dependecy set when subgoals seem to be disappearing unexpectedly. **default-wme-depth** is related to the **print** command. **internal-symbols** is not often used but is helpful when debugging Soar extensions or trying to locate memory leaks.

default-wme-depth

Set the level of detail used to print WMEs.

Synopsis

```
 default-wme-depth [depth]
```

Default Aliases

set-default-depth	default-wme-depth

Options

depth	A non-negative integer.

Description

The default-wme-depth command reflects the default depth used when working memory elements are printed (using the print). The default value is 1. When the command is issued with no arguments, default-wme-depth returns the current value of the default depth. When followed by an integer value, default-wme-depth sets the default depth to the specified value. This default depth can be overridden on any particular call to the print command by explicitly using the `--depth` flag, e.g. `print --depth 10 args`.

By default, the print command prints *objects* in working memory, not just the individual working memory element. To limit the output to individual working memory elements, the `--internal` flag must also be specified in the print command. Thus when the print depth is 0, by default Soar prints the entire object, which is the same behavior as when the print depth is 1. But if `--internal` is also specified, then a depth of 0 prints just the individual WME, while a depth of 1 prints all WMEs which share that same identifier. This is true when printing timetags, identifiers or WME patterns.

When the depth is greater than 1, the identifier links from the specified WME's will be followed, so that additional substructure is printed. For example, a depth of 2 means that the object specified by the identifier, wme-pattern, or timetag will be printed, along with all other objects whose identifiers appear as values of the first object. This may result in multiple copies of the same object being printed out. If `--internal` is also specified, then individuals WMEs and their timetags will be printed instead of the full objects.

See Also

print

gds-print

Print the WMEs in the goal dependency set for each goal.

Synopsis

```
 gds-print
```

Default Aliases

gds_print	gds-print

Description

The Goal Dependency Set (GDS) is described in an appendix of the Soar manual. This command is a debugging command for examining the GDS for each goal in the stack. First it steps through all the working memory elements in the rete, looking for any that are included in *any* goal dependency set, and prints each one. Then it also lists each goal in the stack and prints the wmes in the goal dependency set for that particular goal. This command is useful when trying to determine why subgoals are disappearing unexpectedly: often something has changed in the goal dependency set, causing a subgoal to be regenerated prior to producing a result.

Warnings

gds-print is horribly inefficient and should not generally be used except when something is going wrong and you need to examine the Goal Dependency Set.

internal-symbols

Print information about the Soar symbol table.

Synopsis

```
 internal-symbols
```

Description

The internal-symbols command prints information about the Soar symbol table. Such information is typically only useful for users attempting to debug Soar by locating memory leaks or examining I/O structure.

Example

```
soar> internal-symbols
--- Symbolic Constants: ---
operator
accept
evaluate-object
problem-space
sqrt
interrupt
mod
```

```
goal
io
(...additional symbols deleted for brevity...)
--- Integer Constants: ---
--- Floating-Point Constants: ---
--- Identifiers: ---
--- Variables: ---
<o>
<sso>
<to>
<ss>
<ts>
<so>
<sss>
```

matches

Prints information about partial matches and the match set.

Synopsis

```
matches [options] production_name
matches [options] -[a|r]
```

Options

production_name	Print partial match information for the named production.
-n, --names, -c, --count	For the match set, print only the names of the productions that are about to fire or retract (the default). If printing partial matches for a production, just list the partial match counts.
-t, --timetags	Also print the timetags of the wmes at the first failing condition
-w, --wmes	Also print the full wmes, not just the timetags, at the first failing condition.
-a, --assertions	List only productions about to fire.
-r, --retractions	List only productions about to retract.

Description

The matches command prints a list of productions that have instantiations in the match set, i.e., those productions that will retract or fire in the next Propose or Apply phase. It also

will print partial match information for a single, named production.

Printing the match set   When printing the match set (i.e., no production name is specified), the default action prints only the names of the productions which are about to fire or retract. If there are multiple instantiations of a production, the total number of instantiations of that production is printed after the production name, unless `--timetags` or `--wmes` are specified, in which case each instantiation is printed on a separate line.

When printing the match set, the `--assertions` and `--retractions` arguments may be specified to restrict the output to print only the assertions or retractions.

Printing partial matches for productions   In addition to printing the current match set, the matches command can be used to print information about partial matches for a named production. In this case, the conditions of the production are listed, each preceded by the number of currently active matches for that condition. If a condition is negated, it is preceded by a minus sign `-`. The pointer `>>>>` before a condition indicates that this is the first condition that failed to match.

When printing partial matches, the default action is to print only the counts of the number of WME's that match, and is a handy tool for determining which condition failed to match for a production that you thought should have fired. At levels `--timetags` and `--wmes` the matches command displays the WME's immediately after the first condition that failed to match–temporarily interrupting the printing of the production conditions themselves.

Notes   When printing partial match information, some of the matches displayed by this command may have already fired, depending on when in the execution cycle this command is called. To check for the matches that are about to fire, use the matches command without a named production.

In Soar 8, the execution cycle (decision cycle) is input, propose, decide, apply output; it no longer stops for user input after the decision phase when running by decision cycles (`run -d 1`). If a user wishes to print the match set immediately after the decision phase and before the apply phase, then the user must run Soar by *phases* (`run -p 1`).

Examples

This example prints the productions which are about to fire and the wmes that match the productions on their left-hand sides:

```
 matches --assertions --wmes
```

This example prints the wme timetags for a single production.

```
 matches -t my*first*production
```

memories

Print memory usage for partial matches.

Synopsis

```
memories [options] [number]
memories production_name
```

Options

`-c, --chunks`	Print memory usage of chunks.
`-d, --default`	Print memory usage of default productions.
`-j, --justifications`	Print memory usage of justifications.
`-u, --user`	Print memory usage of user-defined productions.
`production_name`	Print memory usage for a specific production.
`number`	Number of productions to print, sorted by those that use the most memory.
`-T, --template`	Print memory usage of Soar-RL templates.

Description

The memories command prints out the internal memory usage for full and partial matches of production instantiations, with the productions using the most memory printed first. With no arguments, the memories command prints memory usage for all productions. If a `production_name` is specified, memory usage will be printed only for that production. If a positive integer `number` is given, only `number` productions will be printed: the `number` productions that use the most memory. Output may be restricted to print memory usage for particular types of productions using the command options.

Memory usage is recorded according to the tokens that are allocated in the rete network for the given production(s). This number is a function of the number of elements in working memory that match each production. Therefore, this command will not provide useful information at the beginning of a Soar run (when working memory is empty) and should be called in the middle (or at the end) of a Soar run.

The memories command is used to find the productions that are using the most memory and, therefore, may be taking the longest time to match (this is only a heuristic). By identifying these productions, you may be able to rewrite your program so that it will run more quickly. Note that memory usage is just a heuristic measure of the match time: A production might not use much memory relative to others but may still be time-consuming to match, and excising a production that uses a large number of tokens may not speed up your program, because the Rete matcher shares common structure among different productions.

As a rule of thumb, numbers less than 100 mean that the production is using a small amount of memory, numbers above 1000 mean that the production is using a large amount of memory, and numbers above 10,000 mean that the production is using a **very** large amount of memory.

See Also

matches

preferences

Examine details about the preferences that support the specified *identifier* and *attribute*.

Synopsis

```
preferences [options] [identifier [attribute]]
```

Default Aliases

pref	preferences

Options

`-0, -n, --none`	Print just the preferences themselves
`-1, -N, --names`	Print the preferences and the names of the productions that generated them
`-2, -t, --timetags`	Print the information for the `--names` option above plus the timetags of the wmes matched by the LHS of the indicated productions
`-3, -w, --wmes`	Print the information for the `--timetags` option above plus the entire WME matched on the LHS.
`-o, --object`	Print the support for all the WMEs that comprise the object (the specified *identifier*).
`identifier`	Must be an existing Soar object *identifier*.
`attribute`	Must be an existing *attribute* of the specified *identifier*.

Description

The preferences command prints all the preferences for the given object identifier and attribute. If identifier and attribute are not specified, they default to the current state and the current operator. The Soar syntax attribute carat ($^$) is optional when specifying the attribute. The optional arguments indicates the level of detail to print about each preference.

This command is useful for examining which candidate operators have been proposed and what relationships, if any, exist among them. If a preference has O-support, the string, `:O` will also be printed.

When only the identifier is specified on the commandline, if the identifier is a state, Soar uses the default attribute $^\wedge$`operator`. If the identifier is not a state, Soar prints the support information for all WMEs whose value is the identifier.

When an identifier and the `--object` flag are specified, Soar prints the preferences / WME support for all WMEs comprising the specified identifier.

For the time being, [cmd_numeric_indifferent_mode numeric-indifferent] preferences are listed under the heading 'binary indifferents:'.

By default, using the `--wmes` option with a WME on the top state will only print the timetags. To change this, the kernel can be recompiled with `DO_TOP_LEVEL_REF_CTS`, but this has other consequences (see comments in `kernel.h`).

Examples

This example prints the preferences on (`S1` $^\wedge$`operator`) and the production names which created the preferences:

```
soar> preferences S1 operator --names
Preferences for S1 ^operator:
acceptables:
 O2 (fill) +
   From waterjug*propose*fill
 O3 (fill) +
   From waterjug*propose*fill

unary indifferents:
 O2 (fill) =
   From waterjug*propose*fill
 O3 (fill) =
   From waterjug*propose*fill
```

If the current state is `S1`, then the above syntax is equivalent to:

```
preferences -n
```

This example shows the support for the WMEs with the $^\wedge$jug attribute:

```
soar> preferences s1 jug

Preferences for S1 ^jug:

acceptables:
  (S1 ^jug I4)  :O
```

```
   (S1 ^jug J1)   :O
```

This example shows the support for the WMEs with value J1, and the productions that generated them:

```
 soar> pref J1 -1

  Support for (31: O3 ^jug J1)
    (O3 ^jug J1)
      From water-jug*propose*fill

  Support for (11: S1 ^jug J1)
    (S1 ^jug J1)   :O
      From water-jug*apply*initialize-water-jug-look-ahead
```

This example shows the support for all WMEs that make up the object S1:

```
 soar> pref -o s1

 Support for S1 ^problem-space:
   (S1 ^problem-space P1)
 Support for S1 ^name:
   (S1 ^name water-jug)   :O
 Support for S1 ^jug:
   (S1 ^jug I4)   :O
   (S1 ^jug J1)   :O
 Support for S1 ^desired:
   (S1 ^desired D1)   :O
 Support for S1 ^superstate-set:
   (S1 ^superstate-set nil)
 Preferences for S1 ^operator:
 acceptables:
   O2 (fill) +
   O3 (fill) +
 Arch-created wmes for S1 :
 (2: S1 ^superstate nil)
 (1: S1 ^type state)
 Input (IO) wmes for S1 :
 (3: S1 ^io I1)
```

print

Print items in working memory or production memory.

Synopsis

```
print [options] [production_name]
print [options] identifier|timetag|pattern
```

Default Aliases

p	print
pc	print --chunks
wmes	print --internal

Options

Printing items in production memory

What productions to print:

-a, --all	print the names of all productions currently loaded
-c, --chunks	print the names of all chunks currently loaded
-D, --defaults	print the names of all default productions currently loaded
-j, --justifications	print the names of all justifications currently loaded.
-r, --rl	Print Soar-RL rules
-T, --template	Print Soar-RL templates
-u, --user	print the names of all user productions currently loaded
production_name	print the production named `production-name`

How to print the productions:

-f, --full	When printing productions, print the whole production. This is the default when printing a named production.
-F, --filename	also prints the name of the file that contains the production.
-i, --internal	items should be printed in their internal form. For productions, this means leaving conditions in their reordered (rete net) form.
-n, --name	When printing productions, print only the name and not the whole production. This is the default when printing any category of productions, as opposed to a named production.

Printing items in working memory

`-d, --depth n`	This option overrides the default printing depth (see the default-wme-depth command for more detail).
`-e, --exact`	Print only the wmes that match the pattern
`-i, --internal`	items should be printed in their internal form. For working memory, this means printing the individual elements with their timetags and activation, rather than the objects.
`-t, --tree`	wmes should be printed in in a tree form (one wme per line).
`-v, --varprint`	Print identifiers enclosed in angle brackets.
`identifier`	print the object `identifier`. `identifier` must be a valid Soar symbol such as *S1*
`pattern`	print the object whose working memory elements matching the given `pattern`. See Description for more information on printing objects matching a specific `pattern`.
`timetag`	print the object in working memory with the given `timetag`

Printing the current subgoal stack

`-s, --stack`	Specifies that the Soar goal stack should be printed. By default this includes both states and operators.
`-o, --operators`	When printing the stack, print only *operators*.
`-S, --states`	When printing the stack, print only *states*.

Description

The print command is used to print items from production memory or working memory. It can take several kinds of arguments. When printing items from working memory, the Soar objects are printed unless the `--internal` flag is used, in which case the wmes themselves are printed.

```
(identifier ^attribute value [activation] [+])
```

The activation value is only printed if activation is turned on. See wma.

The pattern is surrounded by parentheses. The `identifier`, `attribute`, and `value` must be valid Soar symbols or the wildcard symbol * which matches all occurrences. The optional + symbol restricts pattern matches to acceptable preferences. If wildcards are included, an object will be printed for each pattern match, even if this results in the same object being printed multiple times.

Examples

Print the objects in working memory (and their timetags) which have wmes with identifier `s1` and value `v2` (note: this will print the entire `s1` object for each match found):

```
print --internal (s1 ^* v2)
```

Print the Soar stack which includes states and operators:

```
 print --stack
```

Print the named production in its RETE form:

```
 print -if named*production
```

Print the names of all user productions currently loaded:

```
 print -u
```

Default print vs tree print:

```
 print s1 --depth 2
 (S1 ^io I1 ^reward-link R1 ^superstate nil ^type state)
   (I1 ^input-link I2 ^output-link I3)

 print s1 --depth 2 --tree
 (S1 ^io I1)
   (I1 ^input-link I2)
   (I1 ^output-link I3)
 (S1 ^reward-link R1)
 (S1 ^superstate nil)
 (S1 ^type state)
```

See Also

default-wme-depth wma

production-find

Find productions by condition or action patterns.

Synopsis

```
 production-find [-lrs[n|c]] pattern
```

Options

`-c, --chunks`	Look *only* for chunks that match the pattern.
`-l, --lhs`	Match pattern only against the conditions (left-hand side) of productions (default).
`-n, --nochunks`	*Disregard* chunks when looking for the pattern.
`-r, --rhs`	Match pattern against the actions (right-hand side) of productions.
`-s, --show-bindings`	Show the bindings associated with a wildcard pattern.
`pattern`	Any pattern that can appear in productions.

Description

The production-find command is used to find productions in production memory that include conditions or actions that match a given `pattern`. The pattern given specifies one or more condition elements on the left hand side of productions (or negated conditions), or one or more actions on the right-hand side of productions. Any pattern that can appear in productions can be used in this command. In addition, the asterisk symbol, `*`, can be used as a wildcard for an attribute or value. It is important to note that the whole pattern, including the parenthesis, must be enclosed in curly braces for it to be parsed properly.

The variable names used in a call to production-find do not have to match the variable names used in the productions being retrieved.

The production-find command can also be restricted to apply to only certain types of productions, or to look only at the conditions or only at the actions of productions by using the flags.

Examples

Find productions that test that some object `gumby` has an attribute `alive` with value `t`. In addition, limit the rules to only those that test an operator named `foo`:

```
production-find (<state> ^gumby <gv> ^operator.name foo)(<gv> ^alive t)
```

Note that in the above command, `<state>` does not have to match the exact variable name used in the production.

Find productions that propose the operator `foo`:

```
production-find --rhs (<x> ^operator <op> +)(<op> ^name foo)
```

Find chunks that test the attribute ^pokey:

```
production-find --chunks (<x> ^pokey *)
```

Examples using the water-jugs demo:

```
source demos/water-jug/water-jug.soar
```

```
production-find (<s> ^name *)(<j> ^volume *)
production-find (<s> ^name *)(<j> ^volume 3)
production-find --rhs (<j> ^* <volume>)
```

See Also

sp

8.3   Configuring Trace Information and Debugging

This section describes the commands used primarily for debugging or to configure the trace output printed by Soar as it runs. Users may: specify the content of the runtime trace output; ask that they be alerted when specific productions fire and retract; or request details on Soar's performance.

The specific commands described in this section are:

Summary

> **chunk-name-format** - Specify format of the name to use for new chunks.
>
> **firing-counts** - Print the number of times productions have fired.
>
> **pwatch** - Trace firings and retractions of specific productions.
>
> **stats** - Print information on Soar's runtime statistics.
>
> **verbose** - Control detailed information printed as Soar runs.
>
> **warnings** - Toggle whether or not warnings are printed.
>
> **watch** - Control the information printed as Soar runs.
>
> **watch-wmes** - Print information about wmes that match a certain pattern as they are added and removed

Of these commands, `watch` is the most often used (and the most complex). `pwatch` is related to `watch`, but applies only to specific, named productions. `firing-counts` and `stats` are useful for understanding how much work Soar is doing. `chunk-name-format` is less-frequently used, but allows for detailed control of Soar's chunk naming.

chunk-name-format

Specify format of the name to use for new chunks.

Synopsis

```
chunk-name-format [-sl] -p [prefix]
chunk-name-format [-sl] -c [count]
```

Options

`-s, --short`	Use the short format for naming chunks
`-l, --long`	Use the long format for naming chunks (default)
`-p, --prefix [p]`	If given, use `p` as the prefix for naming chunks. Otherwise, return the current prefix. (defaults to *chunk*)
`-c, --count [c]`	If given, set the chunk counter for naming chunks to c. Otherwise, return the current value of the chunk counter.

Description

The short format for naming newly-created chunks is:

```
<prefix><chunknum>
```

The long (default) format for naming chunks is:

```
<prefix>-<chunknum>*<dc>*<impassetype>*<dcChunknum>
```

where:

- `prefix` is a user-definable prefix string; `prefix` defaults to *chunk* when unspecified by the user. It many not contain the character `*`,

- `chunknum` is a counter set by `--count` or starting at 1 for the first chunk created,

- `dc` is the number of the decision cycle in which the chunk was formed,

- `impassetype` is one of `tie`, `conflict`, `cfailure`, `snochange`, `opnochange`,

- `dcChunknum` is the number of the chunk within that specific decision cycle.

firing-counts

Print the number of times each production has fired.

Synopsis

```
firing-counts [n]
firing-counts production_name
```

Default Aliases

fc	firing-counts

Options

If given, an option can take one of two forms – an integer or a production name:

n	List the top **n** productions. If **n** is 0, only the productions which haven't fired are listed
production_name	Print how many times the production has fired

Description

The firing-counts command prints the number of times each production has fired; production names are given from most requently fired to least frequently fired. With no arguments, it lists all productions. If an integer argument, **n**, is given, only the top **n** productions are listed. If **n** is zero (0), only the productions that haven't fired at all are listed. If a production name is given as an argument, the firing count for that production is printed.

Note that firing counts are reset by a call to [cmd_init_soar init-soar].

Examples

This example prints the 10 productions which have fired the most times along with their firing counts:

```
 firing-counts 10
```

This example prints the firing counts of production `my*first*production`:

```
 firing-counts my*first*production
```

Warnings

Firing-counts are reset to zero after an [cmd_init_soar init-soar].

NB: This command is slow, because the sorting takes time `O(n*log n)`

See Also

init-soar

pwatch

Trace firings and retractions of specific productions.

Synopsis

```
pwatch [-d|e] [production name]
```

Default Aliases

pw	pwatch

Options

-d, --disable, --off	Turn production watching off for the specified production. If no production is specified, turn production watching off for all productions.
-e, --enable, --on	Turn production watching on for the specified production. The use of this flag is optional, so this is pwatch's default behavior. If no production is specified, all productions currently being watched are listed.
production name	The name of the production to watch.

Description

The pwatch command enables and disables the tracing of the firings and retractions of individual productions. This is a companion command to watch, which cannot specify individual productions by name.

With no arguments, pwatch lists the productions currently being traced. With one production-name argument, pwatch enables tracing the production; `--enable` can be explicitly stated, but it is the default action.

If `--disable` is specified followed by a production-name, tracing is turned off for the production. When no production-name is specified, `--enable` lists all productions currently being traced, and `--disable` disables tracing of all productions.

Note that pwatch now only takes one production per command. Use multiple times to watch multiple functions.

See Also

watch

stats

Print information on Soar's runtime statistics.

Synopsis

```
stats [options]
```

Default Aliases

st	stats

Options

-m, --memory	report usage for Soar's memory pools
-r, --rete	report statistics about the rete structure
-s, --system	report the system (agent) statistics (default)
-M, --max	report the per-cycle maximum statistics (decision cycle time, WM changes, production fires)
-R, --reset	zero out the per-cycle maximum statistics reported by --max command
-t, --track	begin tracking the per-cycle maximum statistics reported by --max for each cycle (instead of only the max value)
-T, --stop-track	stop and clear tracking of the per-cycle maximum statistics
-c, --cycle	print out collected per-cycle maximum statistics saved by --track in human-readable form
-C, --cycle-csv	print out collected per-cycle maximum statistics saved by --track in comma-separated form
-S, --sort N	sort the tracked cycle stats by column number N, see table below

Tracked Cycle Stats Columns   For use with `--sort` option. **Negative values sorts descending.**

0	Use default sort
1, -1	Sort by decision cycle (use negative for descending)
2, -2	Sort by DC time (use negative for descending)
3, -3	Sort by WM changes (use negative for descending)
4, -4	Sort by production firings (use negative for descending)

Description

This command prints Soar internal statistics. The argument indicates the component of interest, `--system` is used by default.

With the `--system` flag, the stats command lists a summary of run statistics, including the following:

- **Version** — The Soar version number, hostname, and date of the run.

- **Number of productions** — The total number of productions loaded in the system, including all chunks built during problem solving and all default productions.

- **Timing Information** — Might be quite detailed depending on the flags set at compile time. See note on timers below.

- **Decision Cycles** — The total number of decision cycles in the run and the average time-per-decision-cycle in milliseconds.

- **Elaboration cycles** — The total number of elaboration cycles that were executed during the run, the average number of elaboration cycles per decision cycle, and the average time-per-elaboration-cycle in milliseconds. This is not the total number of production firings, as productions can fire in parallel.

- **Production Firings** — The total number of productions that were fired.

- **Working Memory Changes** — This is the total number of changes to working memory. This includes all additions and deletions from working memory. Also prints the average match time.

- **Working Memory Size** — This gives the current, mean and maximum number of working memory elements.

The stats argument `--memory` provides information about memory usage and Soar's memory pools, which are used to allocate space for the various data structures used in Soar.

The stats argument `--rete` provides information about node usage in the Rete net, the large data structure used for efficient matching in Soar.

The `--max` argument reports per-cycle maximum statistics for decision cycle time, working memory changes, and production fires. For example, if Soar runs for three cycles and there were 23 working memory changes in the first cycle, 42 in the second, and 15 in the third, the `--max` argument would report the highest of these values (42) and what decision cycle that it occurred in (2nd). Statistics about the time spent executing the decision cycle and number of productions fired are also collected and reported by `--max` in this manner. `--reset` zeros out these statistics so that new maximums can be recorded for future runs. The numbers are also zeroed out with a call to init-soar.

The `--track` argument starts tracking the same stats as the `--max` argument but records all data for each cycle instead of the maximum values. This data can be printed using the `--cycle` or `--cycle-csv` arguments. When printing the data with `--cycle`, it may be sorted using the `--sort` argument and a column integer. Use negative numbers for descending sort. Issue `--stop-track` to reset and clear this data.

A Note on Timers   The current implementation of Soar uses a number of timers to provide time-based statistics for use in the stats command calculations. These timers are:

- total CPU time

- total kernel time

- phase kernel time (per phase)

- phase callbacks time (per phase)

- input function time

- output function time

Total CPU time is calculated from the time a decision cycle (or number of decision cycles) is initiated until stopped. Kernel time is the time spent in core Soar functions. In this case, kernel time is defined as the all functions other than the execution of callbacks and the input and output functions. The total kernel timer is only stopped for these functions. The phase timers (for the kernel and callbacks) track the execution time for individual phases of the decision cycle (i.e., input phase, preference phase, working memory phase, output phase, and decision phase). Because there is overhead associated with turning these timers on and off, the actual kernel time will always be greater than the derived kernel time (i.e., the sum of all the phase kernel timers). Similarly, the total CPU time will always be greater than the derived total (the sum of the other timers) because the overhead of turning these timers on and off is included in the total CPU time. In general, the times reported by the single timers should always be greater than than the corresponding derived time. Additionally, as execution time increases, the difference between these two values will also increase. For those concerned about the performance cost of the timers, all the run time timing calculations can be compiled out of the code by defining `NO_TIMING_STUFF` (in `soarkernel.h`) before compilation.

Examples

Track per-cycle stats then print them out using default sort:

```
stats --track
run
stop
stats --cycle
```

Print out per-cycle stats sorting by decision cycle time

```
stats --cycle --sort 2
```

Print out per-cycle stats sorting by firing counts, descending

```
stats --cycle --sort -4
```

Save per-cycle stats to file `stats.csv`

```
 ctf stats.csv stats --cycle-csv
```

See Also

timers init-soar command-to-file

verbose

Control detailed information printed as Soar runs.

Synopsis

```
 verbose [-ed]
```

Options

-d, --disable, --off	Turn verbosity off. (default)
-e, --enable, --on	Turn verbosity on.

Description

The verbose command enables tracing of a number of low-level Soar execution details during a run. The details printed by verbose are usually only valuable to developers debugging Soar implementation details. Invoke with no arguments to query the current setting.

See Also

watch

warnings

Enable or disable the printing of warning messages from the Soar kernel.

Synopsis

```
 warnings [options]
```

Options

`-e, --enable, --on`	Default. Print all warning messages from the kernel.
`-d, --disable, --off`	Disable all, except most critical, warning messages.

Description

Enables and disables the printing of warning messages. If an argument is specified, then the warnings are set to that state. If no argument is given, then the current warnings status is printed. At startup, warnings are initially enabled. If warnings are disabled using this command, then some warnings may still be printed, since some are considered too important to ignore.

The warnings that are printed apply to the syntax of the productions, to notify the user when they are not in the correct syntax. When a lefthand side error is discovered (such as conditions that are not linked to a common state or impasse object), the production is generally loaded into production memory anyway, although this production may never match or may seriously slow down the matching process. In this case, a warning would be printed only if warnings were `--on`. Righthand side errors, such as preferences that are not linked to the state, usually result in the production not being loaded, and a warning regardless of the warnings setting.

watch

Control the run-time tracing of Soar.

Synopsis

```
watch [options]
watch level
```

Default Aliases

`w`	`watch`

Options

When appropriate, a specific option may be turned off using the `remove` argument. This argument has a numeric alias; you can use `0` for `remove`. A mix of formats is acceptable, even in the same command line.

Basic Watch Settings

Option Flag	Argument to Option	Description
`-l, --level`	`0` to `5` (see *Watch Levels* below)	This flag is optional but recommended. Set a specific watch level using an integer `0` to `5`, this is an inclusive operation
`-N, --none`	No argument	Turns off all printing about Soar's internals, equivalent to `--level 0`
`-d, --decisions`	`remove` (optional)	Controls whether state and operator decisions are printed as they are made
`-p, --phases`	`remove` (optional)	Controls whether decisions cycle phase names are printed as Soar executes
`-g, --gds`	`remove` (optional)	Controls printing of warnings about wme changes to GDS
`-P, --productions`	`remove` (optional)	Controls whether the names of productions are printed as they fire and retract, equivalent to `-Dujc`
`-w, --wmes`	`remove` (optional)	Controls the printing of working memory elements that are added and deleted as productions are fired and retracted.
`-r, --preferences`	`remove` (optional)	Controls whether the preferences generated by the traced productions are printed when those productions fire or retract

Watch Levels   Use of the `--level` (`-l`) flag is optional but recommended.

0	watch nothing; equivalent to `-N`
1	watch decisions; equivalent to `-d`
2	watch phases, gds, and decisions; equivalent to `-dpg`
3	watch productions, phases, and decisions; equivalent to `-dpgP`
4	watch wmes, productions, phases, and decisions; equivalent to `-dpgPw`
5	watch preferences, wmes, productions, phases, and decisions; equivalent to `-dpgPwr`

It is important to note that watch level `0` turns off ALL watch options, including backtracing, indifferent selection and learning. However, the other watch levels do not change these settings. That is, if any of these settings is changed from its default, it will retain its new setting until it is either explicitly changed again or the watch level is set to `0`.

Watching Productions By default, the names of the productions are printed as each production fires and retracts (at watch levels 3 and higher). However, it may be more helpful to watch only a specific *type* of production. The tracing of firings and retractions of productions can be limited to only certain types by the use of the following flags:

Option Flag	Argument to Option	Description
-D, --default	remove (optional)	Control only default-productions as they fire and retract
-u, --user	remove (optional)	Control only user-productions as they fire and retract
-c, --chunks	remove (optional)	Control only chunks as they fire and retract
-j, --justifications	remove (optional)	Control only justifications as they fire and retract
-T, --template	remote (optional)	Soar-RL template firing trace

Note: The pwatch command is used to watch individual productions specified by name rather than watch a type of productions, such as --user.

Additionally, when watching productions, users may set the level of detail to be displayed for WMEs that are added or retracted as productions fire and retract. Note that detailed information about WMEs will be printed only for productions that are being watched.

Option Flag	Description
-n, --nowmes	When watching productions, do not print any information about matching wmes
-t, --timetags	When watching productions, print only the timetags for matching wmes
-f, --fullwmes	When watching productions, print the full matching wmes

Watching Learning

Option Flag	Argument to Option	Description
-L, --learning	noprint, print, or fullprint (see table below)	Controls the printing of chunks/justifications as they are created

As Soar is running, it may create justifications and chunks which are added to production memory. The watch command allows users to monitor when chunks and justifications are created by specifying one of the following arguments to the --learning command:

Argument	Alias	Effect
`noprint`	0	Print nothing about new chunks or justifications (default)
`print`	1	Print the names of new chunks and justifications when created
`fullprint`	2	Print entire chunks and justifications when created

Watching other Functions

Option Flag	Argument to Option	Description
`-a, --wma`	`remove` (optional)	Print log of working memory activation events
`-b, --backtracing`	`remove` (optional)	Print backtracing information when a chunk or justification is created
`-e, --epmem`	`remove` (optional)	Print episodic retrieval traces and IDs of newly encoded episodes
`-i, --indifferent-selection`	`remove` (optional)	Print scores for tied operators in random indifferent selection mode
`-R, --rl`	`remove` (optional)	Print RL debugging output
`-s, --smem`	`remove` (optional)	Print log of semantic memory storage events

Description

The watch command controls the amount of information that is printed out as Soar runs. The basic functionality of this command is to trace various *levels* of information about Soar's internal workings. The higher the *level*, the more information is printed as Soar runs. At the lowest setting, `0` (`--none`), nothing is printed. The levels are cumulative, so that each successive level prints the information from the previous level as well as some additional information. The default setting for the *level* is `1`, (`--decisions`).

The numerical arguments *inclusively* turn on all levels up to the number specified. To use numerical arguments to turn off a level, specify a number which is less than the level to be turned off. For instance, to turn off watching of productions, specify `--level 2` (or 1 or 0). Numerical arguments are provided for shorthand convenience. For more detailed control over the watch settings, the named arguments should be used.

With no arguments, this command prints information about the current watch status, i.e., the values of each parameter.

For the named arguments, including the named argument turns on only that setting. To

turn off a specific setting, follow the named argument with `remove` or `0`.

The named argument `--productions` is shorthand for the four arguments `--default`, `--user`, `--justifications`, and `--chunks`.

Examples

The most common uses of watch are by using the numeric arguments which indicate watch levels. To turn off all printing of Soar internals, do any one of the following (not all possibilities listed):

```
watch --level 0
watch -l 0
watch -N
```

Although the `--level` flag is optional, its use is recommended:

```
watch --level 5    # OK
watch 5            # OK, avoid
```

Be careful of where the level is on the command line, for example, if you want level 2 and preferences:

```
watch -r -l 2 # Incorrect: -r flag ignored, level 2 parsed after it and overrides the s
watch -r 2    # Syntax error: 0 or remove expected as optional argument to -r
watch -r -l 2 # Incorrect: -r flag ignored, level 2 parsed after it and overrides the s
watch 2 -r    # OK, avoid
watch -l 2 -r # OK
```

To turn on printing of decisions, phases and productions, do any one of the following (not all possibilities listed):

```
watch --level 3
watch -l 3
watch --decisions --phases --productions
watch -d -p -P
```

Individual options can be changed as well. To turn on printing of decisions and wmes, but not phases and productions, do any one of the following (not all possibilities listed):

```
watch --level 1 --wmes
watch -l 1 -w
watch --decisions --wmes
watch -d --wmes
watch -w --decisions
watch -w -d
```

To turn on printing of decisions, productions and wmes, and turns phases off, do any one of the following (not all possibilities listed):

```
watch --level 4 --phases remove
watch -l 4 -p remove
watch -l 4 -p 0
watch -d -P -w -p remove
```

To watch the firing and retraction of decisions and *only* user productions, do any one of the following (not all possibilities listed):

```
watch -l 1 -u
watch -d -u
```

To watch decisions, phases and all productions *except* user productions and justifications, and to see full wmes, do any one of the following (not all possibilities listed):

```
watch --decisions --phases --productions --user remove --justifications remove --fullwm
watch -d -p -P -f -u remove -j 0
watch -f -l 3 -u 0 -j 0
```

See Also

epmem pwatch print run watch-wmes

watch-wmes

Print information about wmes matching a certain pattern as they are added and removed.

Synopsis

```
watch-wmes -[a|r]  -t type  pattern
watch-wmes -[l|R] [-t type]
```

Options

-a, --add-filter	Add a filter to print wmes that meet the type and pattern criteria.
-r, --remove-filter	Delete filters for printing wmes that match the type and pattern criteria.
-l, --list-filter	List the filters of this type currently in use. Does not use the pattern argument.
-R, --reset-filter	Delete all filters of this type. Does not use pattern arg.
-t, --type	Follow with a type of wme filter, see below.

Pattern   The pattern is an id-attribute-value triplet:

```
id attribute value
```

Note that * can be used in place of the id, attribute or value as a wildcard that maches any string. Note that braces are not used anymore.

Types   When using the -t flag, it must be followed by one of the following:

adds	Print info when a wme is `added`.
removes	Print info when a wme is `retracted`.
both	Print info when a wme is added `or` retracted.

When issuing a `-R` or `-l`, the `-t` flag is optional. Its absence is equivalent to `-t both`.

Description

This commands allows users to improve state tracing by issuing filter-options that are applied when watching wmes. Users can selectively define which `object-attribute-value` triplets are monitored and whether they are monitored for addition, removal or both, as they go in and out of working memory.

Note: The functionality of watch-wmes resided in the watch command prior to Soar 8.6.

Examples

Users can watch an `attribute` of a particular object (as long as that object already exists):

```
 soar> watch-wmes --add-filter -t both D1 speed *
```

or print WMEs that retract in a specific state (provided the `state` already exists):

```
 soar> watch-wmes --add-filter -t removes S3 * *
```

or watch any relationship between objects:

```
 soar> watch-wmes --add-filter -t both * ontop *
```

8.4   Configuring Soar's Runtime Parameters

This section describes the commands that control Soar's Runtime Parameters. Many of these commands provide options that simplify or restrict runtime behavior to enable easier and more localized debugging. Others allow users to select alternative algorithms or methodologies. Users can configure Soar's learning mechanism; examine the backtracing information that supports chunks and justifications; provide hints that could improve the efficiency of the Rete matcher; limit runaway chunking and production firing; choose an alternative algorithm for determining whether a working memory element receives O-support; and configure options for selecting between mutually indifferent operators.

The specific commands described in this section are:

Summary

epmem - Get/Set episodic memory parameters and statistics

explain-backtraces - Print information about chunk and justification back-traces.

indifferent-selection - Controls indifferent preference arbitration.

learn - Set the parameters for chunking, Soar's learning mechanism.

max-chunks - Limit the number of chunks created during a decision cycle.

max-dc-time - Set a wall-clock time limit such that the agent will be interrupted when a single decision cycle exceeds this limit.

max-elaborations - Limit the maximum number of elaboration cycles in a given phase.

max-goal-depth - Limit the sub-state stack depth.

max-memory-usage - Set the number of bytes that when exceeded by an agent, will trigger the memory usage exceeded event.

max-nil-output-cycles - Limit the maximum number of decision cycles executed without producing output.

multi-attributes - Declare multi-attributes so as to increase Rete matching efficiency.

numeric-indifferent-mode - Select method for combining numeric preferences.

o-support-mode - Choose experimental variations of o-support.

predict - Predict the next selected operator

rl - Get/Set RL parameters and statistics

save-backtraces - Save trace information to explain chunks and justifications.

select - Force the next selected operator

set-stop-phase - Controls the phase where agents stop when running by decision.

smem - Get/Set semantic memory parameters and statistics

timers - Toggle on or off the internal timers used to profile Soar.

waitsnc - Generate a wait state rather than a state-no-change impasse.

wma - Get/Set working memory activation parameters

epmem

Control the behavior of episodic memory.

Synopsis

```
epmem
epmem -e|--enable|--on
epmem -d|--disable|--off
epmem -i|--init
epmem -g|--get <parameter>
epmem -s|--set <parameter> <value>
epmem -S|--stats [<statistic>]
epmem -t|--timers [<timer>]
epmem -v|--viz <episode id>
epmem -p|--print <episode id>
epmem -b|--backup <file name>
```

Options

`-e, --enable, --on`	Enable episodic memory.
`-d, --disable, --off`	Disable episodic memory.
`-i, --init`	Re-initialize episodic memory
`-g, --get`	Print current parameter setting
`-s, --set`	Set parameter value
`-S, --stats`	Print statistic summary or specific statistic
`-t, --timers`	Print timer summary or specific statistic
`-v, --viz`	Print episode in graphviz format
`-p, --print`	Print episode in user-readable format
`-b, --backup`	Creates a backup of the episodic database on disk

Description

The `epmem` command is used to change all behaviors of the episodic memory module, except for watch output, which is controlled by the `watch --epmem` command.

Parameters   Due to the large number of parameters, the `epmem` command uses the `--get|--set <parameter> <value>` convention rather than individual switches for each parameter. Running `epmem` without any switches displays a summary of the parameter settings.

Main Parameters

Parameter	Description	Possible values	Default
append-database	Controls whether database is over-written or appended when opening or re-initializing	on, off	off
balance	Linear weight of match cardinality (1) vs. working memory activation (0) used in calculating match score	[0, 1]	1
database	Database storage method	file, memory	memory
exclusions	Toggle the exclusion of an attribute string constant	*any string*	epmem, smem
force	Forces episode encoding/ignoring in the next storage phase	ignore, remember, off	off
learning	Episodic memory enabled	on, off	off
merge	Controls how retrievals interact with long-term identifiers in working memory	none, add	none
path	Location of database file	*empty, some path*	*empty*
phase	Decision cycle phase to encode new episodes and process epmem link commands	output, selection	output
trigger	How episode encoding is triggered	dc, output, none	output

Performance Parameters

Parameter	Description	Possible values	Default
`cache-size`	Number of memory pages used in the SQLite cache	1, 2, ...	10000
`graph-match`	Graph matching enabled	`on`, `off`	`on`
`graph-match-ordering`	Ordering of identifiers during graph match	`undefined`, `dfs`, `mcv`	`undefined`
`lazy-commit`	Delay writing semantic store changes to file until agent exits	`on`, `off`	`on`
`optimization`	Policy for committing data to disk	`safety`, `performance`	`performance`
`page-size`	Size of each memory page used in the SQLite cache	1k, 2k, 4k, 8k, 16k, 32k, 64k	8k
`timers`	Timer granularity	`off`, `one`, `two`, `three`	`off`

The `learning` parameter turns the episodic memory module on or off. When `learning` is set to `off`, no new episodes are encoded and no commands put on the epmem link are processed. This is the same as using the enable and disable commands.

The `phase` parameter determines which decision cycle phase episode encoding and retrieval will be performed.

The `trigger` parameter controls when new episodes will be encoded. When it is set to `output`, new episodes will be encoded only if the agent made modifications to the output-link during that decision cycle. When set to 'dc', new episodes will be encoded every decision cycle.

The `exclusions` parameter can be used to prevent episodic memory from encoding parts of working memory into new episodes. The value of `exclusions` is a list of string constants. During encoding, episodic memory will walk working memory starting from the top state identifier. If it encounters a WME whose attribute is a member of the `exclusions` list, episodic memory will ignore that WME and abort walking the children of that WME, and they will not be included in the encoded episode. Note that if the children of the excluded WME can be reached from top state via an alternative non-excluded path, they will still be included in the encoded episode. The `exclusions` parameter behaves differently from other parameters in that issuing `epmem --set exclusions <val>` does not set its value to `<val>`. Instead, it will toggle the membership of `<val>` in the `exclusions` list.

The `path` parameter specifies the file system path the database is stored in. When `path` is empty, the database is stored in main memory and will be lost when the agent exits. When `path` is set to a valid file system path, the SQLite database is written to that path.

The append-database parameter will determine whether all existing episodes recorded in a database on disk will be erased when epmem loads it. Note that this affects episodic memory re-initialization also, i.e. if the append setting is off, all episodic memories stored to disk will be lost when an init-soar is performed.

Note that changes to database, path and append-database will not have an effect until the database is used after an initialization. This happens either at launch or when an init-soar or database initialization command is issued. To switch databases or database storage types on the fly, set your new parameters and then either perform an init-soar or `epmem --init`.

The `epmem --backup` command can be used to make a copy of the current state of the database, whether in memory or on disk. This command will commit all outstanding changes before initiating the copy.

When the database is stored to disk, the `lazy-commit` and `optimization` parameters control how often cached database changes are written to disk. These parameters trade off safety in the case of a program crash with database performance. When `optimization` is set to `performance`, the agent will have an exclusive lock on the database, meaning it cannot be opened concurrently by another SQLite process such as SQLiteMan. The lock can be relinquished by setting the database to memory or another database and issuing init-soar/`epmem --init` or by shutting down the Soar kernel.

The `balance` parameter sets the linear weight of match cardinality vs. cue activation. As a performance optimization, when the value is 1 (default), activation is not computed. If this value is not 1 (even close, such as 0.99), and working memory activation is enabled, this value will be computed for each leaf WME, which may incur a noticeable cost, depending upon the overall complexity of the retrieval.

The `graph-match-ordering` parameter sets the heuristic by which identifiers are ordered during graph match (assuming `graph-match` is `on`). The default, `undefined`, does not enforce any order and may be sufficient for small cues. For more complex cues, there will be a one-time sorting cost, during each retrieval, if the parameter value is changed. The currently available heuristics are depth-first search (`dfs`) and most-constrained variable (`mcv`). It is advised that you attempt these heuristics to improve performance if the `query_graph_match` timer reveals that graph matching is dominating retrieval time.

The `merge` parameter controls how the augmentations of retrieved long-term identifiers (LTIs) interact with an existing LTI in working memory. If the LTI is not in working memory or has no augmentations in working memory, this parameter has no effect. If the augmentation is in working memory and has augmentations, by default (`none`), episodic memory will not augment the LTI. If the parameter is set to `add` then any augmentations that augmented the LTI in a retrieved episode are added to working memory.

Statistics Episodic memory tracks statistics over the lifetime of the agent. These can be accessed using `epmem --stats <statistic>`. Running `epmem --stats` without a statistic will list the values of all statistics. Unlike timers, statistics will always be updated. Available statistics are:

Name	Label	Description
`time`	Time	Current episode ID
`db-lib-version`	SQLite Version	SQLite library version
`mem-usage`	Memory Usage	Current SQLite memory usage in bytes
`mem-high`	Memory Highwater	High SQLite memory usage watermark in bytes
`queries`	Queries	Number of times the **query** command has been processed
`nexts`	Nexts	Number of times the **next** command has been processed
`prevs`	Prevs	Number of times the **previous** command has been processed
`ncb-wmes`	Last Retrieval WMEs	Number of WMEs added to working memory in last reconstruction
`qry-pos`	Last Query Positive	Number of leaf WMEs in the **query** cue of last cue-based retrieval
`qry-neg`	Last Query Negative	Number of leaf WMEs in the **neg-query** cue of the last cue-based retrieval
`qry-ret`	Last Query Retrieved	Episode ID of last retrieval
`qry-card`	Last Query Cardinality	Match cardinality of last cue-based retrieval
`qry-lits`	Last Query Literals	Number of literals in the DNF graph of last cue-based retrieval

Timers   Episodic memory also has a set of internal timers that record the durations of certain operations. Because fine-grained timing can incur runtime costs, episodic memory timers are off by default. Timers of different levels of detail can be turned on by issuing `epmem --set timers <level>`, where the levels can be `off`, `one`, `two`, or `three`, `three` being most detailed and resulting in all timers being turned on. Note that none of the episodic memory statistics nor timing information is reported by the `stats` command.

All timer values are reported in seconds.

Level one

`_total`	Total epmem operations

Level two

`epmem_api`	Agent command validation
`epmem_hash`	Hashing symbols
`epmem_init`	Episodic store initialization
`epmem_ncb_retrieval`	Episode reconstruction
`epmem_next`	Determining next episode
`epmem_prev`	Determining previous episode
`epmem_query`	Cue-based query
`epmem_storage`	Encoding new episodes
`epmem_trigger`	Deciding whether new episodes should be encoded
`epmem_wm_phase`	Converting preference assertions to working memory changes

Level three

`ncb_edge`	Collecting edges during reconstruction
`ncb_edge_rit`	Collecting edges from relational interval tree
`ncb_node`	Collecting nodes during reconstruction
`ncb_node_rit`	Collecting nodes from relational interval tree
`query_cleanup`	Deleting dynamic data structures
`query_dnf`	Building the first level of the DNF
`query_graph_match`	Graph match
`query_result`	Putting the episode in working memory
`query_sql_edge`	SQL query for an edge
`query_sql_end_ep`	SQL query for the end of the range of an edge
`query_sql_end_now`	SQL query for the end of the now of an edge
`query_sql_end_point`	SQL query for the end of the point of an edge
`query_sql_start_ep`	SQL query for the start of the range of an edge
`query_sql_start_now`	SQL query for the start of the now of an edge
`query_sql_start_point`	SQL query for the start of the point of an edge
`query_walk`	Walking the intervals
`query_walk_edge`	Expanding edges while walking the intervals
`query_walk_interval`	Updating satisfaction while walking the intervals

Visualization When debugging agents using episodic memory it is often useful to inspect the contents of individual episodes. Running `epmem --viz <episode id>` will output the contents of an episode in graphviz format. For more information on this format and visualization tools, see http://www.graphviz.org. The `epmem --print` option has the same syntax, but outputs text that is similar to using the `print` command to get the substructure of an identifier in working memory, which is possibly more useful for interactive debugging.

See Also

watch wma

explain-backtraces

Print information about chunk and justification backtraces.

Synopsis

```
explain-backtraces [options] [prod_name]
```

Default Aliases

eb	explain-backtraces

Options

prod_name	List all conditions and grounds for the chunk or justification.
-c, --condition	Explain why condition number n is in the chunk or justification.
-f, --full	Print the full backtrace for the named production

Description

This command provides some interpretation of backtraces generated during chunking.

The two most useful variants are:

```
explain-backtraces prodname
explain-backtraces -c n prodname
```

The first variant prints a numbered list of all the conditions for the named chunk or justification, and the *ground* which resulted in inclusion in the chunk/justification. A *ground* is a working memory element (WME) which was tested in the supergoal. Just knowing which WME was tested may be enough to explain why the chunk/justification exists. If not, the second variant, where n is the condition of interest, can be used to obtain a list of the productions which fired to obtain this condition in the chunk/justification (and the crucial WMEs tested along the way).

save-backtraces mode must be on when a chunk or justification is created or no explanation will be available. Calling explain-backtraces with no argument prints a list of all chunks and justifications for which backtracing information is available.

Use with no arguments to list all productions that can be ¨explained.¨

Examples

Examining the chunk `chunk-65*d13*tie*2` generated in a water-jug task:

```
soar> explain-backtraces chunk-65*d13*tie*2
 (sp chunk-65*d13*tie*2
   (state <s2> ^name water-jug ^jug <n4> ^jug <n3>)
   (state <s1> ^name water-jug ^desired <d1> ^operator <o1> + ^jug <n1>
          ^jug <n2>)
   (<s2> ^desired <d1>)
   (<o1> ^name pour ^into <n1> ^jug <n2>)
   (<n1> ^volume 3 ^contents 0)
   (<s1> ^problem-space <p1>)
   (<p1> ^name water-jug)
   (<n4> ^volume 3 ^contents 3)
   (<n3> ^volume 5 ^contents 0)
   (<n2> ^volume 5 ^contents 3)
-->
   (<s3> ^operator <o1> -))
 1 :  (state <s2> ^name water-jug)      Ground : (S3 ^name water-jug)
 2 :  (state <s1> ^name water-jug)      Ground : (S5 ^name water-jug)
 3 :  (<s1> ^desired <d1>)              Ground : (S5 ^desired D1)
 4 :  (<s2> ^desired <d1>)              Ground : (S3 ^desired D1)
 5 :  (<s1> ^operator <o1> +)           Ground : (S5 ^operator O18 +)
 6 :  (<o1> ^name pour)                 Ground : (O18 ^name pour)
 7 :  (<o1> ^into <n1>)                 Ground : (O18 ^into N3)
 8 :  (<n1> ^volume 3)                  Ground : (N3 ^volume 3)
 9 :  (<n1> ^contents 0)                Ground : (N3 ^contents 0)
10 :  (<s1> ^jug <n1>)                  Ground : (S5 ^jug N3)
11 :  (<s1> ^problem-space <p1>)        Ground : (S5 ^problem-space P3)
12 :  (<p1> ^name water-jug)            Ground : (P3 ^name water-jug)
13 :  (<s2> ^jug <n4>)                  Ground : (S3 ^jug N1)
14 :  (<n4> ^volume 3)                  Ground : (N1 ^volume 3)
15 :  (<n4> ^contents 3)                Ground : (N1 ^contents 3)
16 :  (<s2> ^jug <n3>)                  Ground : (S3 ^jug N2)
17 :  (<n3> ^volume 5)                  Ground : (N2 ^volume 5)
18 :  (<n3> ^contents 0)                Ground : (N2 ^contents 0)
19 :  (<s1> ^jug <n2>)                  Ground : (S5 ^jug N4)
20 :  (<n2> ^volume 5)                  Ground : (N4 ^volume 5)
21 :  (<n2> ^contents 3)                Ground : (N4 ^contents 3)
22 :  (<o1> ^jug <n2>)                  Ground : (O18 ^jug N4)
```

Further examining condition 21:

```
soar> explain-backtraces -c 21 chunk-65*d13*tie*2
Explanation of why condition  (N4 ^contents 3) was included in chunk-65*d13*tie*2
Production chunk-64*d13*opnochange*1 matched
    (N4 ^contents 3) which caused
production selection*select*failure-evaluation-becomes-reject-preference to match
    (E3 ^symbolic-value failure) which caused
```

```
A result to be generated.
```

See Also

save-backtraces

indifferent-selection

Controls indifferent preference arbitration.

Synopsis

```
indifferent-selection
indifferent-selection [-s]
indifferent-selection [-bgfxl]
indifferent-selection [-et] [value]
indifferent-selection [-p] parameter [reduction_policy]
indifferent-selection [-r] parameter reduction_policy [reduction_rate]
indifferent-selection [-a] [setting]
```

Default Aliases

inds	indifferent-selection

Options

-s, --stats	Summary of settings
policy	Set exploration policy
parameter [value]	Get/Set exploration policy parameters (if value not given, returns the current value)
parameter [reduction_policy]	Get/Set exploration policy parameter reduction policy (if policy not given, returns the current)
parameter reduction_policy [reduction_rate]	Get/Set exploration policy parameter reduction rate for a policy (if rate not give, returns the current)
-a, --auto-reduce [on,off]	Get/Set auto-reduction setting (if setting not provided, returns the current)

Description

The indifferent-selection command allows the user to set options relating to selection between operator proposals that are mutually indifferent in preference memory.

The primary option is the exploration policy (each is covered below). When Soar starts, *softmax* is the default policy.

Note: As of version 9.3.2, the architecture no longer automatically changes the policy to *epsilon-greedy* the first time Soar-RL is enabled.

Some policies have parameters to temper behavior. The indifferent-selection command provides basic facilities to automatically reduce these parameters exponentially and linearly each decision cycle by a fixed rate. In addition to setting these policies/rates, the *auto-reduce* option enables the automatic reduction system (disabled by default), for which the Soar decision cycle incurs a small performance cost.

Exploration Policies

`-b, --boltzmann`	Tempered softmax (uses temperature)
`-g, --epsilon-greedy`	Tempered greedy (uses epsilon)
`-x, --softmax`	Random, biased by numeric indifferent values (if a non-positive value is encountered, resorts to a uniform random selection)
`-f, --first`	Deterministic, first indifferent preference is selected
`-l, --last`	Deterministic, last indifferent preference is selected

Exploration Policy Parameters

Parameter Name	Acceptable Values	Default Value
`-e, --epsilon`	[0, 1]	0.1
`-t, --temperature`	(0, inf)	25

Exploration Policy Parameter Auto-Reduction Policies

Policy Name	Valid Rates	Default Rate
`exponential default`	[0, 1]	1
`linear`	[0, inf]	0

See Also

learn

Set the parameters for chunking.

Synopsis

```
learn [-l]
learn [-d|E|o]
learn [-eabnNpP]
```

Default Aliases

l	learn

Options

-e, --enable, --on	Turn chunking on. Can be modified by -a or -b.
-d, --disable, --off	Turn all chunking off. (default)
-E, --except	Learning is on, except as specified by RHS dont-learn actions.
-o, --only	Chunking is on only as specified by RHS force-learn actions.
-l, --list	Prints listings of dont-learn and force-learn states.
-a, --all-levels	Build chunks whenever a subgoal returns a result. Learning must be enabled.
-b, --bottom-up	Build chunks only for subgoals that have not yet had any subgoals with chunks built. Learning must be enabled.
-n, --local-negations	Build chunks when local negation encountered in backtraces. (default)
-N, --no-local-negations	Do not build chunks when local negation encountered in backtraces.
-p, --desirability-prefs	Add relevant desirability preferences to backtraces.
-P, --no-desirability-prefs	Do not add any desirability preferences to backtraces. (default)

Description

The learn command controls the parameters for chunking. With no arguments, this command prints out the current learning environment status. If arguments are provided, they will alter the learning environment as described in the options and arguments table. The watch command can be used to provide various levels of detail when productions are learned. Learning is *disabled* by default.

With the `--on` flag, chunking is on all the time. With the `--except` flag, chunking is on, but Soar will not create chunks for states that have had RHS `dont-learn` actions executed in them. With the `--only` flag, chunking is off, but Soar will create chunks for only those states that have had RHS `force-learn` actions executed in them. With the `--off` flag, chunking is off all the time.

The `--only` flag and its companion `force-learn` RHS action allow Soar developers to turn learning on in a particular problem space, so that they can focus on debugging the learning problems in that particular problem space without having to address the problems elsewhere in their programs at the same time. Similarly, the `--except` flag and its companion `dont-learn` RHS action allow developers to temporarily turn learning off for debugging purposes. These facilities are provided as debugging tools, and do not correspond to any theory of learning in Soar.

The following final six settings are orthogonal to the `--on`, `--except`, `--only`, and `--off` flags, and so, may be used in combination with them.

The `--all-levels` and `--bottom-up` control when chunks are formed when there are multiple levels of subgoals. With bottom-up learning, chunks are learned only in states in which no subgoal has yet generated a chunk. In this mode, chunks are learned only for the ¨bottom¨ of the subgoal hierarchy and not the intermediate levels. With experience, the subgoals at the bottom will be replaced by the chunks, allowing higher level subgoals to be chunked.

The options `--local-negations` and `--no-local-negations` control whether or not chunks can be created that are derived from rules that check for negated WMEs on the substate (local negations). Chunking through local negations can result in overgeneral chunks, but disabling this ability will reduce the number of chunks formed. The default is to enable chunking through local negations.

If chunking through local negations is disabled, to see when chunks are discarded (and why), set `watch --learning print` (see watch command).

The options `--desirability-prefs` and `--no-desirability-prefs` control whether or not desirability preferences are added to the context dependent preference set, which is the set of operator evaluation preferences that led to the selection of an operator in a subgoal. All preferences in the CDPS are backtraced through when creating justifications and chunks. When this option is disabled, only requirement preferences (requires and prohibits) will be added to the CDPS. When this option is enabled, relevant desirability prefs (better, best, worse, worst, indifferent) will also be added, producing more specific and possibly correct chunks. The default is to not include desirability preferences.

Learning can be turned on or off at any point during a run.

Examples

To enable learning only at the lowest subgoal level:

```
learn -e b
```

To see all the `force-learn` and `dont-learn` states registered by RHS actions

```
 learn -l
```

See Also

watch explain-backtraces save-backtraces

max-chunks

Limit the number of chunks created during a decision cycle.

Synopsis

```
 max-chunks [n]
```

Options

n	Maximum number of chunks allowed during a decision cycle.

Description

The max-chunks command is used to limit the maximum number of chunks that may be created during a decision cycle. The initial value of this variable is 50; allowable settings are any integer greater than 0.

The chunking process will end after max-chunks chunks have been created, *even if there are more results that have not been backtraced through to create chunks*, and Soar will proceed to the next phase. A warning message is printed to notify the user that the limit has been reached.

This limit is included in Soar to prevent getting stuck in an infinite loop during the chunking process. This could conceivably happen because newly-built chunks may match immediately and are fired immediately when this happens; this can in turn lead to additional chunks being formed, etc. If you see this warning, something is seriously wrong; Soar is unable to guarantee consistency of its internal structures. You should not continue execution of the Soar program in this situation; stop and determine whether your program needs to build more chunks or whether you've discovered a bug (in your program or in Soar itself).

max-dc-time

Set a wall-clock time limit such that the agent will be interrupted when a single decision cycle exceeds this limit.

Synopsis

```
max-dc-time [--seconds] [n]
max-dc-time -d
```

Options

n	Maximum decision cycle time in **microseconds**
-d, --disable	Disable this interrupt.
-s, --seconds	Interpret n as seconds (floating point OK)

Description

After output phase, the elapsed decision cycle time is checked to see if it is greater than the old maximum, and the maximum dc time stat is updated (see stats). At this time, this threshold is also checked. If met or exceeded, Soar stops at the end of the current output phase with an interrupted state.

Examples

```
max-dc-time -s 0.05
max-dc-time 4000
```

max-elaborations

Limit the maximum number of elaboration cycles in a given phase. Print a warning message if the limit is reached during a run.

Synopsis

```
max-elaborations [n]
```

Options

n	Maximum allowed elaboration cycles, must be a positive integer.

Description

This command sets and prints the maximum number of elaboration cycles allowed. If `n` is given, it must be a positive integer and is used to reset the number of allowed elaboration cycles. The default value is 100. max-elaborations with no arguments prints the current value.

max-elaborations controls the maximum number of elaborations allowed in a single decision cycle. The elaboration phase will end after max-elaboration cycles have completed, even if there are more productions eligible to fire or retract; and Soar will proceed to the next phase after a warning message is printed to notify the user. This limits the total number of cycles of parallel production firing but does not limit the total number of productions that can fire during elaboration.

This limit is included in Soar to prevent getting stuck in infinite loops (such as a production that repeatedly fires in one elaboration cycle and retracts in the next); if you see the warning message, it may be a signal that you have a bug your code. However some Soar programs are designed to require a large number of elaboration cycles, so rather than a bug, you may need to increase the value of max-elaborations.

max-elaborations is checked during both the Propose Phase and the Apply Phase. If Soar runs more than the max-elaborations limit in either of these phases, Soar proceeds to the next phase (either Decision or Output) even if quiescence has not been reached.

Examples

The command issued with no arguments, returns the max elaborations allowed:

```
max-elaborations
```

to set the maximum number of elaborations in one phase to 50:

```
max-elaborations 50
```

max-goal-depth

Limit the sub-state stack depth.

Synopsis

```
max-goal-depth [n]
```

Options

n	Maximum depth of sub-states allowed.

Description

The max-goal-depth command is used to limit the maximum depth of sub-states. The initial value of this variable is 100; allowable settings are any integer greater than 0.

This limit is included in Soar to prevent getting stuck in an infinite recursive loop, which may come about due to deliberate actions or via an agent bug, such as dropping inadvertently to state-no-change impasses.

max-memory-usage

Get the number of bytes that when exceeded by an agent, will trigger the memory usage exceeded event.

Synopsis

```
max-memory-usage [n]
```

Options

n	Size of limit in bytes.

Description

The max-memory-usage command is used to trigger the memory usage exceeded event. The initial value of this is 100MB (100,000,000); allowable settings are any integer greater than 0. The code supporting this event is not enabled by default because the test can be computationally expensive and is needed only for specific embedded applications. Users may enable the test and event generation by uncommenting code in `mem.cpp`.

Using the command with no arguments displays the current limit.

max-nil-output-cycles

Limit the maximum number of decision cycles that are executed without producing output when run is invoked with run-til-output args.

Synopsis

```
max-nil-output-cycles [n]
```

Options

n	Maximum number of consecutive output cycles allowed without producing output. Must be a positive integer.

Description

This command sets and prints the maximum number of nil output cycles (output cycles that put nothing on the output link) allowed when running using run-til-output (`run --output`). If n is not given, this command prints the current number of nil-output-cycles allowed. If n is given, it must be a positive integer and is used to reset the maximum number of allowed nil output cycles.

max-nil-output-cycles controls the maximum number of output cycles that generate no output allowed when a `run --out` command is issued. After this limit has been reached, Soar stops. The default initial setting of n is 15.

Examples

The command issued with no arguments, returns the max empty output cycles allowed:

```
 max-nil-output-cycles
```

to set the maximum number of empty output cycles in one phase to 25:

```
 max-nil-output-cycles 25
```

See Also

run

multi-attributes

Declare a symbol to be multi-attributed.

Synopsis

```
 multi-attributes [symbol [n]]
```

Options

symbol	Any Soar attribute.
n	Integer greater than 1, estimate of degree of simultaneous values for attribute.

Description

This command declares the given symbol to be an attribute which can take on multiple values. The optional `n` is an integer (greater than 1) indicating an upper limit on the number of expected values that will appear for an attribute. If `n` is not specified, the value 10 is used for each declared multi-attribute. More informed values will tend to result in greater efficiency.

This command is used only to provide hints to the production condition reorderer so it can produce better condition orderings. Better orderings enable the rete network to run faster. This command has no effect on the actual contents of working memory and most users needn't use this at all.

Note that multi-attributes declarations must be made before productions are loaded into soar or this command will have no effect.

Examples

Declare the symbol ¨thing¨ to be an attribute likely to take more than 1 but no more than 4 values:

```
 multi-attributes thing 4
```

numeric-indifferent-mode

Select method for combining numeric preferences.

Synopsis

```
 numeric-indifferent-mode [-as]
```

Options

`-a, --avg, --average`	Use average mode
`-s, --sum`	Use sum mode (default)

Description

The numeric-indifferent-mode command sets how multiple numeric indifferent preference values given to an operator are combined into a single value for use in random selection.

The default procedure is `--sum` which sums all numeric indifferent preference values given to the operator, defaulting to 0 if none exist. The alternative `--avg` mode will average the

values, also defaulting to 0 if none exist.

See Also

rl indifferent-selection

o-support-mode

Choose experimental variations of o-support.

Synopsis

```
 o-support-mode [n]
```

Options

3	Mode 3 is the same as mode 2 except that operator elaborations (adding attributes to operators) now get i-support even though you have to test the operator to elaborate an operator. In cases where the rule mixes support types, support defaults to o-support (and a warning is printed).
4	Mode 4 is the default. It is the same as mode 3 except where a rule mixes support types, support defaults to i-support (and a warning is still printed).

Description

The o-support-mode command is used to control the way that o-support is determined for preferences. Only o-support modes 3 & 4 are valid (other modes require Soar 7, which is no longer supported). O-support mode 4 should be considered an improved version of mode 3. The default o-support mode is mode 4.

In o-support modes 3 & 4, support is given production by production; that is, all preferences generated by the RHS of a single instantiated production will have the same support. The difference between the two modes is in how they handle productions with both operator and non-operator augmentations on the RHS. For more information on o-support calculations, see the relevant appendix in the Soar manual.

Running o-support-mode with no arguments prints out the current o-support-mode.

predict

Predict the next selected operator

Synopsis

```
 predict
```

Description

The predict command determines, based upon current operator proposals, which operator will be chosen during the next decision phase. If predict determines an operator tie will be encountered, ¨tie¨ is returned. If predict determines no operator will be selected (state no-change), ¨none¨ is returned. If predict determines a conflict will arise during the decision phase, ¨conflict¨ is returned. If predict determines a constraint failure will occur, ¨constraint¨ is returned. Otherwise, predict will return the id of the operator to be chosen. If operator selection will require probabilistic selection, and no alterations to the probabilities are made between the call to predict and decision phase, predict will manipulate the random number generator to enforce its prediction.

See Also

select

rl

Control how numeric indifferent preference values in RL rules are updated via reinforcement learning.

Synopsis

```
 rl -g|--get <parameter>
 rl -s|--set <parameter> <value>
 rl -t|--trace <parameter> <value>
 rl -S|--stats <statistic>
```

Options

`-g, --get`	Print current parameter setting
`-s, --set`	Set parameter value
`-t, --trace`	Print, clear, or init traces
`-S, --stats`	Print statistic summary or specific statistic

Description

The `rl` command sets parameters and displays information related to reinforcement learning.
The `print` and `watch` commands display additional RL related information not covered by
this command.

Parameters   Due to the large number of parameters, the `rl` command uses the `--get|--set`
`<parameter> <value>` convention rather than individual switches for each parameter. Run-
ning `rl` without any switches displays a summary of the parameter settings.

Parameter	Description	Possible values	Default
`chunk-stop`	If enabled, chunking does not create duplicate RL rules that differ only in numeric-indifferent preference value	`on, off`	`on`
`decay-mode`	How the learning rate changes over time	`normal,` `exponential,` `logarithmic,` `delta-bar-delta`	`normal`
`discount-rate`	Temporal discount (gamma)	[0, 1]	0.9
`eligibility-trace-decay-rate`	Eligibility trace decay factor (lambda)	[0, 1]	0
`eligibility-trace-tolerance`	Smallest eligibility trace value not considered 0	(0, inf)	0.001
`hrl-discount`	Discounting of RL updates over time in impassed states	`on, off`	`off`
`learning`	Reinforcement learning enabled	`on, off`	`off`
`learning-rate`	Learning rate (alpha)	[0, 1]	0.3
`learning-policy`	Value update policy	`sarsa,` `q-learning`	`sarsa`
`meta`	Store rule metadata in header string	`on, off`	`off`
`temporal-discount`	Discount RL updates over gaps	`on, off`	`on`
`temporal-extension`	Propagation of RL updates over gaps	`on, off`	`on`
`trace`	Update the trace	`on, off`	`off`
`update-log-path`	File to log information about RL rule updates	`""`, `<filename>`	`""`

Apoptosis Parameters

Parameter	Description	Possible values	Default
`apoptosis`	Automatic excising of productions via base-level decay	`none`, `chunks`, `rl-chunks`	`none`
`apoptosis-decay`	Base-level decay parameter	`[0, 1]`	0.5
`apoptosis-thresh`	Base-level threshold parameter (negates supplied value)	(0, inf)	2

Apoptosis is a process to automatically excise chunks via the base-level decay model (where rule firings are the activation events). A value of `chunks` has this apply to any chunk, whereas `rl-chunks` means only chunks that are also RL rules can be forgotten.

Statistics   Soar tracks some RL statistics over the lifetime of the agent. These can be accessed using `rl --stats <statistic>`. Running `rl --stats` without a statistic will list the values of all statistics.

`update-error`	Difference between target and current values in last RL update
`total-reward`	Total accumulated reward in the last update
`global-reward`	Total accumulated reward since agent initialization

Delta-Bar-Delta   This is an experimental feature of Soar RL. It based on the work in Richard S. Sutton's paper ¨Adapting Bias by Gradient Descent: An Incremental Version of Delta-Bar-Delta¨, available online at http://webdocs.cs.ualberta.ca/~sutton/papers/sutton-92a.pdf.

Delta Bar Delta (DBD) is implemented in Soar RL as a decay mode. It changes the way all the rules in the eligibility trace get their values updated. In order to implement this, the agent gets an additional learning parameter ¨meta-learning-rate¨ and each rule gets two additional decay parameters: beta and h. The meta learning rate is set manually; the per-rule features are handles automatically by the DBD algorithm. The key idea is that the meta parameters keep track of how much a rule's RL value has been updated recently, and if a rule gets updates in the same direction multiple times in a row then subsequent updates in the same direction will have more effect. So DBD acts sort of like momentum for the learning rate.

To enable DBD, use ¨rl –set decay-mode delta-bar-delta¨. To change the meta learning rate, use e.g. ¨rl –set meta-learning-rate 0.1¨. When you execute ¨rl¨, under the Experimental section you'll see the current settings for decay-mode and meta-learning-rate. Also, if a rule gets printed concisely (e.g. by executing ¨p¨), and the rule is an RL rule, and the decay mode is set to delta-bar-delta, then instead of printing the rule name followed by the update count and the RL value, it will print the rule name, beta, h, update count, and RL value.

Note that DBD is a different feature than ¨meta¨. Meta determines whether metadata about

a production is stored in its header string. If meta is on and DBD is on, then each rule's beta and h values will be stored in the header string in addition to the update count, so you can print out the rule, source it later and that metadata about the rule will still be in place.

RL Update Logging Sets a path to a file that Soar RL will write to whenever a production's RL value gets updated. This can be useful for logging these updates without having to capture all of Soar's output and parse it for these updates. Enable with e.g. ¨rl –set update-log-path rl_log.txt¨. Disable with <rl –set update-log-path ¨¨> - that is, use the empty string ¨¨ as the log path. The current log path appears under the experimental section when you execute ¨rl¨.

RL Trace If ¨rl –set trace on¨ has been called, then proposed operators will be recorded in the trace for all goal levels. Along with operator names and other attribute-value pairs, transition probabilities derived from their numeric preferences are recorded.

Legal arguments following ¨rl -t¨ or ¨rl –trace¨ are as follows:

print	Print the trace for the top state.
clear	Erase the traces for all goal levels.
init	Restart recording from the beginning of the traces for all goal levels.

These may be followed by an optional numeric argument specifying a specific goal level to print, clear, or init. ¨rl -t init¨ is called automatically whenever Soar is reinitialized. However, ¨rl -t clear¨ is never called automatically.

The format in which the trace is printed is designed to be used by the program dot, as part of the Graphviz suite. The command ¨ctf rl.dot rl -t¨ will print the trace for the top state to the file ¨rl.dot¨. (The default behavior for ¨rl -t¨ is to print the trace for the top state.)

Here are some sample dot invocations for the top state:

`dot -Tps rl.dot -o rl.ps`	`ps2pdf rl.ps`	Generate a .ps file and convert it to .pdf.
`dot -Tsvg rl.dot -o rl.svg`	`inkscape -f rl.svg -A rl.pdf`	Generate a .svg file and convert it to .pdf.

The .svg format works better for large traces.

See Also

excise print watch

save-backtraces

Save trace information to explain chunks and justifications.

Synopsis

```
save-backtraces [-ed]
```

Options

`-e, --enable, --on`	Turn explain sysparam on.
`-d, --disable, --off`	Turn explain sysparam off.

Description

The save-backtraces variable is a toggle that controls whether or not backtracing information (from chunks and justifications) is saved.

When save-backtraces is set to `off`, backtracing information is not saved and explanations of the chunks and justifications that are formed can not be retrieved. When save-backtraces is set to `on`, backtracing information can be retrieved by using the [cmd_explain_backtraces explain-backtraces] command. Saving backtracing information may slow down the execution of your Soar program, but it can be a very useful tool in understanding how chunks are formed.

See Also

explain-backtraces

select

Force the next selected operator

Synopsis

```
select id
```

Options

If no ¨id¨ is supplied, currently ¨select¨ed operator (if applicable) is displayed

`id`	case-insensitive operator id of the operator to be selected in the next decision phase

Description

The select command will force the selection of an operator, whose id is supplied as an argument, during the next decision phase. If the argument is not a proposed operator in the next decision phase, an error is raised and operator selection proceeds as if the select command had not been called. Otherwise, the supplied operator will be selected as the next operator, regardless of preferences. If select is called with no id argument, the command returns the operator id currently forced for selection (by a previous call to select), if one exists.

Examples

Assuming operator ¨O2¨ is a valid operator, this would select it as the next operator to be selected:

```
select O2
```

After this command, issuing just ¨select¨ will get ¨O2¨ as a return:

```
select
```

See Also

predict

set-stop-phase

Controls the phase where agents stop when running by decision.

Synopsis

```
set-stop-phase -[ABadiop]
```

Options

Options `-A` and `-B` are optional and mutually exclusive. If not specified, the default is `-B`.

Only one of `-a`, `-d`, `-i`, `-o`, `-p` must be selected.

With no options, reports the current stop phase.

-A, --after	Stop after specified phase.
-B, --before	Stop before specified phase (the default).
-a, --apply	Select the apply phase.
-d, --decision	Select the decision phase.
-i, --input	Select the input phase.
-o, --output	Select the output phase.
-p, --proposal	Select the proposal phase.

Description

When running by decision cycle it can be helpful to have agents stop at a particular point in its execution cycle. This command allows the user to control which phase Soar stops in. The precise definition is that ¨running for n decisions and stopping before phase ph means to run until the decision cycle counter has increased by n and then stop when the next phase is ph¨. The phase sequence (as of this writing) is: input, proposal, decision, apply, output. Stopping after one phase is exactly equivalent to stopping before the next phase.

On initialization Soar defaults to stopping before the input phase (or after the output phase, however you like to think of it).

Setting the stop phase applies to all agents.

Examples

```
set-stop-phase -Bi                 // stop before input phase
set-stop-phase -Ad                 // stop after decision phase (before apply phase)
set-stop-phase -d                  // stop before decision phase
set-stop-phase --after --output    // stop after output phase
set-stop-phase                     // reports the current stop phase
```

smem

Control the behavior of semantic memory.

Synopsis

```
smem
smem -e|--enable|--on
smem -d|--disable|--off
smem -i|--init
smem -g|--get <parameter>
smem -s|--set <parameter> <value>
smem -S|--stats [<statistic>]
```

```
smem -t|--timers [<timer>]
smem -a|--add <concept>
smem -p|--print [<lti>] [<depth>]
smem -v|--viz [<lti>] [<depth>]
smem -b|--backup <file name>
```

Options

`-e, --enable, --on`	Enable semantic memory.
`-d, --disable, --off`	Disable semantic memory.
`-g, --get`	Print current parameter setting
`-s, --set`	Set parameter value
`-i, --init`	Reinitialize ALL memories
`-S, --stats`	Print statistic summary or specific statistic
`-t, --timers`	Print timer summary or specific statistic
`-a, --add`	Add concepts to semantic memory
`-p, --print`	Print semantic store in user-readable format
`-v, --viz`	Print semantic store in graphviz format
`-b, --backup`	Creates a backup of the semantic database on disk

Description

The `smem` command changes the behavior of and displays information about semantic memory. The command `watch --smem` displays additional trace information for semantic memory not controlled by this command.

Parameters   Due to the large number of parameters, the `smem` command uses the `--get|--set <parameter> <value>` convention rather than individual switches for each parameter. Running `smem` without any switches displays a summary of the parameter settings.

Main Parameters

Parameter	Description	Possible values	Default
append-database	Controls whether database is over-written or appended when opening or re-initializing	on, off	off
database	Database storage method	file, memory	memory
learning	Semantic memory enabled	on, off	off
merge	Controls how re-trievals interact with long-term identifiers in working memory	none, add	add
mirroring	Controls automatic encoding of working-memory changes	on, off	off
path	Location of database file	*empty, some path*	*empty*

The `learning` parameter turns the episodic memory module on or off. When `learning` is set to `off`, no new episodes are encoded and no commands put on the epmem link are processed. This is the same as using the enable and disable commands.

The `path` parameter specifies the file system path the database is stored in. When `path` is empty, the database is stored in main memory and will be lost when the agent exits. When `path` is set to a valid file system path, the SQLite database is written to that path.

The append-database parameter will determine whether all existing facts stored in a database on disk will be erased when semantic memory loads it. Note that this affects semantic memory re-initialization also, i.e. if the append setting is off, all semantic facts stored to disk will be lost when an init-soar is performed.

Note that changes to database, path and append-database will not have an effect until the database is used after an initialization. This happens either at launch or when an init-soar or database initialization command is issued. To switch databases or database storage types on the fly, set your new parameters and then either perform an init-soar or `smem --init`.

The `smem --backup` command can be used to make a copy of the current state of the database, whether in memory or on disk. This command will commit all outstanding changes before initiating the copy.

For semantic memory to be reinitialized, all references to long-term identifiers in all of Soar's memories must be removed. Consequently, `smem --init` will reinitialize episodic, semantic, procedural, and working memories. It is equivalent to wiping the semantic store and executing these commands:

```
epmem --close
excise --all
```

```
init-soar
```

The `merge` parameter controls how the augmentations of retrieved long-term identifiers (LTIs) interact with an existing LTI in working memory. If the LTI is not in working memory or has no augmentations in working memory, this parameter has no effect. If the augmentation is in working memory and has augmentations, by default (`add`), semantic memory will add any augmentations that augmented the LTI in a retrieved LTI are added to working memory. If the parameter is set to `none` then semantic memory will not augment the LTI. Note this is opposite of the value of the same parameter in episodic memory.

The `mirroring` parameter controls a useful form of automatic encoding. If enabled (`on`), all changes to long-term identifiers (LTIs) in working memory are ¨mirrored¨ to semantic memory (assuming the LTI in working memory has at least one augmentation - i.e. no accidental clearing). The `mirrors` statistic is incremented for each LTI that is updated in this way.

Activation Parameters

Parameter	Description	Possible values	Default
`activation-mode`	Sets the ordering bias for retrievals that match more than one memory	`recency`, `frequency`, `base-level`	`recency`
`activate-on-query`	Determines if the results of queries should be activated	`on`, `off`	`on`
`base-decay`	Sets the decay parameter for base-level activation computation	> 0	0.5
`base-update-policy`	Sets the policy for re-computing base-level activation	`stable`, `naive`, `incremental`	`stable`
`base-incremental-threshes`	Sets time deltas after which base-level activation is re-computed for old memories	1, 2, 3, ...	10
`thresh`	Threshold for activation locality	0, 1, ...	100

If `activation-mode` is `base-level`, three parameters control bias values. The `base-decay` parameter sets the free decay parameter in the base-level model. Note that we do implement the (Petrov, 2006) approximation, with a history size set as a compile-time parameter (default=10). The `base-update-policy` sets the frequency with which activation is re-

computed. The default, `stable`, only recomputes activation when a memory is referenced (through storage or retrieval). The `naive` setting will update the entire candidate set of memories (defined as those that match the most constraining cue WME) during a retrieval, which has severe performance detriment and should be used for experimentation or those agents that require high-fidelity retrievals. The `incremental` policy updates a constant number of memories, those with last-access ages defined by the `base-incremental-threshes` set.

Performance Parameters

Parameter	Description	Possible values	Default
`cache-size`	Number of memory pages used in the SQLite cache	1, 2, ...	10000
`lazy-commit`	Delay writing semantic store changes to file until agent exits	`on`, `off`	`on`
`optimization`	Policy for committing data to disk	`safety`, `performance`	`performance`
`page-size`	Size of each memory page used in the SQLite cache	1k, 2k, 4k, 8k, 16k, 32k, 64k	8k
`timers`	Timer granularity	`off`, `one`, `two`, `three`	`off`

When the database is stored to disk, the `lazy-commit` and `optimization` parameters control how often cached database changes are written to disk. These parameters trade off safety in the case of a program crash with database performance. When `optimization` is set to `performance`, the agent will have an exclusive lock on the database, meaning it cannot be opened concurrently by another SQLite process such as SQLiteMan. The lock can be relinquished by setting the database to memory or another database and issuing init-soar/`smem --init` or by shutting down the Soar kernel.

Statistics   Semantic memory tracks statistics over the lifetime of the agent. These can be accessed using `smem --stats <statistic>`. Running `smem --stats` without a statistic will list the values of all statistics. Unlike timers, statistics will always be updated. Available statistics are:

Name	Label	Description
`act_updates`	Activation Updates	Number of times memory activation has been calculated
`db-lib-version`	SQLite Version	SQLite library version
`edges`	Edges	Number of edges in the semantic store
`mem-usage`	Memory Usage	Current SQLite memory usage in bytes
`mem-high`	Memory Highwater	High SQLite memory usage watermark in bytes
`mirrors`	Mirrors	Number of LTIs that have been automatically encoded via `mirroring`
`nodes`	Nodes	Number of nodes in the semantic store
`queries`	Queries	Number of times the **query** command has been issued
`retrieves`	Retrieves	Number of times the **retrieve** command has been issued
`stores`	Stores	Number of times the **store** command has been issued

Timers Semantic memory also has a set of internal timers that record the durations of certain operations. Because fine-grained timing can incur runtime costs, semantic memory timers are off by default. Timers of different levels of detail can be turned on by issuing `smem --set timers <level>`, where the levels can be `off`, `one`, `two`, or `three`, `three` being most detailed and resulting in all timers being turned on. Note that none of the semantic memory statistics nor timing information is reported by the `stats` command.

All timer values are reported in seconds.

Level one

`_total`	Total smem operations

Level two

`smem_api`	Agent command validation
`smem_hash`	Hashing symbols
`smem_init`	Semantic store initialization
`smem_ncb_retrieval`	Adding concepts (and children) to working memory
`smem_query`	Cue-based queries
`smem_storage`	Concept storage

Level three

three_activation	Recency information maintenance

Manual Storage   Concepts can be manually added to the semantic store using the `smem`
`--add <concept>` command. The format for specifying the concept is similar to that of
adding WMEs to working memory on the RHS of productions. For example:

```
smem --add {
    (<arithmetic> ^add10-facts <a01> <a02> <a03>)
    (<a01> ^digit1 1 ^digit-10 11)
    (<a02> ^digit1 2 ^digit-10 12)
    (<a03> ^digit1 3 ^digit-10 13)
}
```

Although not shown here, the common ¨dot-notation¨ format used in writing productions
can also be used for this command. Unlike agent storage, manual storage is automatically
recursive. Thus, the above example will add a new concept (represented by the temporary
¨arithmetic¨ variable) with three children. Each child will be its own concept with two
constant attribute/value pairs.

Visualization   When debugging agents using semantic memory it is often useful to in-
spect the contents of the semantic store. Running `smem --viz [<ltid>] [<depth>]` will
output the concept rooted at `<ltid>` to depth `<depth>` in graphviz format, including long-
term identifier activation levels. If `<ltid>` is omitted, the entire contents of the seman-
tic store are outputted. For more information on this format and visualization tools, see
http://www.graphviz.org. The `smem --print` option has the same syntax, but outputs text
that is similar to using the `print` command to get the substructure of an identifier in working
memory, which is possibly more useful for interactive debugging.

See Also

watch

timers

Toggle on or off the internal timers used to profile Soar.

Synopsis

```
timers [options]
```


Options

`-d, --disable, --off`	Disable all timers.
`-e, --enable, --on`	Enable timers as compiled.

Description

This command is used to control the timers that collect internal profiling information while Soar is running. With no arguments, this command prints out the current timer status. Timers are ENABLED by default. The default compilation flags for soar enable the basic timers and disable the detailed timers. The timers command can only enable or disable timers that have already been enabled with compiler directives. See the stats command for more info on the Soar timing system.

See Also

stats

waitsnc

Generate a wait state rather than a state-no-change impasse.

Synopsis

```
wait -[e|d]
```

Options

`-e, --enable, --on`	Turns a state-no-change into a `wait` state.
`-d, --disable, --off`	Default. A state-no-change generates an impasse.

Description

In some systems, espcially those that model expert (fully chunked) knowledge, a state-no-change may represent a *wait state* rather than an impasse. The waitsnc command allows the user to switch to a mode where a state-no-change that would normally generate an impasse (and subgoaling), instead generates a *wait* state. At a *wait* state, the decision cycle will repeat (and the decision cycle count is incremented) but no *state-no-change* impasse (and therefore no substate) will be generated.

When issued with no arguments, waitsnc returns its current setting.

wma

Control the behavior of working memory activation.

Synopsis

```
wma
wma -g|--get <parameter>
wma -s|--set <parameter> <value>
wma -S|--stats [<statistic>]
wma -t|--timers [<timer>]
wma -h|--history <timetag>
```

Options

-g, --get	Print current parameter setting
-s, --set	Set parameter value
-S, --stats	Print statistic summary or specific statistic
-t, --timers	Print timer summary or specific timer
-h, --history	Print reference history of a WME

Description

The `wma` command changes the behavior of and displays information about working memory activation.

To get the activation of individual WMEs, use `print -i`. To get the reference history of an individual WME, use `wma -h|--history <timetag>`. For example:

```
print --internal s1
(4000016: S1 ^ct 1000000 [3.6])
(4: S1 ^epmem E1 [1])
(11: S1 ^io I1 [1])
(20: S1 ^max 1000000 [3.4])
(18: S1 ^name ct [3.4])
(4000018: S1 ^operator O1000001 [1] +)
(4000019: S1 ^operator O1000001 [1])
(3: S1 ^reward-link R1 [1])
(8: S1 ^smem S2 [1])
(2: S1 ^superstate nil [1])
(14: S1 ^top-state S1 [1])
(1: S1 ^type state [1])
```

The bracketed values are activation. To get the history of an individual element:

```
wma --history 18
history (60/5999999, first @ d1):
  6 @ d1000000 (-1)
  6 @ d999999 (-2)
  6 @ d999998 (-3)
```

```
6 @ d999997 (-4)
6 @ d999996 (-5)
6 @ d999995 (-6)
6 @ d999994 (-7)
6 @ d999993 (-8)
6 @ d999992 (-9)
6 @ d999991 (-10)
```

```
considering WME for decay @ d1019615
```

This shows the last 60 references (of 5999999 in total, where the first occurred at decision cycle 1). For each reference, it says how many references occurred in the cycle (such as 6 at decision 1000000, which was one cycle ago at the time of executing this command). Note that references during the current cycle will not be reflected in this command (or computed activation value) until the end of output phase. If `forgetting` is `on`, this command will also display the cycle during which the WME will be considered for decay. Even if the WME is not referenced until then, this is not necessarily the cycle at which the WME will be forgotten. However, it is guaranteed that the WME will not be forgotten before this cycle.

Parameters   The `wma` command uses the `--get|--set <parameter> <value>` convention rather than individual switches for each parameter. Running `wma` without any switches displays a summary of the parameter settings.

Parameter	Description	Possible values	Default
`activation`	Enable working memory activation	`on`, `off`	`off`
`decay-rate`	WME decay factor	[0, 1]	0.5
`decay-thresh`	Forgetting threshold	(0, inf)	2.0
`forgetting`	Enable removal of WMEs with low activation values	`on`, `off`	`off`
`forget-wme`	If `lti` only remove WMEs with a long-term id	`all`, `lti`	`all`
`max-pow-cache`	Maximum size, in MB, for the internal `pow` cache	1, 2, ...	10
`petrov-approx`	Enables the (Petrov 2006) long-tail approximation	`on`, `off`	`off`
`timers`	Timer granularity	`off`, `one`	`off`

The `decay-rate` and `decay-thresh` parameters are entered as positive decimals, but are internally converted to, and printed out as, negative.

The `petrov-approx` may provide additional validity to the activation value, but comes at a significant computational cost, as the model includes unbounded positive exponential

computations, which cannot be reasonably cached.

When `activation` is enabled, the system produces a cache of results of calls to the `pow` function, as these can be expensive during runtime. The size of the cache is based upon three run-time parameters (`decay-rate`, `decay-thresh`, and `max-pow-cache`), and one compile time parameter, `WMA_REFERENCES_PER_DECISION` (default value of 50), which estimates the maximum number of times a WME will be referenced during a decision. The cache is composed of `double` variables (i.e. 64-bits, currently) and the number of cache items is computed as follows:

$e^\wedge((\text{decay\_thresh} - \ln(\text{max\_refs})) / \text{decay\_rate})$

With the current default parameter values, this will incur about 1.04MB of memory. Holding the `decay-rate` constant, reasonable changes to `decay-thresh` (i.e. +/- 5) does not greatly change this value. However, small changes to `decay-rate` will dramatically change this profile. For instance, keeping everything else constant, a `decay-thresh` of 0.3 requires ~2.7GB and 0.2 requires ~50TB. Thus, the `max-pow-cache` parameter serves to allow you to control the space vs. time tradeoff by capping the maximum amount of memory used by this cache. If `max-pow-cache` is much smaller than the result of the equation above, you may experience somewhat degraded performance due to relatively frequent system calls to `pow`.

If `forget-wme` is `lti` and `forgetting` is `on`, only those WMEs whose id is a long-term identifier **at the decision of forgetting** will be removed from working memory. If, for instance, the id is stored to semantic memory after the decision of forgetting, the WME will not be removed till some time after the next WME reference (such as testing/creation by a rule).

Statistics   Working memory activation tracks statistics over the lifetime of the agent. These can be accessed using `wma --stats <statistic>`. Running `wma --stats` without a statistic will list the values of all statistics. Unlike timers, statistics will always be updated. Available statistics are:

Name	Label	Description
`forgotten-wmes`	Forgotten WMEs	Number of WMEs removed from working memory due to forgetting

Timers   Working memory activation also has a set of internal timers that record the durations of certain operations. Because fine-grained timing can incur runtime costs, working memory activation timers are off by default. Timers of different levels of detail can be turned on by issuing `wma --set timers <level>`, where the levels can be `off` or `one`, `one` being most detailed and resulting in all timers being turned on. Note that none of the working memory activation statistics nor timing information is reported by the `stats` command.

All timer values are reported in seconds.

Level one

`wma_forgetting`	Time to process forgetting operations each cycle
`wma_history`	Time to consolidate reference histories each cycle

See Also

print

8.5   File System I/O Commands

This section describes commands which interact in one way or another with operating system input and output, or file I/O. Users can save/retrieve information to/from files, redirect the information printed by Soar as it runs, and save and load the binary representation of productions. The specific commands described in this section are:

Summary

> **cd** - Change directory.
>
> **clog** - Record all user-interface input and output to a file. *(was `log`)*
>
> **command-to-file** - Dump the printed output and results of a command to a file.
>
> **dirs** - List the directory stack.
>
> **echo** - Print a string to the current output device.
>
> **ls** - List the contents of the current working directory.
>
> **popd** - Pop the current working directory off the stack and change to the next directory on the stack.
>
> **pushd** - Push a directory onto the directory stack, changing to it.
>
> **pwd** - Print the current working directory.
>
> **rete-net** - Save the current Rete net, or restore a previous one.
>
> **set-library-location** - Set the top level directory containing demos/help/etc.
>
> **source** - Load and evaluate the contents of a file.

The **source** command is used for nearly every Soar program. The directory functions are important to understand so that users can navigate directories/folders to load/save the files of interest. Soar applications that include a graphical interface or other simulation environment will often require the use of **echo** .

cd

Change directory.

Synopsis

```
cd [directory]
```

Default Aliases

chdir	cd

Options

directory	The directory to change to, can be relative or full path.

Description

Change the current working directory. If run with no arguments, returns to the directory that the command line interface was started in, often referred to as the *home* directory.

Examples

To move to the relative directory named `../home/soar/agents`:

```
cd ../home/soar/agents
```

See Also

dirs ls pushd popd source pwd

clog

Record all user-interface input and output to a file.

Synopsis

```
clog -[Ae] filename
clog -a string
clog [-cdoq]
```

Options

`filename`	Open filename and begin logging.
`-c, --close, -o, --off, -d, --disable`	Stop logging, close the file.
`-a, --add string`	Add the given string to the open log file.
`-q, --query`	Returns open if logging is active or closed if logging is not active.
`-A, --append, -e, --existing`	Opens existing log file named `filename` and logging is added at the end of the file.

Description

The `clog` command allows users to save all user-interface input and output to a file. When Soar is logging to a file, everything typed by the user and everything printed by Soar is written to the file (in addition to the screen).

Invoke `clog` with no arguments (or with `-q`) to query the current logging status. Pass a filename to start logging to that file (relative to the command line interface's home directory). Use the `close` option to stop logging.

Examples

To initiate logging and place the record in foo.log:

```
clog foo.log
```

To append log data to an existing foo.log file:

```
clog -A foo.log
```

To terminate logging and close the open log file:

```
clog -c
```

Known Issues

Does not log everything when structured output is selected.

See also

command-to-file

command-to-file

Dump the printed output and results of a command to a file.

Synopsis

```
command-to-file [-a] filename command [args]
```

Options

-a, --append	Append if file exists.
filename	The file to log the results of the command to
command	The command to log
args	Arguments for command

Description

This command logs a single command. It is almost equivalent to opening a log using clog, running the command, then closing the log, the only difference is that input isn't recorded.

Running this command while a log is open is an error. There is currently not support for multiple logs in the command line interface, and this would be an instance of multiple logs.

This command echos output both to the screen and to a file, just like clog.

See also

clog

dirs

List the directory stack.

Synopsis

```
dirs
```

Description

This command lists the directory stack. Agents can move through a directory structure by pushing and popping directory names. The dirs command returns the stack.

The command pushd places a new ¨agent current directory¨ on top of the directory stack and changes to it. The command popd removes the directory at the top of the directory stack and changes to the previous directory which now appears at the top of the stack.

See Also

cd ls pushd popd source pwd

echo

Print a string to the current output device.

Synopsis

```
echo [--nonewline] [string]
```

Options

string	The string to print.
-n, --nonewline	Supress printing of the newline character

Description

This command echos the args to the current output stream. This is normally stdout but can be set to a variety of channels. If an arg is `--nonewline` then no newline is printed at the end of the printed strings. Otherwise a newline is printed after printing all the given args. Echo is the easiest way to add user comments or identification strings in a log file.

Examples

This example will add these comments to the screen and any open log file.

```
echo This is the first run with disks = 12
```

See Also

clog

ls

List the contents of the current working directory.

Synopsis

```
 ls
```

Default Aliases

dir	ls

Description

List the contents of the working directory.

See Also

cd dirs pushd popd source pwd

popd

Pop the current working directory off the stack and change to the next directory on the stack. Can be relative pathname or fully specified path.

Synopsis

```
 popd
```

== Description =

This command pops a directory off of the directory stack and [cmd_cd changes to it]. See the [cmd_dirs dirs] command for an explanation of the directory stack.

See Also

cd dirs ls pushd source pwd

pushd

Push a directory onto the directory stack, changing to it.

Synopsis

```
pushd directory
```

Description

Maintain a stack of working directories and push the directory on to the stack. Can be relative path name or fully specified.

See Also

cd dirs ls popd source pwd

pwd

Print the current working directory.

Synopsis

```
pwd
```

Default Aliases

topd	pwd

Description

Prints the current working directory of Soar.

rete-net

Save the current Rete net, or restore a previous one.

Synopsis

```
rete-net -s|l filename
```

Default Aliases

rn	rete-net

Options

-s, --save	Save the Rete net in the named file. Cannot be saved when there are justifications present. Use excise -j
-l, -r, --load, --restore	Load the named file into the Rete network. working memory and production memory must both be empty. Use excise
filename	The name of the file to save or load.

Description

The rete-net command saves the current Rete net to a file or restores a Rete net previously saved. The Rete net is Soar's internal representation of production and working memory; the conditions of productions are reordered and common substructures are shared across different productions. This command provides a fast method of saving and loading productions since a special format is used and no parsing is necessary. Rete-net files are portable across platforms that support Soar.

Normally users wish to save only production memory. Note that justifications cannot be present when saving the Rete net. Issuing an init-soar before saving a Rete net will remove all justifications and working memory elements.

If the filename contains a suffix of `.Z`, then the file is compressed automatically when it is saved and uncompressed when it is loaded. Compressed files may not be portable to another platform if that platform does not support the same uncompress utility.

See Also

excise init-soar

set-library-location

Set the top level directory containing demos/help/etc.

Synopsis

```
set-library-location [directory]
```

Options

`directory`	The new desired library location.

Description

Invoke with no arguments to query what the current library location is. The library location should contain at least the help/ subdirectory and the command-names file for help to work.

See Also

help

source

Load and evaluate the contents of a file.

Synopsis

```
source [options] filename
```

Options

`filename`	The file of Soar productions and commands to load.
`-a, --all`	Enable a summary for each file sourced
`-d, --disable`	Disable all summaries
`-v, --verbose`	Print excised production names

Description

Load and evaluate the contents of a file. The `filename` can be a relative path or a fully qualified path. source will generate an implicit push to the new directory, execute the command, and then pop back to the current working directory from which the command was issued.

After the source completes, the number of productions sourced and excised is printed:

```
agent> source demos/mac/mac.soar
*****************
Total: 18 productions sourced.
Source finished.
agent> source demos/mac/mac.soar
```

```
#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*
Total: 18 productions sourced. 18 productions excised.
Source finished.
```

This can be disabled by using the **-d** flag:

```
agent> source demos/mac/mac.soar -d
*****************
Source finished.
agent> source demos/mac/mac.soar -d
#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*
Source finished.
```

A list of excised productions is available using the **-v** flag:

```
agent> source demos/mac/mac.soar -v
#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*
Total: 18 productions sourced. 18 productions excised.
Excised productions:
        mac*detect*state*success
        mac*evaluate*state*failure*more*cannibals
        monitor*move-boat
        monitor*state*left
 ...
```

A separate summary for each file sourced can be enabled using the **-a** flag:

```
agent> source demos/mac/mac.soar -a
_firstload.soar: 0 productions sourced.
all_source.soar: 0 productions sourced.
**
goal-test.soar: 2 productions sourced.
***
monitor.soar: 3 productions sourced.
****
search-control.soar: 4 productions sourced.
top-state.soar: 0 productions sourced.
elaborations_source.soar: 0 productions sourced.
_readme.soar: 0 productions sourced.
**
initialize-mac.soar: 2 productions sourced.
*******
move-boat.soar: 7 productions sourced.
mac_source.soar: 0 productions sourced.
mac.soar: 0 productions sourced.
Total: 18 productions sourced.
Source finished.
```

Combining the **-a** and **-v** flags add excised production names to the output for each file.

See Also

8.6   Soar I/O Commands

This section describes the commands used to manage Soar's Input/Output (I/O) system, which provides a mechanism for allowing Soar to interact with external systems, such as a computer game environment or a robot.

Soar I/O functions make calls to `add-wme` and `remove-wme` to add and remove elements to the **io** structure of Soar's working memory.

The specific commands described in this section are:

Summary

> **add-wme** - Manually add an element to working memory.
>
> **capture-input** - Store the input wmes in a file for reloading later.
>
> **remove-wme** - Manually remove an element from working memory.
>
> **replay-input** - Load input wmes for each decision cycle from a file.

These commands are used mainly when Soar needs to interact with an external environment. Users might take advantage of these commands when debugging agents, but care should be used in adding and removing wmes this way as they do not fall under Soar's truth maintenance system.

add-wme

Manually add an element to working memory.

Synopsis

```
 add-wme id [^]attribute value [+]
```

Default Aliases

aw	add-wme

Options

id	Must be an existing identifier.
^	Leading $^\wedge$ on attribute is optional.
attribute	Attribute can be any Soar symbol. Use * to have Soar create a new identifier.
value	Value can be any soar symbol. Use * to have Soar create a new identifier.
+	If the optional preference is specified, its value must be + (acceptable).

Description

Manually add an element to working memory. add-wme is often used by an input function to update Soar's information about the state of the external world.

add-wme adds a new wme with the given id, attribute, value and optional preference. The given id must be an existing identifier. The attribute and value fields can be any Soar symbol. If * is given in the attribute or value field, Soar creates a new identifier (symbol) for that field. If the preference is given, it can only have the value + to indicate that an acceptable preference should be created for this wme.

Note that because the id must already exist in working memory, the WME that you are adding will be attached (directly or indirectly) to the top-level state. As with other WME's, any WME added via a call to add-wme will automatically be removed from working memory once it is no longer attached to the top-level state.

Examples

This example adds the attribute/value pair $^\wedge$`message-status received` to the identifier (symbol) S1:

```
 add-wme S1 ^message-status received
```

This example adds an attribute/value pair with an acceptable preference to the identifier (symbol) Z2. The attribute is `message` and the value is a unique identifier generated by Soar. Note that since the $^\wedge$ is optional, it has been left off in this case.

```
 add-wme Z2 message * +
```

Warnings

Be careful how you use this command. It may have weird side effects (possibly even including system crashes). For example, the chunker can't backtrace through wmes created via add-wme nor will such wmes ever be removed thru Soar's garbage collection. Manually removing context/impasse wmes may have unexpected side effects.

See Also

remove-wme

capture-input

Store the input wmes in a file for reloading later.

Synopsis

```
capture-input --open filename [--flush]
capture-input [--query]
capture-input --close
```

Options

filename	Open filename and begin recording input.
-o, --open	Writes captured input to file overwriting any existing data.
-f, --flush	Writes input to file as soon as it is encountered instead of storing it in RAM and writing when capturing is turned off.
-q, --query	Returns open if input capturing is active or closed if capturing is not active.
-c, --close	Stop capturing input and close the file, writing captured input unless the flush option is given.

Description

Save agent commands issued from the input cycle function in a file for reloading later.

This command may be used to start and stop the recording of input function commands as created by an external simulation. Commands are recorded decision cycle by decision cycle. Use the command replay-input to replay the sequence.

Note that this command seeds the random number generator and writes the seed to the capture file.

See Also

replay-input

remove-wme

Manually remove an element from working memory.

Synopsis

```
remove-wme timetag
```

Default Aliases

rw	remove-wme

Options

timetag	A positive integer matching the timetag of an existing working memory element.

Description

The remove-wme command removes the working memory element with the given timetag. This command is provided primarily for use in Soar input functions; although there is no programming enforcement, remove-wme should only be called from registered input functions to delete working memory elements on Soar's input link.

Beware of weird side effects, including system crashes.

Warnings

remove-wme should never be called from the RHS: if you try to match a wme on the LHS of a production, and then remove the matched wme on the RHS, Soar will crash.

If used other than by input and output functions interfaced with Soar, this command may have weird side effects (possibly even including system crashes). Removing input wmes or context/impasse wmes may have unexpected side effects. You've been warned.

See Also

add-wme

replay-input

Load input wmes for each decision cycle from a file.

Synopsis

```
replay-input --open filename
replay-input [--query]
replay-input --close
```

Options

`filename`	Open filename and load input and random seed.
`-o, --open`	Reads captured input from file in to memory and seeds the random number generator.
`-q, --query`	Returns open if input replay is active or closed if not active.
`-c, --close`	Stop replaying input.

Description

Replays input stored using the capture-input command. The replay file also includes a random number generator seed and seeds the generator with that.

See Also

capture-input

8.7 Miscellaneous

The specific commands described in this section are:

Summary

alias - Define a new alias, or command, using existing commands and arguments.

allocate - Allocate additional 32 kilobyte blocks of memory for a specified memory pool without running Soar.

cli - Load and manipulate command line module add-ons.

echo-commands - Set whether or not commands are echoed to other connected debuggers.

edit-production - Fire event to Move focus in an open editor to this production.

load-library - Load a shared library into the local client

port - Returns the port the kernel instance is listening on.

rand - Generate a random number.

soarnews - Prints information about the current release.

srand - Seed the random number generator.

time - Uses a default system clock timer to record the wall time required while executing a command.

unalias - Remove an existing alias.

version - Returns version number of Soar kernel.

alias

Define a new alias, or command, using existing commands and arguments.

Synopsis

```
alias [name [cmd [args]]]
```

Default Aliases

a	alias

Description

This command defines new aliases by creating Soar procedures with the given name. The new procedure can then take an arbitrary number of arguments which are post-pended to the given definition and then that entire string is executed as a command. The definition must be a single command, multiple commands are not allowed. The alias procedure checks to see if the name already exists, and does not destroy existing procedures or aliases by the same name. Existing aliases can be removed by using the unalias command. With no arguments, alias returns the list of defined aliases. With only the name given, alias returns the current definition.

Examples

The alias `wmes` is defined as:

```
alias wmes print -i
```

If the user executes a command such as:

```
wmes {(* ^superstate nil)}
```

... it is as if the user had typed this command:

```
print -i {(* ^superstate nil)}
```

To check what a specific alias is defined as, you would type

```
 alias wmes
```

See Also

unalias

allocate

Allocate additional 32 kilobyte blocks of memory for a specified memory pool without running Soar.

Synopsis

```
 allocate [pool blocks]
```

Description

Soar allocates blocks of memory for its memory pools as it is needed during a run (or during other actions like loading productions). Unfortunately, this behavior translates to an increased run time for the first run of a memory-intensive agent. To mitigate this, blocks can be allocated before a run by using this command.

Issuing the command with no parameters lists current pool usage, exactly like stats command's memory flag.

Issuing the command with part of a pool's name and a positive integer will allocate that many additional blocks for the specified pool. Only the first few letters of the pool's name are necessary. If more than one pool starts with the given letters, which pool will be chosen is unspecified.

Memory pool block size in this context is approximately 32 kilobytes, the exact size determined during agent initialization.

See Also

stats

cli

Manipulate external command line interface modules.

Synopsis

```
cli <module-name> on|off
```

Options

`on`	Turn a command-line module on.
`off`	Turn a command-line module on

Description

The `cli` command is used to interface to special, external libraries that extend the function-ality of the Soar command line interface. For example, a module may provide the ability to run Tcl or Python code from any Soar command line.

The first time that you turn a module on, Soar will attempt to look for a dynamically-linked library based on the module-name passed to the cli command. Specifically, it looks for a dll named

`[dllprefix][language]SoarLib.[dllextension]`

(The dll prefix and extension vary depending on the platform.)

When the `on` parameter is given, all future Soar commands will be passed to the module for processing. This allows the module to provide additional functionality.

Similarly, the off message will send the CLI library an `off` message that will cause it to unregister all relevant callback handlers and stop processing Soar commands.

Examples

To turn on Tcl mode:

```
cli tcl on
```

To turn off Tcl mode:

```
cli tcl off
```

echo-commands

Set whether or not commands are echoed to other connected debuggers.

Synopsis

```
echo-commands [-yn]
```

Options

`-n, --no`	Do not echo commands.
`-y, --yes`	Do echo commands.

Description

Setting this on will echo typed commands to other connected debuggers. Otherwise, the output is displayed without the initiating command, and this can be confusing.

edit-production

Move focus in an editor to this production.

Synopsis

```
edit-production production_name
```

Options

`production_name`	The name of the production to edit.

Description

If an editor (currently limited to Visual Soar) is open and connected to Soar, this command causes the editor to open the file containing this production and move the cursor to the start of the production. If there is no editor connected to Soar, the command does nothing. In order to connect Visual Soar to Soar, launch Visual Soar and choose Connect from the Soar Runtime menu. Then open the Visual Soar project that you're working on. At that point, you're set up and edit-production will start to work.

Examples

```
edit-production my*production*name
```

See Also

load-library

Load a shared library into the local client (for the purpose of, e.g., providing custom event handling).

Synopsis

```
load-library [library_name] [arguments]
```

Options

library_name	The root name of the library (without the .dll or .so extension; this is added for you depending on your platform).
arguments	Whatever arguments the library's initialization function is expecting, if any.

Description

Sometimes, a user will want to extend an existing environment. For example, the person may want to provide custom RHS functions, or register for print events for the purpose of logging trace information. If modifying the existing environment is cumbersome or impossible, then the user has two options: create a remote client that provides the functionality, or use load-library. Load-library creates extensions in the local client, making it orders of magnitude faster than a remote client.

To create a loadable library, the library must contain the following function:

```
#ifdef __cplusplus
extern "C" {
#endif

    EXPORT char* sml_InitLibrary(Kernel* pKernel, int argc, char** argv) {
        // Your code here
    }

#ifdef __cplusplus
} // extern "C"
#endif
```

This function is called when load-library loads your library. It is responsible for any initialization that you want to take place (e.g. registering custom RHS functions, registering for events, etc).

The argc and argv arguments are intended to mirror the arguments that a standard SML client would get. Thus, the first argument is the name of the library, and the rest are whatever other arguments are provided. This is to make it easy to use the same codebase to create a loadable library or a standard remote SML client (e.g. when run as a standard client, just pass the arguments main gets into `sml_InitLibrary`).

The return value of `sml_InitLibrary` is for any error messages you want to return to the load-library call. If no error occurs, return a zero-length string.

An example library is provided in the `Tools/TestExternalLibraryLib` project. This example can also be compiled as a standard remote SML client. The `Tools/TestExternalLibraryExe` project tests loading the `TestExternalLibraryLib` library.

Examples

To load `TestExternalLibraryLib`:

```
load-library TestExternalLibraryLib
```

To load a library that takes arguments (say, a logger):

```
load-library my-logger -filename mylog.log
```

port

Returns the port the kernel instance is listening on.

Synopsis

```
port
```

rand

Generate a random number.

Synopsis

```
rand                  # returns a real number in [0,1] (calls SoarRand())
rand n                # returns a real number in [0,n] (calls SoarRand(max))
rand --integer        # returns an integer in [0,2^32-1] (calls SoarRandInt)
```

```
rand --integer n     # returns an integer in [0,n], n<2^32 (calls SoarRandInt(max))
```

Options

-i, --integer	Return an integer, optional argument is upper-bound (inclusive)

Description

Generates a random non-negative number, returning the result in a string.

Examples

```
 rand --integer 10 # returns 0..10, for example: "4"
```

srand

Seed the random number generator.

Synopsis

```
 srand [seed]
```

Options

seed	Random number generator seed.

Description

Seeds the random number generator with the passed seed. Calling srand without providing a seed will seed the generator based on the contents of /dev/urandom (if available) or else based on time() and clock() values.

Examples

```
 srand 0
```

soarnews

Prints information about the current release.

Synopsis

```
soarnews
```

Default Aliases

sn	soarnews

time

Use a default system clock timer to record the wall time required while executing a command.

Synopsis

```
time command [arguments]
```

Options

command	The command to execute.
arguments	Optional command arguments.

unalias

Undefine an existing alias

Synopsis

```
unalias name
```

Default Aliases

un	[cmd_unalias unalias]

Description

This command undefines a previously created alias. This command takes exactly one argument: the name of the alias to remove. Use the alias command by itself to list all defined aliases.

Examples

```
 unalias varprint
```

See Also

alias

version

Returns the version number of the Soar kernel.

Synopsis

```
 version
```

Description

This command gives version information about the current Soar kernel. It returns the version
number and build date which can then be stored by the agent or the application.

Appendix A

The Blocks-World Program

```
###############################################################################
###
### File             : blocks.soar
### Original author(s): John E. Laird <laird@eecs.umich.edu>
### Organization      : University of Michigan AI Lab
### Created on        : 15 Mar 1995, 13:53:46
### Last Modified By  : Clare Bates Congdon <congdon@eecs.umich.edu>
### Last Modified On  : 17 Jul 1996, 16:35:14
### Soar Version      : 7
###
### Description : A new, simpler implementation of the blocks world
###               with just three blocks being moved at random.
###
### Notes:
###    CBC, 6/27: Converted to Tcl syntax
###    CBC, 6/27: Added extensive comments
###############################################################################


###############################################################################
# Create the initial state with blocks A, B, and C on the table.
#
# This is the first production that will fire; Soar creates the initial state
#   as an architectural function (in the 'zeroth' decision cycle), which will
#   match against this production.
# This production does a lot of work because it is creating (preferences for)
# all the structure for the initial state:
# 1. The state has a problem-space named 'blocks'. The problem-space limits
#    the operators that will be selected for a task. In this simple problem,
#    it isn't really necessary (there is only one operator), but it's a
#    programming convention that you should get used to.
# 2. The state has four 'things' -- three blocks and the table.
# 3. The state has three 'ontop' relations
# 4. Each of the things has substructure: their type and their names. Note that
#    the fourth thing is actually a 'table'.
# 5. Each of the ontop relations has substructure: the top thing and the
#    bottom thing.
# Finally, the production writes a message for the user.
```

217

```
#
# Note that this production will fire exactly once and will never retract.

sp {blocks-world*elaborate*initial-state
   (state <s> ^superstate nil)
-->
   (<s> ^problem-space blocks
        ^thing <block-A> <block-B> <block-C> <table>
        ^ontop <ontop-A> <ontop-B> <ontop-C>)
   (<block-A> ^type block ^name A)
   (<block-B> ^type block ^name B)
   (<block-C> ^type block ^name C)
   (<table> ^type table ^name TABLE)
   (<ontop-A> ^top-block <block-A> ^bottom-block <table>)
   (<ontop-B> ^top-block <block-B> ^bottom-block <table>)
   (<ontop-C> ^top-block <block-C> ^bottom-block <table>)
   (write (crlf) |Initial state has A, B, and C on the table.|)}


##############################################################################
# State elaborations - keep track of which objects are clear
# There are two productions - one for blocks and one for the table.
##############################################################################

##############################################################################
# Assert table always clear
#
# The conditions establish that:
#  1. The state has a problem-space named 'blocks'.
#  2. The state has a thing of type table.
# The action:
#  1. creates an acceptable preference for an attribute-value pair asserting
#     the table is clear.
#
# This production will also fire once and never retract.

sp {elaborate*table*clear
   (state <s> ^problem-space blocks
              ^thing <table>)
   (<table> ^type table)
-->
   (<table> ^clear yes)}

##############################################################################
# Calculate whether a block is clear
#
# The conditions establish that:
#  1. The state has a problem-space named 'blocks'.
#  2. The state has a thing of type block.
#  3. There is no 'ontop' relation having the block as its 'bottom-block'.
# The action:
#  1. create an acceptable preference for an attribute-value pair asserting
#     the block is clear.
#
# This production will retract whenever an 'ontop' relation for the given block
```

```
#  is created. Since the (<block> ^clear yes) wme only has i-support, it will
#  be removed from working memory automatically when the production retracts.


sp {elaborate*block*clear
   (state <s> ^problem-space blocks
              ^thing <block>)
   (<block> ^type block)
   -(<ontop> ^bottom-block <block>)
-->
   (<block> ^clear yes)}



################################################################################
# Suggest MOVE-BLOCK operators
#
# This production proposes operators that move one block ontop of another block.
# The conditions establish that:
#  1. The state has a problem-space named 'blocks'
#  2. The block moved and the block moved TO must be both be clear.
#  3. The block moved is different from the block moved to.
#  4. The block moved must be type block.
#  5. The block moved must not already be ontop the block being moved to.
# The actions:
#  1. create an acceptable preference for an operator.
#  2. create acceptable preferences for the substructure of the operator (its
#     name, its 'moving-block' and the 'destination).

sp {blocks-world*propose*move-block
   (state <s> ^problem-space blocks
              ^thing <thing1> {<> <thing1> <thing2>}
              ^ontop <ontop>)
   (<thing1> ^type block ^clear yes)
   (<thing2> ^clear yes)
   (<ontop> ^top-block <thing1>
            ^bottom-block <> <thing2>)
-->
   (<s> ^operator <o> +)
   (<o> ^name move-block
        ^moving-block <thing1>
        ^destination <thing2>)}

################################################################################
# Make all acceptable move-block operators also indifferent
#
# The conditions establish that:
#  1. the state has an acceptable preference for an operator
#  2. the operator is named move-block
# The actions:
#  1. create an indifferent prefererence for the operator

sp {blocks-world*compare*move-block*indifferent
   (state <s> ^operator <o> +)
   (<o> ^name move-block)
-->
   (<s> ^operator <o> =)}
```

```
###############################################################################
# Apply a MOVE-BLOCK operator
#
# There are two productions that are part of applying the operator.
# Both will fire in parallel.
###############################################################################

###############################################################################
# Apply a MOVE-BLOCK operator
#   (the block is no longer ontop of the thing it used to be ontop of)
#
# This production is part of the application of a move-block operator.
# The conditions establish that:
#  1. An operator has been selected for the current state
#     a. the operator is named move-block
#     b. the operator has a 'moving-block' and a 'destination'
#  2. The state has an ontop relation
#     a. the ontop relation has a 'top-block' that is the same as the
#        'moving-block' of the operator
#     b. the ontop relation has a 'bottom-block' that is different from the
#        'destination' of the operator
# The actions:
#  1. create a reject preference for the ontop relation

sp {blocks-world*apply*move-block*remove-old-ontop
   (state <s> ^operator <o>
              ^ontop <ontop>)
   (<o> ^name move-block
        ^moving-block <block1>
        ^destination <block2>)
   (<ontop> ^top-block <block1>
            ^bottom-block { <> <block2> <block3> })
-->
   (<s> ^ontop <ontop> -)}


###############################################################################
# Apply a MOVE-BLOCK operator
#   (the block is now ontop of the destination)
#
# This production is part of the application of a move-block operator.
# The conditions establish that:
#  1. An operator has been selected for the current state
#     a. the operator is named move-block
#     b. the operator has a 'moving-block' and a 'destination'
# The actions:
#  1. create an acceptable preference for a new ontop relation
#  2. create (acceptable preferences for) the substructure of the ontop
#     relation: the top block and the bottom block

sp {blocks-world*apply*move-block*add-new-ontop
   (state <s> ^operator <o>)
```

```
      (<o> ^name move-block
           ^moving-block <block1>
           ^destination <block2>)
-->
      (<s> ^ontop <ontop>)
      (<ontop> ^top-block <block1>
               ^bottom-block <block2>)}



################################################################################
################################################################################
# Detect that the goal has been achieved
#
# The conditions establish that:
#  1. The state has a problem-space named 'blocks'
#  2. The state has three ontop relations
#     a. a block named A is ontop a block named B
#     b. a block named B is ontop a block named C
#     c. a block named C is ontop a block named TABLE
# The actions:
#  1. print a message for the user that the A,B,C tower has been built
#  2. halt Soar

sp {blocks-world*detect*goal
   (state <s> ^problem-space blocks
                ^ontop <AB>
                 { <> <AB> <BC>}
                 { <> <AB> <> <BC> <CT> } )
   (<AB> ^top-block <A> ^bottom-block <B>)
   (<BC> ^top-block <B> ^bottom-block <C>)
   (<CT> ^top-block <C> ^bottom-block <T>)
   (<A> ^type block ^name A)
   (<B> ^type block ^name B)
   (<C> ^type block ^name C)
   (<T> ^type table ^name TABLE)
-->
   (write (crlf) |Achieved A, B, C|)
   (halt)}



################################################################################
################################################################################
# Monitor the state: Print a message every time a block is moved
#
# The conditions establish that:
#  1. An operator has been selected for the current state
#     a. the operator is named move-block
#     b. the operator has a 'moving-block' and a 'destination'
#  2. each block has a name
# The actions:
#  1. print a message for the user that the block has been moved to the
#     destination.

sp {blocks-world*monitor*move-block
   (state <s> ^operator <o>)
```

```
    (<o> ^name move-block
         ^moving-block <block1>
         ^destination <block2>)
    (<block1> ^name <block1-name>)
    (<block2> ^name <block2-name>)
-->
    (write (crlf) |Moving Block: | <block1-name>
                  | to: | <block2-name> ) }
```

Appendix B

Grammars for production syntax

This appendix contains the BNF grammars for the conditions and actions of productions. (BNF stands for Backus-Naur form or Backus normal form; consult a computer science book on theory, programming languages, or compilers for more information. However, if you don't already know what a BNF grammar is, it's unlikely that you have any need for this appendix.)

This information is provided for advanced Soar users, for example, those who need to write their own parsers.

B.1   Grammar of Soar productions

A grammar for Soar productions is:

```
<soar-production>  ::= sp "{" <production-name> [<documentation>] [<flags>]
                       <condition-side> --> <action-side> "}"
<documentation>    ::= """ [<string>] """
<flags>            ::= ":" (o-support | i-support | chunk | default)
```

B.1.1   Grammar for Condition Side

Below is a grammar for the condition sides of productions:

```
<condition-side>   ::= <state-imp-cond> <cond>*
<state-imp-cond>   ::= "(" (state | impasse) [<id_test>]
                       <attr_value_tests>+ ")"
<cond>             ::= <positive_cond> | "-" <positive_cond>
<positive_cond>    ::= <conds_for_one_id> | "{" <cond>+ "}"
<conds_for_one_id> ::= "(" [(state|impasse)] <id_test>
                       <attr_value_tests>+ ")"
<id_test>          ::= <test>
```

```
<attr_value_tests> ::= ["-"] "^" <attr_test> ("." <attr_test>)*
                       <value_test>*
<attr_test>         ::= <test>
<value_test>        ::= <test> ["+"] | <conds_for_one_id> ["+"]

<test>              ::= <conjunctive_test> | <simple_test>
<conjunctive_test> ::= "{" <simple_test>+ "}"
<simple_test>       ::= <disjunction_test> | <relational_test>
<disjunction_test> ::= "<<" <constant>+ ">>"
<relational_test>  ::= [<relation>] <single_test>
<relation>          ::= "<>" | "<" | ">" | "<=" | ">=" | "=" | "<=>"
<single_test>       ::= <variable> | <constant>
<variable>          ::= "<" <sym_constant> ">"
<constant>          ::= <sym_constant> | <int_constant> | <float_constant>
```

Notes on the Condition Side

- In an <id_test>, only a <variable> may be used in a <single_test>.

B.1.2   Grammar for Action Side

Below is a grammar for the action sides of productions:

```
<rhs>                     ::= <rhs_action>*
<rhs_action>              ::= "(" <variable> <attr_value_make>+ ")"
                             | <func_call>
<func_call>               ::= "(" <func_name> <rhs_value>* ")"
<func_name>               ::= <sym_constant> | "+" | "-" | "*" | "/"
<rhs_value>               ::= <constant> | <func_call> | <variable>
<attr_value_make>         ::= "^" <variable_or_sym_constant>
                             ("." <variable_or_sym_constant>)* <value_make>+
<variable_or_sym_constant> ::= <variable> | <sym_constant>
<value_make>              ::= <rhs_value> <preference_specifier>*

<preference-specifier>    ::= <unary-preference> [","]
                             | <unary-or-binary-preference> [","]
                             | <unary-or-binary-preference> <rhs_value> [","]
<unary-pref>              ::= "+" | "-" | "!" | "~" | "@"
<unary-or-binary-pref>    ::= ">" | "=" | "<" | "&"
```

Appendix C

The Calculation of O-Support

This appendix provides a description of when a preference is given O-support by an instantiation (a preference that is not given O-support will have I-support). Soar has four possible procedures for deciding support, which can be selected among with the o-support-mode command (see page 174). However, only o-support modes 3 & 4 can be considered current to Soar 8, and o-support mode 4 should be considered an improved version of mode 3. The default o-support mode is mode 4.

In O-support modes 3 & 4, support is given by production; that is, all preferences generated by the RHS of a single instantiated production will have the same support.

In both modes, a production must meet the following two requirements to create o-supported preferences:

1. The RHS has no operator proposals, i.e. nothing of the form

   ```
   (<s> ^operator <o> +)
   ```

2. The LHS has a condition that tests the current operator, i.e. something of the form [1]

   ```
   (<s> ^operator <o>)
   ```

In condition 1, the variable `<s>` must be bound to a state identifier. In condition 2, the variable `<s>` must be bound to the lowest state identifier. That is to say, each (positive) condition on the LHS takes the form (id $^\wedge$attr value), some of these id's match state identifiers, and the system looks for the deepest matched state identifier. The tested current operator must be on this state. For example, in the production-

```
sp {elaborate*state*operator*name
  (state <s> ^superstate <s1>)
  (<s1> ^operator <o>)
```

---

[1]Sometimes, o-support mode 3 does not notice that this condition is true. This is a bug, which is unlikely to be fixed, since users are encouraged to use mode 4.

```
   (<o> ^name <name>)
-->
   (<s> ^name something)}
```

the RHS action gets i-support. Of course, the state bound to `<s>` is destroyed when (`<s1>` $^\wedge$`operator <o>`) retracts, so o-support would make little difference. On the other hand, the production-

```
sp {operator*superstate*application
   (state <s> ^superstate <s1>)
   (<s> ^operator <o>)
   (<o> ^name <name>)
 -->
   (<s1> ^sub-operator-name <name>)}
```

gives o-support to its RHS action, which remains after the substate bound to `<s>` is destroyed.

There is a third condition that determines support, and it is in this condition that modes 3 & 4 differ. An extension of condition 1 is that operator augmentations should always receive i-support. Soar has been written to recognize augmentations directly off the operator (ie, (`<o>` $^\wedge$`augmentation value`)), and to attempt to give them i-support. However, there was some confusion about what to do about a production that simultaneously tests an operator, doesn't propose an operator, adds an operator augmentation, and adds a non-operator augmentation, such as-

```
sp {operator*augmentation*application
  (state <s> ^task test-support
         ^operator <o>)
-->
   (<o> ^new augmentation)
   (<s> ^new augmentation)}
```

In o-support mode 3, both RHS actions receive o-support; in o-support mode 4, both receive i-support. In either case, Soar will print a warning on firing this production, because this is considered bad coding style.

Appendix D

The Resolution of Operator Preferences

During the decision phase, operator preferences are evaluated in a sequence of eight steps, in an effort to select a single operator. Each step handles a specific type of preference, as illustrated in Figure D.1. (The figure should be read starting at the top where all the operator preferences are collected and passed into the procedure. At each step, the procedure either exits through a arrow to the right, or passes to the next step through an arrow to the left.)

Input to the procedure are the set of current operator preferences, and the output consists of:

1. a subset of the candidate operators, either the empty set, a set consisting of a single, winning candidate, or a larger set of candidates that may be conflicting, tied, or indifferent.

2. an impasse-type, possibly NONE_IMPASSE_TYPE.

The procedure has several potential exit points. Some occur when the procedure has detected a particular type of impasse. The others occur when the number of candidates has been reduced to one (necessarily the winner) or zero (a no-change impasse).

Each step in Figure D.1 is described below:

RequireTest (!) This test checks for required candidates in preference memory and also constraint-failure impasses involving require preferences (see Section 2.6 on page 23).

- If there is exactly one candidate operator with a require preference and that candidate does not have a prohibit preference, then that candidate is the winner and preference semantics terminates.

- Otherwise — If there is more than one required candidate, then a constraint-failure impasse is recognized and preference semantics terminates by returning the set of required candidates.

Preference resolution:

-all operator preferences are input to the resolution procedure
-each step may add or remove some operator candidates
-only some steps may exit

Figure D.1: An illustration of the preference resolution process. There are eight steps; only five of these provide exits from the resolution process.

- Otherwise — If there exists a required candidate that is also prohibited, a constraint-failure impasse with the required/prohibited value is recognized and preference semantics terminates.

- Otherwise — The candidates are passed to AcceptableCollect.

AcceptableCollect (+)   This operation builds a list of operators for which there is an acceptable preference in preference memory. This list of candidate operators is passed

to the ProhibitFilter.

ProhibitFilter ($\sim$) This filter removes the candidates that have prohibit preferences in memory. The rest of the candidates are passed to the RejectFilter.

RejectFilter ($-$) This filter removes the candidates that have reject preferences in memory.

Exit Point 1
- At this point, if the set of remaining candidates is either empty or has one member, preference semantics terminates and this set is returned.
- Otherwise, the remaining candidates are passed to the BetterWorseFilter.

BetterWorseFilter ($>$), ($<$) This filter removes any candidates that are worse than another candidate.

Exit Point 2
- If the set of remaining candidates is empty, a conflict impasse is created returning the set of all candidates passed into this filter, i.e. all of the conflicted operators.
- If the set of remaining candidates has one member, preference semantics terminates and this set is returned.
- Otherwise, the remaining candidates are passed to the BestFilter.

BestFilter ($>$) If some remaining candidate has a best preference, this filter removes any candidates that do not have a best preference. If there are no best preferences for any of the current candidates, the filter has no effect. The remaining candidates are passed to the WorstFilter.

Exit Point 3
- At this point, if the set of remaining candidates is either empty or has one member, preference semantics terminates and this set is returned.
- Otherwise, the remaining candidates are passed to the WorstFilter.

WorstFilter ($<$) This filter removes any candidates that have a worst preference. If all remaining candidates have worst preferences or there are no worst preferences, this filter has no effect.

Exit Point 4
- At this point, if the set of remaining candidates is either empty or has one member, preference semantics terminates and this set is returned.
- Otherwise, the remaining candidates are passed to the IndifferentFilter.

IndifferentFilter ($=$) This operation traverses the remaining candidates and marks each candidate for which one of the following is true:

- the candidate has a unary indifferent preference
- the candidate has a numeric indifferent preference

This filter then checks every candidate that is not one of the above two types to see if it has a binary indifferent preference with every other candidate. If one of the candidates fails this test, then the procedure signals a tie impasse and returns the complete set of candidates that were passed into the IndifferentFilter. Otherwise, the candidates are mutually indifferent, in which case an operator is chosen according to the method set by the **indifferent-selection** command, described on page 164.

Appendix E

A Goal Dependency Set Primer[1]

This document briefly describes the Goal Dependency Set (GDS), which was introduced with Soar 8. There are three sections: a brief discussion of the motivation for the GDS, a discussion of the consequences of the GDS from a behavior developer/modeler's point of view, and some details on the kernel implementation of the GDS, for anyone working at the architecture level. This document is by no means complete, but introduces the GDS in Soar-specific terms.

Why the GDS was needed

As a symbol system, Soar attempts to approximate the knowledge level but will necessarily always fall short . We can informally think of the way in which Soar falls short of the knowledge level as its peculiar "psychology." Those interested in using Soar to model human psychology would like Soar's "psychology" to approximate human psychology. Those using Soar to create agent systems would like to make Soar's processing approximate the knowledge level as closely as possible. However, Soar 7 had a number of symbol-level "quirks" that appeared inconsistent with human psychology and that made building large-scale, knowledge-based systems in Soar more difficult than necessary. Bob Wray's thesis [2] addressed many of these symbol-level problems in Soar, among them logical inconsistency in symbol manipulations, non-contemporaneous constraints in chunks , race conditions in rule firings and in the decision process, and contention between original task knowledge and learned knowledge .

The Goal Dependency Set implements a solution to logical inconsistencies between persistent (o-supported) working memory elements (WMEs) in a substate and its "context". The context consists of all the WMEs in any superstates above the local goal/state[3]. In Soar, any action (application) of an operator receives an o-support preference. This preference

---

[1]A preliminary draft by Robert Wray, contact at `wrayre@acm.org`.

[2]Robert E. Wray. *Ensuring Reasoning Consistency in Hierarchical Architectures*. PhD thesis, University of Michigan, 1998.

[3]This report will use "state," not "goal." At the kernel level, states are still called "goals" and "goal" is often still used to refer to states. As a result, a confusion in terminology results, with "**Goal** Dependency Set" a specific example, even though "goals" have not been an explicit, behavior-level Soar construct since

makes the resulting WME persistent: it will remain in memory until explicitly removed (or until its local state is removed), regardless of whether it continues to be justified.

Persistent WMEs are pervasive in Soar, because operators are the main unit of problem solving. Persistence is necessary for taking any non-monotonic step in a problem space. However, persistent WMEs also are dependent on WMEs in the superstate context. The problem in Soar 7, especially when trying to create large-scale systems like TacAir-Soar , is that the knowledge developer must always think about which dependencies can be "ignored" and which need to result in a reconsideration of the persistent WME. For example, imagine an exploration robot that makes a persistent decision to travel to some distant destination based, in part, on its power reserves. Now suppose that the agent notices that its power reserves have failed. If this change is not communicated to the state where the travel decision was made, the agent will continue to act as if its full power reserves were still available.

Of course, for this specific example, the knowledge designer can encode some knowledge to react to this inconsistency. The fundamental problem is that the knowledge designer has to consider *all* possible interactions between all o-supported WMEs and all contexts. Soar systems often use the architecture's impasse mechanism to realize a form of decomposition. These potential interactions mean that the knowledge developer cannot focus on individual problem spaces when creating knowledge, which makes knowledge development more difficult. Further, in all but the simplest systems, the knowledge designer will miss some potential interactions. The result is agents are that were unnecessarily brittle, failing in difficult-to-understand, difficult-to-duplicate ways.

The GDS also solves the the problem of non-contemporaneous constraints in chunks. A non-contemporaneous constraint refers to two or more conditions that never co-occur simultaneously. An example might be a driving robot that learned a rule that attempted to match "red light" and "green light" simultaneously. Obviously, for functioning traffic lights, this rule would never fire. By ensuring that local persistent elements are always consistent with the higher-level context, non-contemporaneous constraints in chunks are *guaranteed* not to happen.

The GDS captures context dependencies during processing, meaning the architecture will identify and respond to inconsistencies automatically. The knowledge designer then does not have to consider potential inconsistencies between local, o-supported WMEs and the context. The following sections describe further how the GDS works and how to use the GDS in behavior systems, as well as how the GDS is implemented in the Soar kernel.

Behavior-level view of the Goal Dependency Set

This section discusses what the GDS does, and how that impacts production knowledge design and implementation.

Soar 6

Figure E.1: Simplified Representation of the context dependencies (above the line), local os-upported WMEs (below the line), and the generation of a result. In Soar 7, this situation led to non-contemporaneous constraints in the chunk that generates **3**.

Operation of the Goal Dependency Set

Whenever a feature is created (added to working memory) in the Soar 7 architecture, that feature will persist for some time. The persistence of features may differ with respect to how long the features remain in memory, and more importantly, what circumstances cause the feature to be removed. The Soar 7 architecture utilizes three primary types of persistence: i-support, o-support, and c-support.

The weakest persistence is instantiation support. An i-supported feature exists in memory only as long as the production which lead to the feature's creation remains instantiated. Thus, the WME depends upon this production instantiation (and, more specifically, the features the instantiation tests). When one of the conditions in the production instantiation no longer matches, the instantiation is retracted, resulting in the loss of the acceptable preference for the WME.[4] I-support is illustrated in Figure E.1. A copy of **A** in the subgoal, **A**$_s$, is retracted automatically when **A** changes to **A'**. The substate WME persists only as long as it remains justified by **A**. This justification is called "instantiation support" (I-support) in Soar (and should not be confused with result *justifications*.)

In the broadest sense, we can say that some feature <b> is "dependent" upon another element <a> if <a> was used in the creation of <b>, i.e., if <a> was tested in the production instantiation that created <b>. Further, a dependent change with respect to feature <b> is a change to any of its instantiating features. In Figure E.1, the change from **A** to **A'** is a

---

[4]Importantly, in a technical sense, the WME is only retracted when it loses instantiation support, not when the creating production is retracting. For example, a WME could receive i-support from several different instantiations and the retraction of one would not lead to the retraction of the WME. However, the the following generally discusses direct dependency unmediated by preferences, ignoring this complication for clarity.

dependent change for feature **1** because **A** was used to create **1**.

In Soar 7, some features are insensitive to dependent changes. These features are often referred to as "persistent WMEs" because, unlike i-supported WMEs, they remain in memory until explicitly removed. There are two different types of this stronger persistence: o-support and c-support.

Any feature created by the action of an operator receives "operator support." An o-supported feature remains in memory until explicitly rejected (or until the superstructure to which it is attached is removed). Removal is architecturally independent of the WME's instantiating conditions.

Context-support affects the persistence of an operator itself, rather than its effects. Once a unique operator has been chosen by the decision procedure, the choice persists until explicitly re-decided (via a reconsider preference). C-support ensures that the WME for a selected operator remains available even if the production that proposed the operator is no longer instantiated. Soar 8 eliminates c-support, so that operators now persist only as long as they receive instantiation support. This change was integral to the overall solution Soar 8 provides, but is distinct from the GDS.

The GDS provides a solution to the first problem. When **A** changes, the persistent WME **1** may be no longer consistent with its context (e.g., **A'**). The specific solution is inspired by the chunking algorithm. In Soar 8, whenever an o-supported WME is created in the local state, the superstate dependencies of that new feature are determined and added to the *goal dependency set* (GDS) of that state. Conceptually speaking, whenever a working memory change occurs, the dependency sets for every state in the context hierarchy are compared to working memory changes.[5] If a removed element is found in a GDS, the state is removed from memory (along with all existing substructure). The dependency set includes only dependencies for o-supported features. For example, in Figure E.2, at time t_0, because only i-supported features have been created in the subgoal, the dependency set is empty.

Three types of features can be tested in the creation of an o-supported feature. Each requires a slightly different type of update to the dependency set.

Elements in the superstate: WMEs in the superstate are added directly to the goal's dependency set. In Figure E.2, the persistent subgoal item **3** is dependent upon **A** and **D**. These superstate WMEs are added to the subgoal's dependency set when **3** is added to working memory at time t_1. It does not matter that **A** is i-supported and **D** o-supported.[6]

Local I-Supported Features: Local i-supported features are not added to the goal dependency set. Instead, the superstate WMEs that led to the creation of the i-supported feature are determined and added to the GDS. In the example, when **4** is created, **A**, **B** and **C** must be added to the dependency set because they are the superstate

---

[5]The implementation is slightly different, trading additional memory overhead to avoid scanning all the goal dependency sets after each WM change. See the next section.

[6]In addition, superstate WMEs can also include context slot preferences, which are represented in the architecture as working memory elements.

Figure E.2: The Dependency Set in Soar 8.

features that led to **1**, which in turn led to **2** and finally **4**. However, because item **A** was previously added to the dependency set at t_1, it is unnecessary to add it again.

Local O-Supported Features: The dependencies of a local o-supported feature have already been added to the state's GDS. Thus, tests of local o-supported WMEs do not require additions to the dependency set. In Figure E.2, the creation of element **5** does not change the dependency set because it is dependent only upon persistent items **3** and **4**, whose features had been previously added to the GDS.

In Soar 8, any change to the current dependency set will cause the retraction of all subgoal structure. Thus, any time after time t_1, either the **D** to **D'** or **A** to **A'** transition would cause the removal of the entire subgoal. The **E** to **E'** transition causes no retraction because **E** is not in the goal's dependency set.

The role of the GDS in agent design

The GDS places some design time constraints on operator implementation. These constraints are:

- Operator actions that are used to remember a previous state/situation should be asserted in the top state

- All operator elaborations should be i-supported

- Any operator with local actions should be designed to be re-entrant

This section describes these issues.

Soar says any operator effect is o-supported, regardless of whether that assertion is entailed by the current situation, or whether it reflects an assumption about it. The GDS adds additional (needed) constraint. Because any context dependencies for subgoal, o-supported assertions will be added to the GDS, the developer must decide if an o-supported element should be represented in a substate or the top state.

This decision is straightforward if the functional role of the persistent element is considered. Four important capabilities that require persistence are:

1. **Reasoning hypothetically:** Some assertions may need to reflect hypothetical states. Such assertions are "assumptions" because a hypothetical inference cannot always be grounded in the current context. In other problem solvers with truth maintenance, only assumptions are persistent.

2. **Reasoning non-monotonically:** Sometimes the result of an inference changes one of the assertions on which the inference is dependent. As an example, consider the task of counting. Each newly counted item replaces the old value of the count.

3. **Remembering:** Agents oftentimes need to remember an external situation or stimulus, even when that perception is no longer available.

4. **Avoiding Expensive Computations:** In some situations, an agent may have the information needed to assert some belief in a new world state but the expense of performing the computation necessary for the assertion, given what is already known, makes the computation avoidable. For example, in dynamic, complex domains, determining when to make an expensive calculation is often formulated as an explicit agent task .

When remembering or avoiding an expensive computation, the agent/designer is making a commitment to retain something even though it might not be supported in the current context. In Soar 8, these WMEs should be asserted in the top state. *For many Soar systems, especially those focused on execution in a dynamic environment, most o-supported elements will need to be stored on the top state.*

For any kind of local, non-monotonic reasoning about the context (counting, projection planning), features should be stored locally. When a dependent context change occurs, the GDS interrupts the processing by removing the state. While this may seem like a severe over-reaction, formal and empirical analysis have suggested that this solution is less computationally expensive than attempting to identify the specific dependent assumption .

Operator Elaborations

Operator elaborations (i.e., placing some information on an operator WME) should be i-supported when using Soar 8, since this information is, by definition, temporary/not persistent (because it's located on the non-persistent operator). However, the kernel itself hasn't kept up with this change. Prior to Soar 8.5, Soar's o-support modes computed operator

elaborations as o-supported, resulting in the context conditions being added to the GDS. This often leads to unwanted/unnecessary retractions. If you are using a version prior to Soar 8, you should declare any operator elaborations i-supported (i.e., using :i-support).

Kernel-level view of the Goal Dependency Set

The actual implementation of the GDS in the Soar kernel is slightly more complex than the conceptual description of the previous section (but not significantly so).

Elements are added to the GDS via `elaborate_gds()`, a procedure in decide.c that mimics the chunking backtrace function. The algorithm is shown in Figure E.3. When an o-supported preference is asserted, elaborate_gds() is called. Conditions in a production instantiation that are located in a higher context can be added directly to the GDS (1). For local conditions, `elaborate_gds()` first checks whether the tested WME is o-supported, or if it has been previously been back traced through (2). If either of these are true, the WME can be ignored because it's dependencies have been added to the GDS previously. If not, `elaborate_gds()` is called recursively, to find the context dependencies for the local, contributing WME, c (3).

When WME changes occur, each goal/state must be checked to determine if the WME appeared on that goal's GDS. Because WME changes occur in nearly every Soar elaboration cycle, we chose to extend the WME data structure to avoid this scanning. Figure E.4 illustrates the relationship. Each GDS structure consists of a pointer to its goal and a pointer to a linked list of WMEs. The `gds_next` and `gds_prev` pointers on the WME structure define the GDS WMEs for a particular GDS and the GDS pointer provides a link back from each GDS WME to the GDS data structure.

When a WME is removed, the GDS pointer can be checked to determine immediately if the goal should be removed. No scanning is necessary.

Other implementation issues

- Allocating memory for the GDS
  The GDS memory is created for each goal when the goal is created. The GDS is deallocated when the goal is removed. A NIL WME pointer for the GDS indicates a goal has no WMEs in its GDS.

- Updating a WME GDS pointer
  A WME should appear in only the GDS of the highest goal for which it is dependent. If a WME is determined to already be in a GDS lower than the current goal, its GDS pointer is updated to the higher goal, it is removed from the gds_WME DLL of the lower goal, and added to the higher one. If there are no other WMEs on the gds_WME DLL of the lower goal, its WME pointer is set to NIL (the GDS itself is retained, because we don't want to have to reallocate memory for the GDS if we need to add to it later.)

PROC *create_new_assertion*(...)
    Whenever a new o-supported element is asserted, the GDS is updated
    to include any new context dependencies.
    . . .
    $A_{inst} \leftarrow$ instantiation that asserted acceptable preference for A
    IF A is an o-supported WME
        G is the goal/state in which A is asserted
        $G_{GDS} \leftarrow append(G_{GDS}, elaborate\_GDS(A))$
    . . .
END

PROC *elaborate_GDS*(*assertion A*)
    $S \leftarrow \{NIL\}$
    FOR Each assertion c in A_{inst}, the instantiation supporting A
①   **IF** $\{GoalLevel(c)$ closer to top state than $GoalLevel(A)\}$
            $append(c, S)$ (append context dependency to GDS)

②   **ELSEIF** $\{GoalLevel(c)$ same as $GoalLevel(A)$ AND
               c is NOT an o-supported WME   AND
               c has not previously been inspected $\}$
③ $S \leftarrow append(S, elaborate\_GDS(c))$
               (compute GDS dependencies for c and add to goal's GDS)
④ $c_{inspected} \leftarrow true$
               (c's context dependencies have been added to the GDS;
                  no need to consider it again for this GDS)
    return S, the list of new dependencies in the GDS
END

PROC *GoalLevel*(*assertion   A*)
    Return the goal level associated with assertion A

Figure E.3: The algorithm for determining members of the GDS.

Figure E.4: The GDS and WME data structures

.

Index

!, 58, 78, 227
&, 58
+, 48, 58, 229
-, 45, 58, 229
., 50
<, 43, 58, 229
<<, >>, 44, 49
<=, 43
<=>, 43
<>, 43
=, 43, 58, 229
>, *see* best preference, 43, 58, 229
>=, 43
@, 58
^, (carat symbol), 33
~, 58, 229

, 29

acceptable preference, 48, 229
action side of production, 56
action-side grammar, 224
add-wme, 203
alias, 208
allocate, 209
arithmetic operations, 60
attribute, 8, 13, 14, 33, 34
    multi-valued attribute, 35
augmentation, *see* working memory element

backtracing, 75, 76
best preference, *see* best preference, 78, 229
better preference, 78, 229
bottom-up chunking, 74

capitalize-symbol, 64
capture-input, 205
carriage return, line feed, 60
cd, 193
chunk, 30
    overgeneral, 28
chunk-name-format, 140

chunking, *see* learning, 73
    actions, 75
    bottom-up, 74
    conditions, 76, 80
    creation, 73
    determining actions, 75
    determining conditions, 76
    duplicate chunks, 74
    incorrect chunks, 77, 80
    negated conditions, 75, 82
    ordering conditions, 80
    overgeneral, 77, 80
    refractory inhibition, 80
    variablization, 79
    when active, 73
cli, 209
clog, 194
cmd, 65
command-to-file, 196
comments, 40
compute, 60
concat, 64
condition
    acceptable preference , 48
condition side, 41
condition-side grammar, 223
Conditions, 41
conflict impasse, 24, 67
conjunctive
    conditions, 44
    negation, 46
constant, 34, 224
constraint-failure impasse, 24, 67, 227
Context-Dependent Preference Set, 28, 76, 77, 81
crlf, 60

decision
    procedure, 19, 66, 78, 227
decision cycle, 7, 21
    pseudo code, 23

241

Summary of Soar Aliases and Functions

Predefined Aliases

There are a number of Soar "commands" that are shorthand for other Soar commands:

Alias	Command	Page
?	help	120
a	alias	208
aw	add-wme	203
chdir	cd	193
d	run -d 1	121
dir	ls	198
e	run -e 1	121
eb	explain-backtraces	162
ex	excise	117
fc	firing-counts	141
gds_print	gds-print	128
h	help	120
inds	indifferent-selection	164
init	init-soar	121
interrupt	stop-soar	125
is	init-soar	121
l	learn	166
man	help	120
p	print	135
pc	print --chunks	135
pr	preferences	133
pw	pwatch	143
rn	rete-net	199
rw	remove-wme	206
set-default-depth	default-wme-depth	127
sn	soarnews	214
ss	stop-soar	125
st	stats	144
step	run 1	121
stop	stop-soar	125
topd	pwd	199
un	alias -d	208
unalias	alias -d	208
w	watch	148
wmes	print -i	135

Summary of Soar Functions

The following table lists the commands in Soar. See the referenced page number for a complete description of each command.

Command	Summary	Page
add-wme	Manually add an element to working memory.	203
alias	Define a new alias, or command, using existing commands and arguments.	208
allocate	Allocate additional 32 kilobyte blocks of memory for a specified memory pool without running Soar.	209
capture-input	Store the input wmes in a file for reloading later.	205
cd	Change directory.	193
chunk-name-format	Specify format of the name to use for new chunks.	140
cli	Manipulate external command line interface modules.	209
clog	Record all user-interface input and output to a file.	194
command-to-file	Dump the printed output and results of a command to a file.	196
default-wme-depth	Set the level of detail used to print WMEs.	127
dirs	List the directory stack.	196
echo-commands	Set whether or not commands are echoed to other connected debuggers.	210
echo	Print a string to the current output device.	197
edit-production	Move focus in an editor to this production.	211
epmem	Control the behavior of episodic memory.	155
excise	Delete Soar productions from production memory.	117
explain-backtraces	Print information about chunk and justification backtraces.	162
firing-counts	Print the number of times each production has fired.	141
gds-print	Print the WMEs in the goal dependency set for each goal.	128
gp-max	Set the upper-limit to the number of productions generated by the gp command.	119
gp	Generate productions according to a specified pattern.	118
help	Provide formatted usage information about Soar commands.	120
indifferent-selection	Controls indifferent preference arbitration.	164
init-soar	Empties working memory and resets run-time statistics.	121
internal-symbols	Print information about the Soar symbol table.	129
learn	Set the parameters for chunking.	166