The Soar User's Manual

Version 8.6

Edition 1

John E. Laird and Clare Bates Congdon
Electrical Engineering and Computer Science Department
University of Michigan

Draft of: June 14, 2005

Draft: Do not quote or distribute.

Errors may be reported to John E. Laird (laird@umich.edu)

Development of earlier versions of this manual were supported under contract N00014-92-K-2015 from the Advanced Systems Technology Office of the Advanced Research Projects Agency and the Naval Research Laboratory, and contract N66001-95-C-6013 from the Advanced Systems Technology Office of the Advanced Research Projects Agency and the Naval Command and Ocean Surveillance Center, RDT&E division.

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 8.5.

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 them.

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. In Soar there is 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).
2. All decisions are made through the combination of relevant knowledge at runtime. 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 learning mechanism. Not all users will make use of chunking, but it is important to know that this capability exists.

Chapter 5 describes the Soar user interface — how the user interacts with Soar. The chapter is a catalog of user-interface commands, grouped by functionality. For quick reference, an alphabetical listing of commands is also provided on the back cover of the manual.

Advanced users will refer most often to Chapter 5, 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*.

For novice Soar users, try *The Soar 8 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.

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`.

To report Soar bugs, to check whether a bug has been reported, or to check the status of a previously reported bug, visit:

`https://winter.eecs.umich.edu/soar-bugzilla/`.

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
1101 Beal Ave.
Ann Arbor, MI 48109-2110
USA

1.3 A Note on Different Platforms and Operating Systems

Soar runs on a wide variety of computers, including Unix (and Linux) machines, Macintoshes, and PCs running the Windows (95, 98, NT) operating system.

This manual documents Soar generally, although all references to files and directories use Unix format rather than Macintosh or Windows 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 2.2.

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

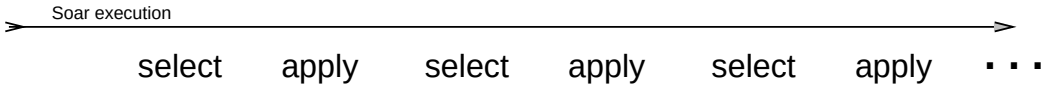

Figure 2.2: Soar is continually trying to select and apply operators.

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 application. These functions are composed of 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 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. Some productions *retract* their actions when the conditions are no longer met; this will be discussed later.

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;

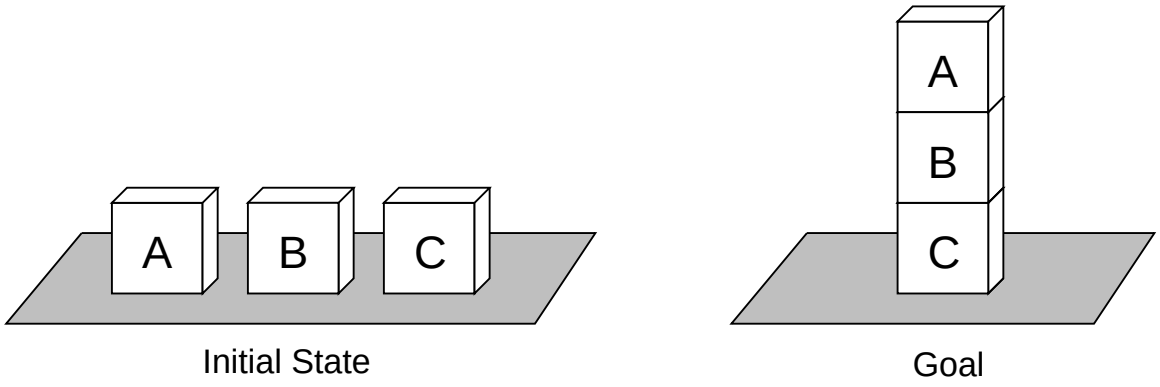

Figure 2.4: The initial state and goal of the "blocks-world" task.

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 2.4.

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)

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 2.6.

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

Figure 2.8 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

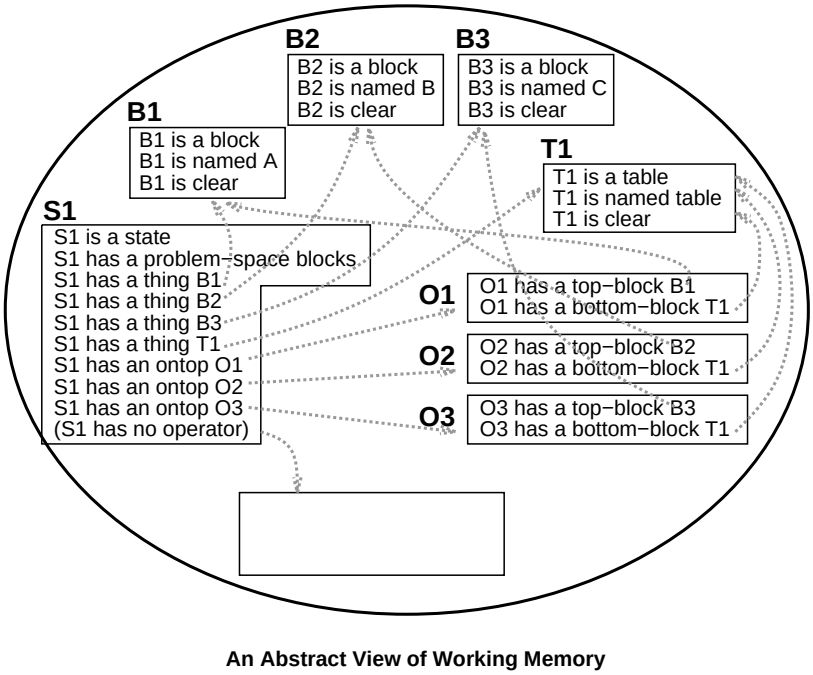

Figure 2.6: An abstract illustration of the initial state of the blocks world as working memory objects. At this stage of problem solving, no operators have been proposed or selected.

---

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 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.

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 different (but similar) operators for the initial state as illustrated in Figure 2.10. 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,

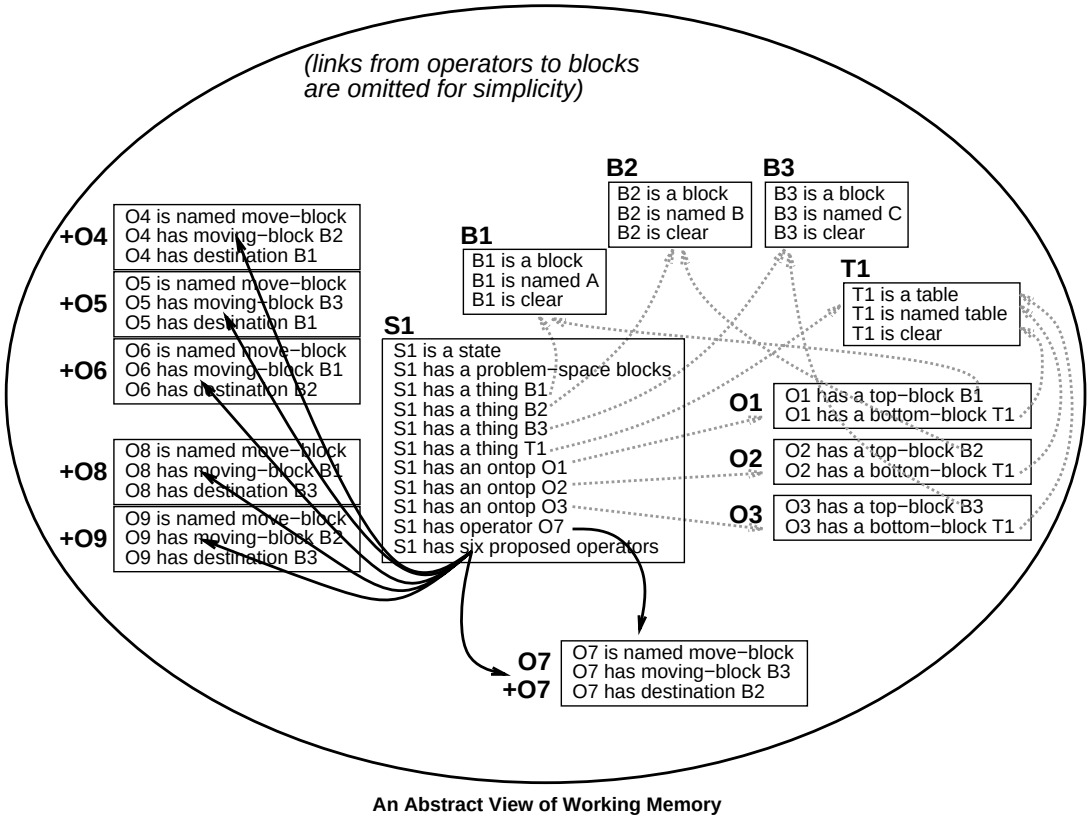

Figure 2.8: An abstract illustration of working memory in the blocks world after the first operator has been selected.

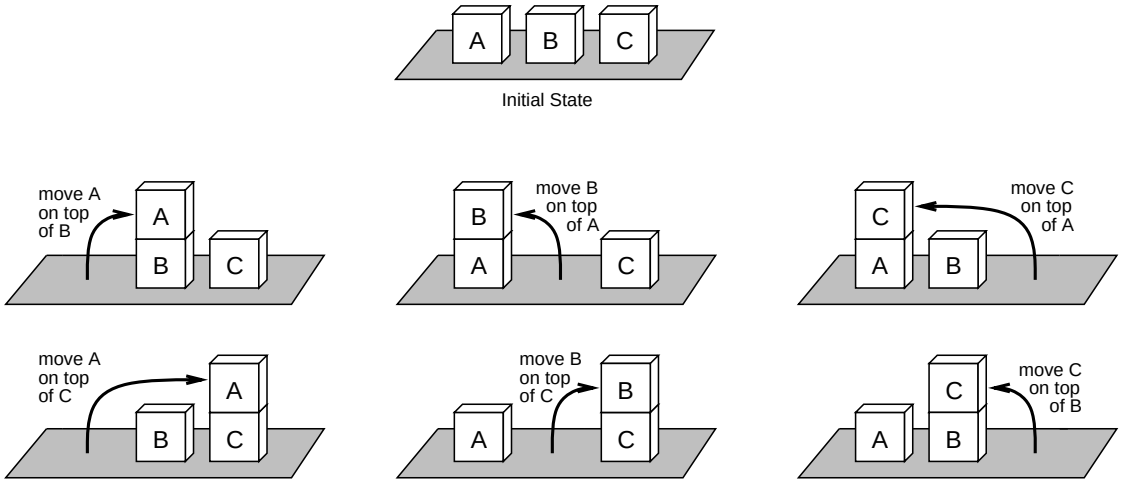

Figure 2.10: The six operators proposed for the initial state of the blocks world each move one block to a new location.

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. 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 shown 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].

The blocks-world task could also be implemented using an external simulator (as is done in the examples in *The Soar Coloring Book*). 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 as through operator applications or changes in sensory data.

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

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

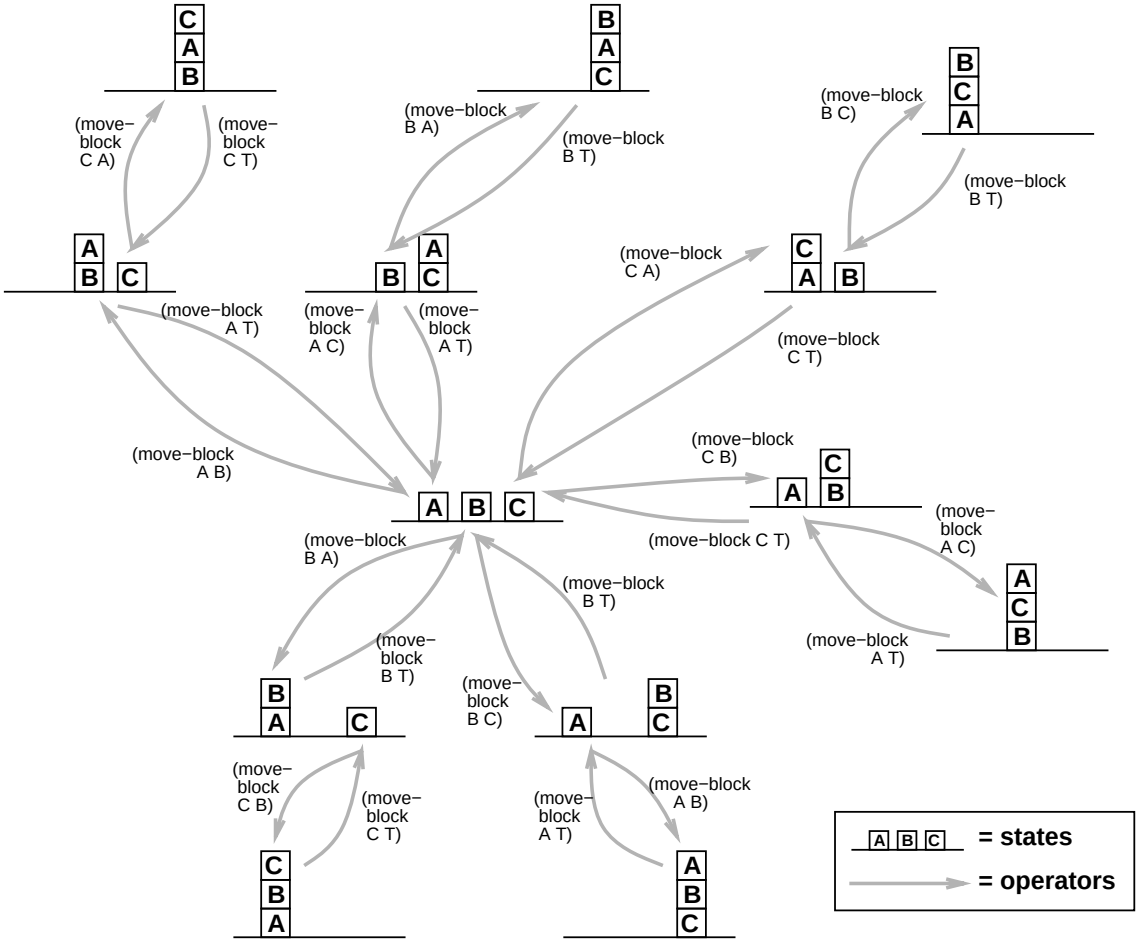

Figure 2.12: The problem space in the blocks-world includes all operators that move blocks from one location to another and all possible configurations of the three blocks.

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 2.12. 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.

The following sections describe the memories and processes of Soar: working memory, production memory, preference memory, the decision procedure, learning, and input and output.

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 (as well as any substates and operators generated because of impasses) 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 small 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 called 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 2.6 on page 9.

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 `O6`. 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.

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 allowed 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, in a sense, defined by its augmentations and *not* by its identifier. 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, so 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`; a feature of an object, such as the color `red`; a relation between objects, such as `ontop`; classes of objects, such as `blocks`; etc. The names of attributes and 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 names and to use them consistently.

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

1. The actions of productions create most working memory elements. The actions of productions must not create or modify the working memory elements created by the decision procedure or the I/O system (described below).
2. The decision procedure automatically creates some special state augmentations (type, superstate, impasse, ...) when 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. The I/O system creates working memory elements on the input-link for sensory data.

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

[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.

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.
4. Productions remove the i-supported working memory elements they created when they 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 Soar automatically creates are listed in Section 3.4. Productions may create any other attributes for states.

Preferences are held in *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.

2.3 Productions: Long-term Knowledge

Soar represents long-term knowledge as *productions* that are stored in *production memory*, illustrated in Figure 2.14. 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
```

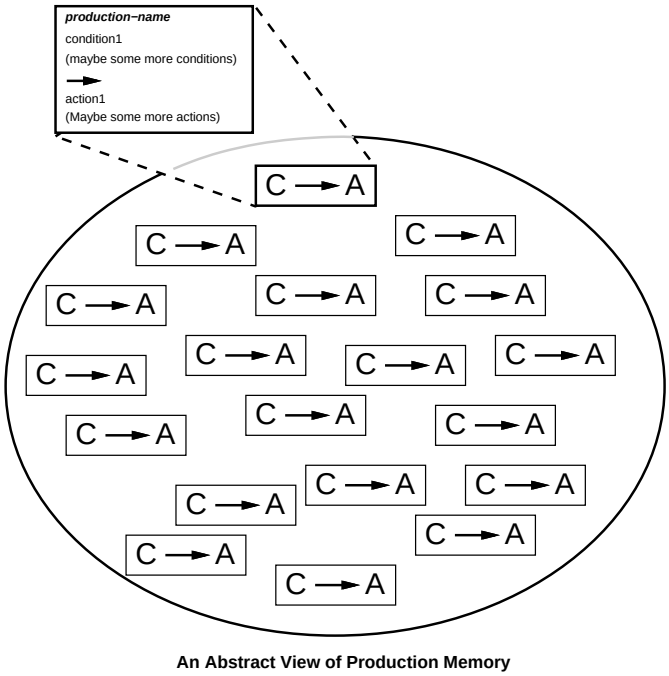

Figure 2.14: An abstract view of production memory. The productions are not related to one another.

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 multiple occurrences of the same variable is used to link 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, including the three knowledge-retrieval problem-solving functions and state elaboration function, as 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

The two main actions of a production are to create preferences for operator selection, and create or 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 change as the state changes, so that 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, 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 `rejected`, `prohibited`, or `worse` than another), it will be selected over any other value that is not also `best` (or `required`). 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 `rejected` or `prohibited`) is `better` than a `best` value, the `better` value will be selected. (This result is counter-intuitive, 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 120 in Chapter 5.)

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. Additionally, the preference weights the operator's probability of being selected according to the number given. For instance, given the preferences

```
(<s> ^operator <o1> = 40)
(<s> ^operator <o2> = 10)
```

the operator bound to `<o1>` would be more likely to be selected, whereas

```
(<s> ^operator <o1> =)
(<s> ^operator <o2> =)
```

would give equal probability to the two choices. There are two schemes for combining multiple numeric-indifferent preferences and performing the probabilistic selection; details are given in the description of the `numeric-indifferent-mode` command on page 126.

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.

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.

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

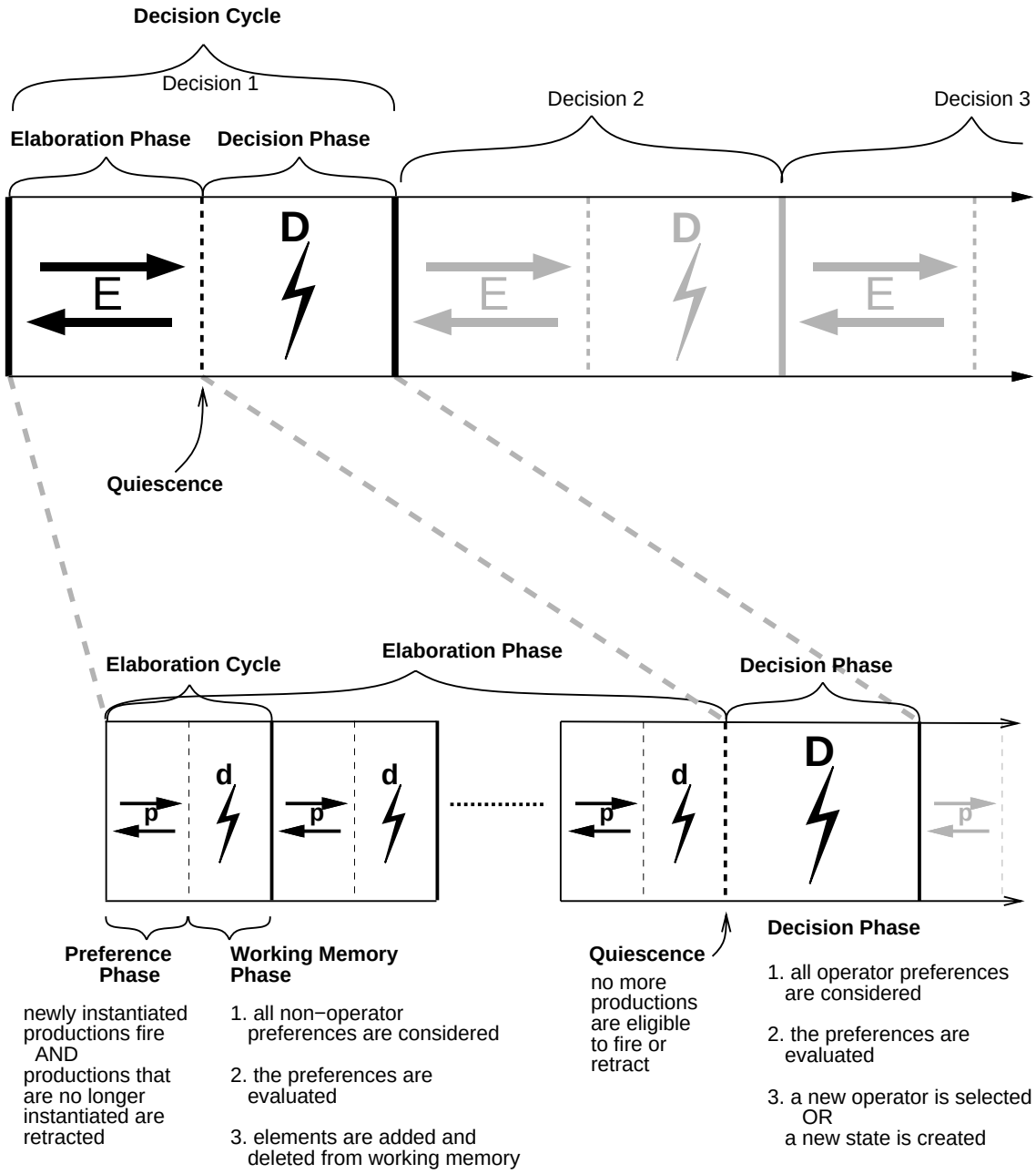

Figure 2.16: A detailed illustration of Soar's decision cycle: out of date

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.

```
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 2.17: A simplified version of the Soar algorithm.

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 `rejected`). 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.

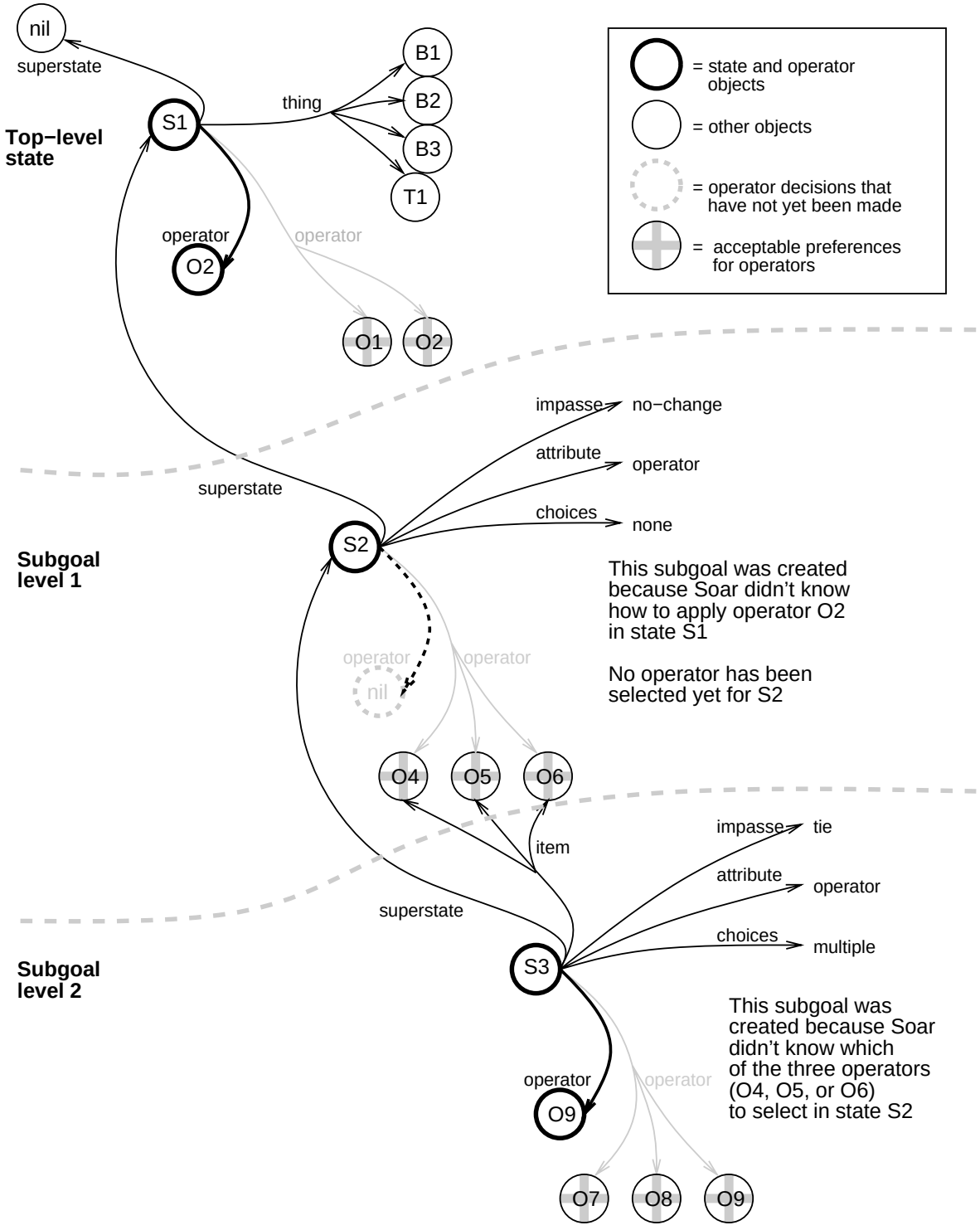

Figure 2.19: A simplified illustration of a subgoal stack.

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 2.19 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 elmenet 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 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

[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.

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 4.6 on page 80 for details).

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 2.19, 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.

Soar's learning mechanism 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 backtracing 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 2.8.

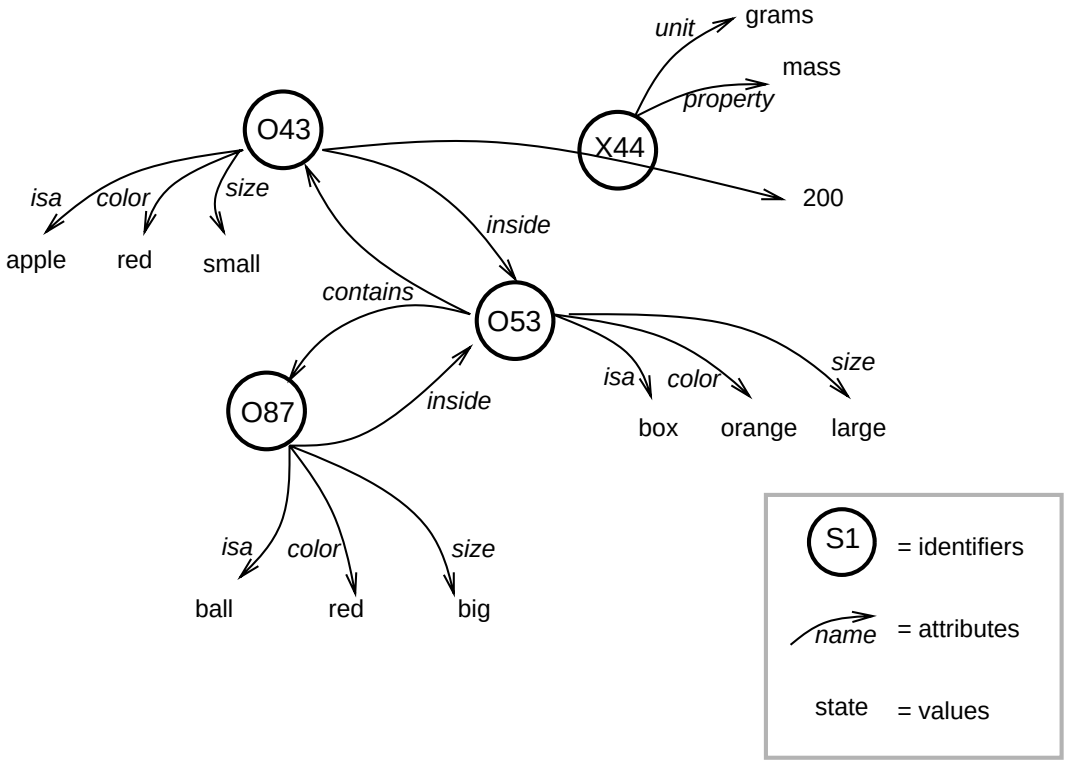

Figure 3.2: A semantic net illustration of four objects in working memory.

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 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 3.2 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 sourced 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 5.2.7 on page 100.

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

```
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 3.3: An example production from the example blocks-world task.

---

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
    }
```

An example production, named "`blocks-world*propose*move-block`", is shown in Figure 3.3. 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 5.1.6, which begins on Page 89.

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 3.3.6.7 on Page 60.

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 predi-

cates, 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 3.3, 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 3.3, 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 3.3, 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> ^child oprah)` or `(<p1> ^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 operators, 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, ^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 3.3 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 3.3 (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 ^operator value)`	acceptable
`(id ^operator value +)`	acceptable
`(id ^operator value !)`	require
`(id ^operator value ~)`	prohibit
`(id ^operator value -)`	reject
`(id ^operator value > value2)`	better
`(id ^operator value < value2)`	worse
`(id ^operator value >)`	best
`(id ^operator value <)`	worst
`(id ^operator value =)`	unary indifferent
`(id ^operator value = value2)`	binary indifferent
`(id ^operator value = number)`	numeric indifferent

The identifier and value will always be variables, such as `(<s1> ^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> ^operator <op1>)` is assumed to mean `(<s> ^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 `ctrl-c` 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 described on page **??**.

```
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 Language (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 automatically 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`.

```
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))) }
```

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)
```

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 producttion

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

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

3.3.6.12 Controlling learning

Soar's learning mechanism, called 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 121.) 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 22.)

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.

^`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 75 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 <s> ^superstate nil)
   (state <ss> ^impasse constraint-failure ^superstate <s>)
   -->
   (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

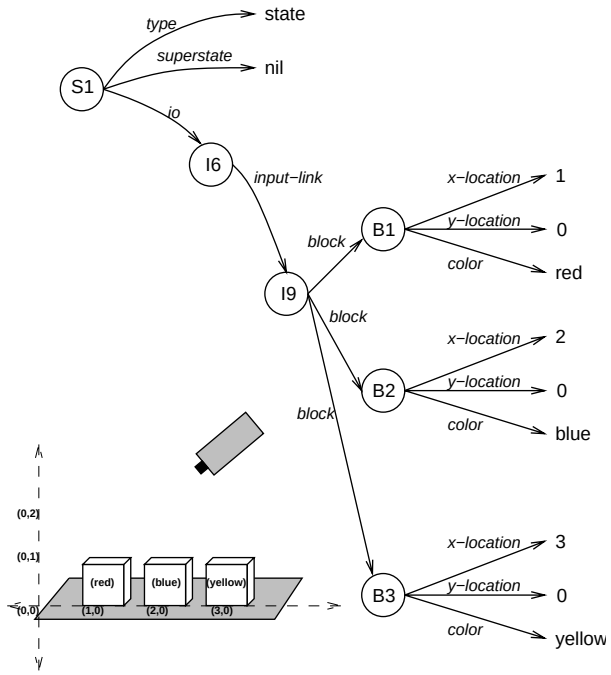

Figure 3.5: An example portion of the input link for the blocks-world task.

---

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 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 2.4.

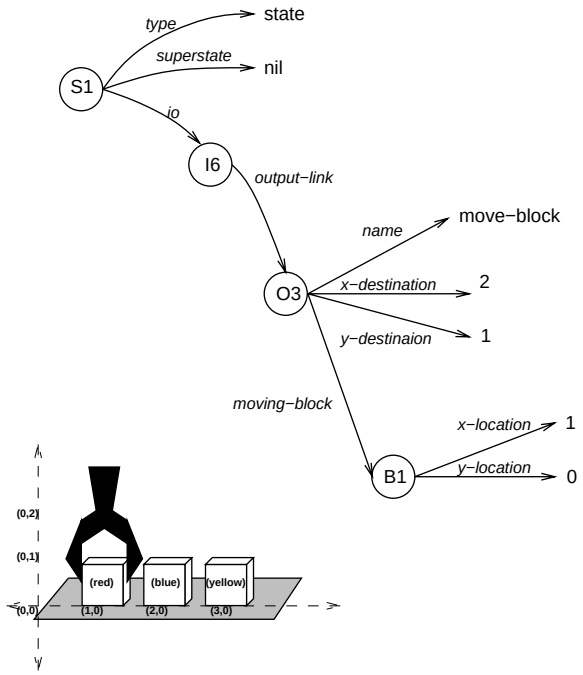

Figure 3.7: An example portion of the output link for the blocks-world task.

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:

```
(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

Learning

Chunking is Soar's learning mechanism, the sole learning mechanism in Soar. 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 4.3 through 4.4 examine the construction of chunks in further detail. Section 4.5 explains how and why chunks are prevented from matching with the WME's that led to their creation. Section 4.6 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 5.4.4 on page 121 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 5.4.4 on page 121 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 "bot-

tom" 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.[1]

3. 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.

4. The augmentation, ^`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 4.6 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 ^`quiescence t` as a condition.

5. 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 the conditions of the productions, finding

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

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 ^item augmentation lists all values that lead to the impasse. Chunking is complicated by the fact that the `^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 `^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 4.6 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 5.3.6 on page 110). This prints out a trace of the conditions as they are collected.

Certain productions do not participate in backtracing. If a production creates only a `reject` preference or a desirability preference (`better`, `worse`, `indifferent`, or `parallel`), then neither the preference nor the objects that led to its creation will be included in the chunk. (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.) 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. The argument is that such preferences should affect only the *efficiency* and not the *correctness* of problem solving, and therefore are not necessary to produce the results. Necessity preferences (`require` or `prohibit`) should be used to enforce the correctness of problem solving; the productions that create these preferences will be included in backtracing.

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.

4.3 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.

4.4 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.

4.5 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.

4.6 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 three known problems in chunking.

4.6.1 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 76). 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, necessity preferences (`require` and `prohibit`) should be used whenever a control decision is being made that also incorporates goal-attainment knowledge. The necessity preferences are included in the backtrace by chunking, thereby avoiding overgenerality.

4.6.2 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.

4.6.3 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 ^`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 ^`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.

Chapter 5

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. Commands are then described fully, in alphabetical order.

Throughout this chapter, each function description includes a specification of its syntax and an example of its use.

For a concise overview of the Soar interface functions, see the Function Summary and Index on page 175. 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.

5.1 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.

help - Provide formatted, on-line information about Soar commands.

init-soar - Reinitialize Soar so a program can be rerun from scratch.

quit - Close log file, terminate Soar, and return user to the operating system.

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.

5.1.1 `excise`

Delete Soar productions from production memory.

Synopsis

```
excise production_name [production_name ...]
excise -[acdtu]
```

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
-t, –task	Remove chunks, justifications, and user productions from memory
-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 init-soar:

```
excise --all
```

Default Aliases

Alias	Maps to
ex	excise

See Also

init-soar

5.1.2 help

Provide formatted usage information about Soar commands.

Synopsis

```
help [command_name]
```

Options

command_name	Print usage syntax for the command.

Description

This command prints formatted help for the given command name.

Examples

To see the syntax for the *excise* command:

```
help excise
```

To see what commands help is available for:

```
help
```

Default Aliases

Alias	Maps to
?	help
h	help
man	help

5.1.3 init-soar

Empties working memory and resets run-time statistics.

Synopsis

```
init-soar
```

Options

No options.

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.

Default Aliases

Alias	Maps to
init	init-soar
is	init-soar

See Also

excise

5.1.4 `quit`

Close log file, terminate Soar, and return user to the operating system.

Synopsis

```
quit
```

Options

No options.

Description

This command stops the run, quits the log and closes Soar.

Default Aliases

Alias	Maps to
exit	quit

5.1.5 `run`

Begin Soar's execution cycle.

Synopsis

```
run [count]
run -[d|e|p|o][fs][un] [count]
```

Options

-d, –decision	Run Soar for count decision cycles.
-e, –elaboration	Run Soar for count elaboration cycles.
-f, –forever	Run until halted by problem-solving completion or until stopped by an interrupt.
-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.

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 a control-C). 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 two 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 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 is triggered, the environment will get information about how the "run" was executed. If a "run" was executed with the –**update** option, then the event sends a flag requesting that the environnment 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

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.

Default Aliases

Alias	Maps to
d	run -d 1
e	run -e 1
step	run 1

5.1.6 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 proces. 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 Section 3.3. 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. Section 3.3 gives an elaborate discussion of the design and coding of productions.

If the name of the new production is the same as an existing one, the old production will be overwritten (excised).

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)
```

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

5.1.7 stop-soar

Pause Soar.

Synopsis

```
stop-soar [-s] [reason string]
```

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).

Default Aliases

Alias	Maps to
interrupt	stop-soar
ss	stop-soar
stop	stop-soar

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.

5.2 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 WME's.

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.

5.2.1 `default-wme-depth`

Set the level of detail used to print WMEs.

Synopsis

```
default-wme-depth [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** command or **wmes** alias). 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** or **wmes** 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.

Default Aliases

Alias	Maps to
set-default-depth	default-wme-depth

See Also

print

5.2.2 `gds-print`

Print the WMEs in the goal dependency set for each goal.

Synopsis

```
gds-print
```

Options

No options.

Description

The Goal Dependency Set (GDS) is described in Appendix E. 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.

Default Aliases

Alias	Maps to
gds_print	gds-print

5.2.3 `internal-symbols`

Print information about the Soar symbol table.

Synopsis

```
internal-symbols
```

Options

No options.

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>
```

5.2.4 matches

Prints information about partial matches and the match set.

Synopsis

```
matches [-nc0t1w2] production_name
matches -[a|r] [-nc0t1w2]
```

Options

production_name	Print partial match information for the named production.
-0, -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.
-1, -t, –timetags	Also print the timetags of the wmes at the first failing condition
-2, -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 **1** and **2** (or –**timetags** and –**wmes** arguments) 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
```

5.2.5 memories

Print memory usage for partial matches.

Synopsis

```
memories [-cdju] [n]
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.
n	Number of productions to print, sorted by those that use the most memory.

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 n is given, only n productions will be printed: the n 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

5.2.6 `preferences`

Examine details about the preferences that support the specified *id* and *attribute*

Synopsis

```
preferences [-0123nNtw] [id] [[^]attribute]
```

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 indicated productions
-3, -w, –wmes	Print the information for the –timetags option above plus the entire wme.
id	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 id and attribute. If *id* and *attribute* are not specified, they default to the current state and the current operator. The '^' 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.

Note

For the time being, **numeric-indifferent** preferences are listed under the heading "binary indifferents:".

Examples

This example prints the preferences on (S1 ^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 ^jug attribute:

```
soar> preferences s1 jug
Preferences for S1 ^jug:
acceptables:
 I5  +:O
 J1  +:O
```

Default Aliases

Alias	Maps to
pr	preferences

5.2.7 `print`

Print items in working memory or production memory.

Synopsis

```
print [-fFin] production_name
print -[a|c|D|j|u][fFin]
print [-i] [-d <depth>] identifier | timetag | pattern
print -s[oS]
```

Options

Printing items in production memory

-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
-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.
-j, –justifications	print the names of all justifications currently loaded.
-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.
-u, –user	print the names of all user productions currently loaded
production_name	print the production named production-name

Printing items in working memory

-d, –depth *n*	This option overrides the default printing depth (see the default-wme-depth command for more detail).
-i, –internal	items should be printed in their internal form. For working memory, this means printing the individual elements with their timetags, rather than the objects.
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 [+])
```

The pattern is surrounded by parentheses. The *identifier* , *attribute* , and *value* must be valid Soar symbols or the wildcard symbol * which matches all occurences. The optional + symbol restricts pattern matches to acceptable preferences.

Examples

Print the working memory elements (and their timetags) which have the identifier s1 as object and v2 as value:

```
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 prodname
```

Print the names of all user productions currently loaded:

```
print -u
```

Default Aliases

Alias	Maps to
p	print
pc	print –chunks
wmes	print -i

See Also

default-wme-depth predefined-aliases

5.2.8 production-find

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 *)}
```

See Also

sp

5.3 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.

warnings - Toggle whether or not warnings are printed.

watch - Control the information printed as Soar runs.

watch-wmes - Trace WMEs matching specific patterns.

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 output.

5.3.1 `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 [<prefix>]	If <prefix> is given, use <prefix> as the prefix for naming chunks. Otherwise, return the current *prefix* . (defaults to "**chunk**")
-c, –count [<count>]	If <count> is given, set the chunk counter for naming chunks to <count>. Otherwise, return the current value of the chunk counter.

Description

The short format for naming newly-created chunks is: *prefixChunknum*. The long (default) format for naming chunks is: *prefix-Chunknum**d*dc***impassetype***dcChunknum* where: *prefix* is a user-definable prefix string; *prefix* defaults to "**chunk**" when unspecified by the user. It may not contain the character *. *Chunknum* is <count> for the first chunk created, <count>+1 for the second chunk created, etc. *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.

5.3.2 `firing-counts`

Print the number of times each production has fired.

Synopsis

```
firing-counts [n]
firing-counts production_names
```

Options

If given, an option can take one of two forms – an integer or a list of production names:

n	List the top *n* productions. If *n* is 0, only the productions which haven't fired are listed
production_name	For each production in production_names, 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 one or more production names are given as arguments, only firing counts for these productions are printed. Note that firing counts are reset by a call to **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 productions my*first*production and my*second*production:

```
firing-counts my*first*production my*second*production
```

Warnings

Firing-counts are reset to zero after an init-soar.
NB: This command is slow, because the sorting takes time O(n*log n)

Default Aliases

Alias	Maps to
fc	firing-counts

See Also

init-soar

5.3.3 pwatch

Trace firings and retractions of specific productions.

Synopsis

```
pwatch [-d|e] [production name]
```

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, **pwatch** –**enable** lists all productions currently being traced, and **pwatch** –**disable** disables tracing of all productions. Note that **pwatch** now only takes one production per command. Use multiple times to watch multiple functions.

Default Aliases

Alias	Maps to
pw	pwatch

See Also

watch

5.3.4 `stats`

Print information on Soar's runtime statistics.

Synopsis

Structured Output

`stats`

Raw Output

```
stats [-s|-m|-r]
```

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. This is the default if no args are specified.

Description

This command prints Soar internal statistics. The argument indicates the component of interest.

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.

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 everage 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 optional **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 optional **stats** argument –**rete** provides information about node usage in the Rete net, the large data structure used for efficient matching in Soar.

Default Aliases

Alias	Maps to
st	stats

See Also

timers

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.

5.3.5 `warnings`

Synopsis

```
warnings -[e|d]
```

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.

5.3.6 `watch`

Control the run-time tracing of Soar.

Synopsis

```
watch
watch [--level] [0|1|2|3|4|5]
watch -N
watch -[dpPwrDujcbi] [<remove>] -[n|t|f]
watch --learning [<print|noprint|fullprint>]
```

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
-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 and decisions; equivalent to -dp
3	watch productions, phases, and decisions; equivalent to -dpP
4	watch wmes, productions, phases, and decisions; equivalent to -dpPw
5	watch preferences, wmes, productions, phases, and decisions; equivalent to -dpPwr

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

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	*Argument to Option*	*Description*
-n, –nowmes	No argument	When watching productions, do not print any information about matching wmes
-t, –timetags	No argument	When watching productions, print only the timetags for matching wmes
-f, –fullwmes	No argument	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 **watch** –**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*
-b, –backtracing	remove (optional)	Controls the printing of backtracing information when a chunk or justification is created
-i, –indifferent-selection	remove (optional)	Controls the printing of the scores for tied operators in random indifferent selection mode

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 **watch *level*** is **1** , (or –**decisions**). Each level can be indicated with either a number or a series of flags as follows:

```
0 or --none
1 or --decisions
2 or --decisions --phases
3 or --decisions --phases --productions
4 or --decisions --phases --productions --wmes
5 or --decisions --phases --productions --wmes --preferences
```

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 "watch –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, but try to 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 setting)
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 setting)
watch 2 -r    (... OK, but try to 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 --fullwmes
watch -d -p -P -f -u remove -j 0
watch -f -l 3 -u 0 -j 0
```

Default Aliases

Alias	Maps to
w	watch

See Also

pwatch print run watch-wmes

5.3.7 watch-wmes

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 *
```

5.4 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

attribute-preferences-mode In Soar7 mode, controls handling of preferences for non-context slots.

explain-backtraces - Print information about chunk and justification backtraces.

indifferent-selection - Controls how indifferent selections are made.

learn - Set the parameters for chunking, Soar's learning mechanism.

max-chunks - Limit the number of chunks created during a decision cycle.

max-elaborations - Limit the maximum number of elaboration cycles.

max-nil-output-cycles

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.

save-backtraces - Save trace information to explain chunks and justifications.

soar8 - Toggle between Soar 8 methodology and Soar 7 methodology.

timers - Toggle on or off the internal timers used to profile Soar.

waitsnc - Generate a wait state rather than a state-no-change impasse.

5.4.1 `attribute-preferences-mode`

For Soar 7, this command sets and prints the attributes preferences mode to control the handling of preferences (other than acceptable and reject preferences) for non-context slots.

Synopsis

```
attribute-preferences-mode [0|1|2]
```

Options

0	Handle preferences the normal (Soar 6) way.
1	Handle preferences the normal (Soar 6) way, but print a warning message whenever a preference other than + or - is created for a noncontext slot.
2	When a preference other than + or - created for a non-context slot, print an error message and ignore (discard) that preference. For non-context slots, the set of values installed in working memory is always equal to the set of acceptable values minus the set of rejected values.

Description

For Soar 7, this command sets and prints the attributes preferences mode to control the handling of preferences (other than acceptable and reject preferences) for non-context slots. The command issued with no arguments, returns the current mode. This command is obsolete for Soar 8. In Soar 8, the code automatically operates as if attribute-preferences-mode = 2.

5.4.2 `explain-backtraces`

Print information about chunk and justification backtraces.

Synopsis

```
explain-backtraces -f prod_name
explain-backtraces [-c <n>] prod_name
```

Options

(no args)	List all productions that can be "explained"
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, **explain-backtraces -c n prodname** , 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.

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.
```

Default Aliases

Alias	Maps to
eb	explain-backtraces

See Also

save-backtraces

5.4.3 indifferent-selection

Controls indifferent preference arbitration.

Synopsis

```
indifferent-selection [-aflr]
```

Options

-a, –ask	Ask the user to choose. **Not implemented.**
-f, –first	Select the first indifferent object from Soar's internal list.
-l, –last	Select the last indifferent object from Soar's internal list.
-r, –random	Select randomly (default).

Description

The **indifferent-selection** command allows the user to set which option should be used to select between operator proposals that are mutally indifferent in preference memory. The default option is –**random** which chooses an operator at random from the set of mutually indifferent proposals, with the selection biased by any existing numeric preferences. For repeatable results, the user may choose the –**first** or –**last** option. "First" refers to the list of operator augmentations internal to Soar; the ordering of the augmentations is arbitrary but deterministic, so that if you run Soar repeatedly, –**first** will always make the same decision. Similarly, –**last** chooses the last of the tied objects from the internal list. For complete control over the decision process, the –**ask** option prompts the user to select the next operator from a list of the tied operators. If no argument is provided, **indifferent-selection** will display the current setting.

Default Aliases

Alias	Maps to
inds	indifferent-selection

See Also

numeric-indifference-mode

5.4.4 `learn`

Set the parameters for chunking, Soar's learning mechanism.

Synopsis

```
learn [-l]
learn -[d|E|o]
learn -e [ab]
```

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.

Description

The learn command controls the parameters for chunking (Soar's learning mechanism). 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 –**all-levels** and –**bottom-up** flags are orthogonal to the –**on** , –**except** , –**only** , and –**off** flags, and so, may be used in combination with them. 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. 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
```

Default Aliases

Alias	Maps to
l	learn

See Also

watch explain-backtraces save-backtraces

5.4.5 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).

5.4.6 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* . In Soar8, *max-elaborations* is checked during both the Propose Phase and the Apply Phase. If Soar8 runs more than the max-elaborations limit in either of these phases, Soar8 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
```

5.4.7 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

5.4.8 multi-attributes

Declare a symbol to be multi-attributed.

Synopsis

```
multi-attributes [symbol [n]]
```

Options

symbol	Any Soar attribute.
n	Integer > 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 (>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
```

5.4.9 numeric-indifferent-mode

Select method for combining numeric preferences.

Synopsis

```
numeric-indifferent-mode [-as]
```

Options

-a, –avg, –average	Use average mode (default).
-s, –sum	Use sum mode.

Description

The numeric-indifferent-mode command is used to select the method for combining numeric preferences. This command is only meaningful in indifferent-selection –random mode. The default procedure is –**avg** (average) which assigns a final value to an operator according to the rule:

- If the operator has at least one numeric preference, assign it the value that is the average of all of its numeric preferences.
- If the operator has no numeric preferences (but has been included in the indifferent selection through some combination of non-numeric preferences), assign it the value 50.

The intended range of numeric-preference values for –**avg** mode is 0-100. The other combination option –**sum** assigns a final value according to the rule:

- Add together any numeric preferences for the operator (defaulting to 0 if there are none).
- Assign the operator the value e^{PreferenceSum / AgentTemperature}, where AgentTemperature is a compile-time constant currently set at 25.0.

Any real-numbered preference may be used in –**sum** mode. Once a value has been computed for each operator, the next operator is selected probabilistically, with each candidate operator's chance weighted by its computed value.

5.4.10 `o-support-mode`

Choose experimental variations of o-support.

Synopsis

```
o-support-mode [0|1|2|3|4]
```

Options

0	Mode 0 is the base mode. O-support is calculated based on the structure of working memory that is tested and modified. Testing an operator or operator acceptable preference results in state or operator augmentations being o-supported. The support computation is very complex (see soar manual).
1	Not available through gSKI.
2	Mode 2 is the same as mode 0 except that all support is calculated the production structure, not from working memory structure. Augmentations of operators are still o-supported.
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.
4	Mode 4 is the default.

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 can be considered current to Soar8, 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 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.

5.4.11 `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 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

5.4.12 `soar8`

Toggle between Soar 8 methodology and Soar 7 methodology.

Synopsis

```
soar8 [-ed]
```

Options

-e, –enable, –on	Use Soar 8 methodology. (Default)
-d, –disable, –off	Use Soar 7 methodology.

Description

The soar8 command allows users to revert to Soar 7 methodology in order to run older Soar programs. Both production memory and working memory must be empty to toggle between Soar 7 and Soar 8 mode. The soar8 command with no arguments returns the current mode, the default is Soar 8. Users can toggle between modes ONLY when production memory and working memory are both empty. This means that users must either change the mode at startup before any productions are loaded, or must first issue "excise -all" (which does an "init-soar" as well) before changing modes. Note that there are differences in the preference mechanism and in operator termination (among other things) between Soar 8 and Soar 7. Users should read the Soar 8.2 Release Notes for more details.

Warnings

Production memory and working memory must be empty to switch between modes.

5.4.13 `timers`

Toggle on or off the internal timers used to profile Soar.

Synopsis

```
timers [-ed]
```

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

5.4.14 `waitsnc`

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.

5.5 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.

dirs - List the directory stack.

echo - Print a string to the current output device.

log - Record all user-interface input and output to a file.

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** .

5.5.1 cd

Change directory.

Synopsis

```
cd [directory]
```

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
```

Default Aliases

Alias	Maps to
chdir	cd

See Also

dirs home ls pushd popd source topd

5.5.2 `dirs`

List the directory stack

Synopsis

```
dirs
```

Options

No options.

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 cd's to it. The command **popd** removes the directory at the top of the directory stack and cd's to the previous directory which now appears at the top of the stack.

See Also

cd home ls pushd popd source topd

5.5.3 `echo`

Print a string to the current output device.

Synopsis

```
echo string
```

Options

string	The string to print.

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

log

5.5.4 log

Record all user-interface input and output to a file.

Synopsis

```
log [-Ae] filename
log -a  string
log [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 **log** 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 **log** 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 (see the home command)). Use the **close** option to stop logging.

Examples

To initiate logging and place the record in foo.log:

```
log foo.log
```

To append log data to an existing foo.log file:

```
log -A foo.log
```

To terminate logging and close the open log file:

```
log -c
```

Known Issues

Does not log everything when structured output is selected.

5.5.5 `ls`

List the contents of the current working directory.

Synopsis

```
ls
```

Options

No options.

Description

List the contents of the working directory.

Default Aliases

Alias	Maps to
dir	ls

See Also

cd dirs home pushd popd source topd

5.5.6 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
```

Options

No options.

Description

This command pops a directory off of the directory stack and cd's to it. See the dirs command for an explanation of the directory stack.

See Also

cd dirs home ls pushd source topd

5.5.7 pushd

Push a directory onto the directory stack, changing to it.

Synopsis

```
pushd directory
```

Options

directory	Directory to change to, saving the current directory on to the stack.

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 home ls popd source topd

5.5.8 pwd

Print the current working directory.

Synopsis

```
pwd
```

Options

No options.

Description

Prints the current working directory of Soar.

Default Aliases

Alias	Maps to
topd	pwd

5.5.9 rete-net

Save the current Rete net, or restore a previous one.

Synopsis

```
rete-net -s|l filename
```

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 -a
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.

Default Aliases

Alias	Maps to
rn	rete-net

See Also

excise init-soar

5.5.10 `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

5.5.11 `source`

Load and evaluate the contents of a file.

Synopsis

```
source filename
```

Options

filename	The file of Soar productions and commands to load.

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.

See Also

cd dirs home ls pushd popd topd

5.6 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. output links.

remove-wme - Manually remove an element from working memory.

These commands are used only when Soar needs to interact with an external environment.

5.6.1 add-wme

Manually add an element to working memory.

Synopsis

```
add-wme id [^]attribute value [+]
```

Options

id	Must be an existing identifier.
^	Leading ^ 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 "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 ^ is optional, it has been left off in this case.

```
add-wme Z2 message * +
```

Default Aliases

Alias	Maps to
aw	add-wme

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

5.6.2 `remove-wme`

Manually remove an element from working memory.

Synopsis

```
remove-wme timetag
```

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.

Default Aliases

Alias	Maps to
rw	remove-wme

See Also

add-wme

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.

5.7 Miscellaneous

The specific commands described in this section are:

Summary

alias - Define command aliases.

soarnews - Prints information about the current release.

time - Uses a default system clock timer to record the wall time required while executing a command.

version - Returns version number of Soar kernel.

5.7.1 alias

Define a new alias, or command, using existing commands and arguments.

Synopsis

```
alias name [cmd <args>]
alias -d name
alias
```

Options

-d, –disable, –off	Remove the named alias.
name	The name of the alias, i.e. the new command.
cmd	An existing command that will be invoked when the alias is entered on the commandline.
args	Valid arguments to the cmd (optional & optional number).

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 **-d** flag. 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
```

Default Aliases

Alias	Maps to
a	alias
un	alias -d
unalias	alias -d

See Also

unalias

5.7.2 soarnews

Prints information about the current release.

Synopsis

```
soarnews
```

Default Aliases

Alias	Maps to
sn	soarnews

5.7.3 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.

5.7.4 `version`

Synopsis

```
version
```

Options

No options

Description

This command gives version information about the current Soar kernel. It returns the version number itself, 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.
```

```
#
# 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 127). 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 production 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 ^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-

[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.

```
sp {elaborate*state*operator*name
  (state <s> ^superstate <s1>)
  (<s1> ^operator <o>)
  (<o> ^name <name>)
-->
  (<s> ^name something)}
```

the RHS action gets i-support. Of course, the state bound to `<s>` is destroyed when `(<s1> ^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> ^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.2. (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.2 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 22).

- 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.

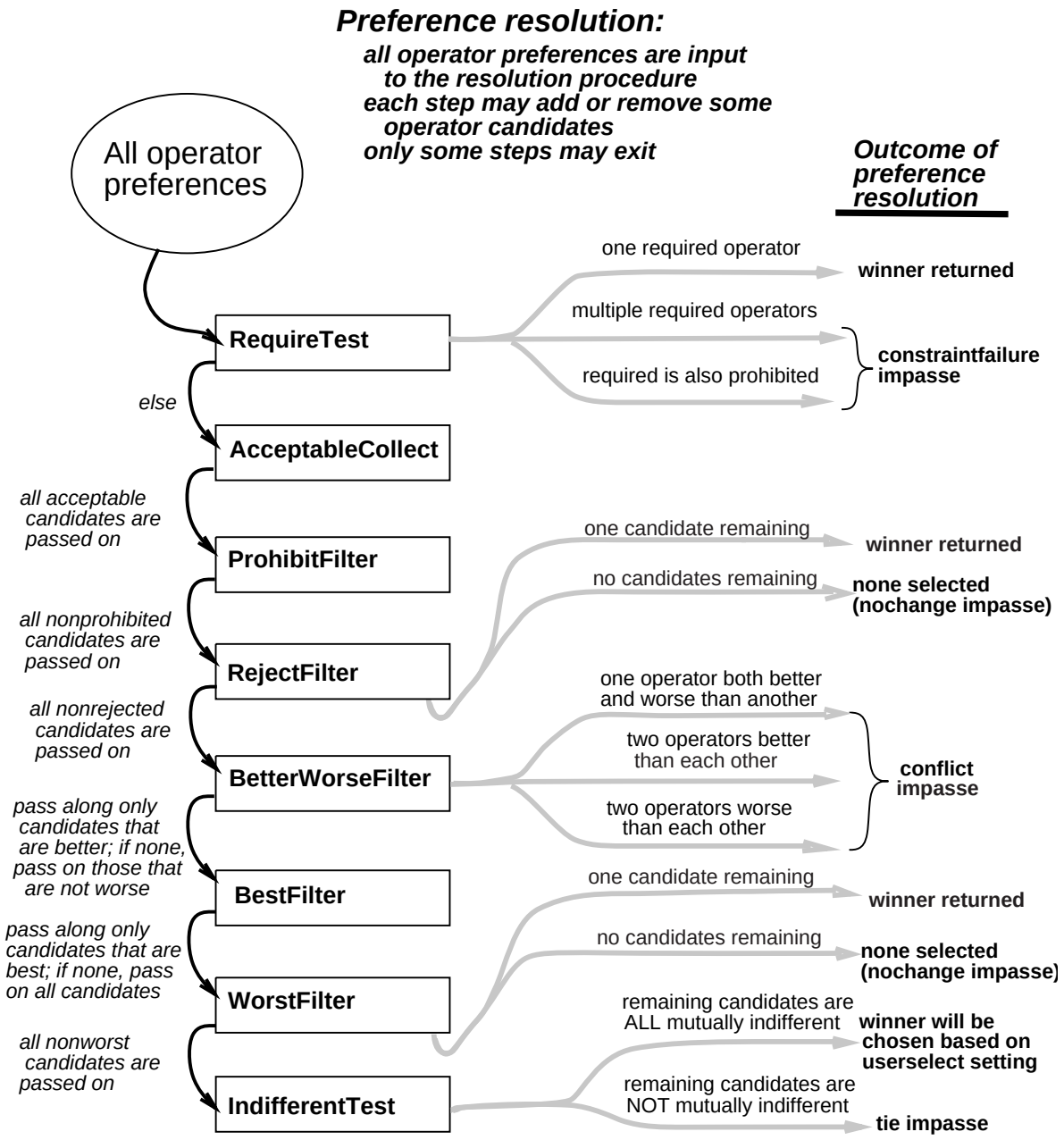

Figure D.2: An illustration of the preference resolution process. There are eight steps; only five of these provide exits from the resolution process.

- 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.
- 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 (∼) 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.

- 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 checks for better worse conflicts, and otherwise filters out candidates based on better and worse preferences.

- A better/worse conflict occurs when one candidate has both better and worse preferences with respect to another operator (i.e., $\mathbf{A} < \mathbf{B}$ & $\mathbf{B} < \mathbf{A}$). Since preferences are not transitive, the situation $\mathbf{A} < \mathbf{B} < \mathbf{C} < \mathbf{A}$ is not a conflict. If there are better/worse conflicts, preference semantics terminates by declaring a conflict impasse and returning the set of conflicted items.
- Otherwise — Filter out of the candidates the ones that have another candidate that is better, or are worse than another candidate. The resulting 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.

WorstFilter (<) If all remaining candidates have worst preferences, this filter has no effect. Otherwise, the filter removes any candidates that have a worst preference.

- Once again, 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 IndifferentTest.

IndifferentTest (=) 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
- the candidate is binary indifferent to all of the remaining candidate operators

If some candidate is left unmarked, then the procedure signals a tie impasse and returns the complete set of candidates that passed into the IndifferentTest. 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 120.

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 (1998) 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[2]. In Soar, any action (application) of an operator receives an o-support

[1] A preliminary draft by Robert Wray, contact at `wrayre@acm.org`.

[2] 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**

preference. This preference 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.

Dependency Set" a specific example, even though "goals" have not been an explicit, behavior-level Soar construct since Soar 6

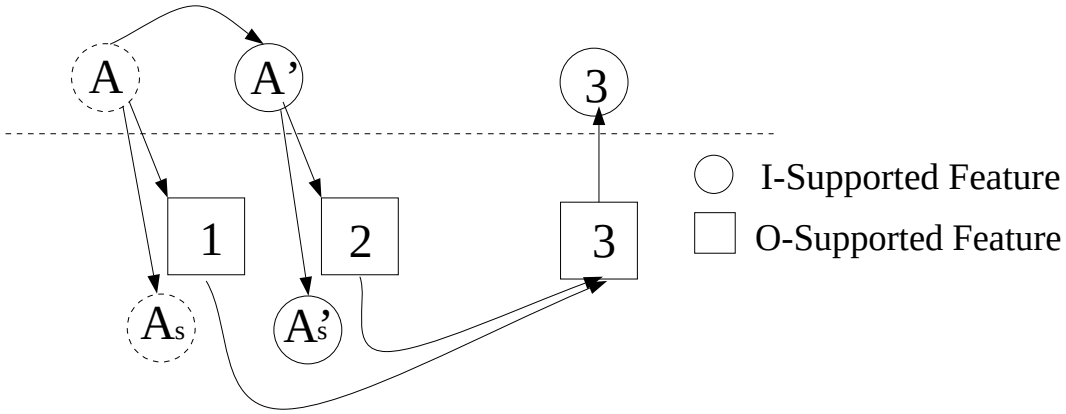

Figure E.2: Simplified Representation of the context dependencies (above the line), local os-upported WMEs (below the line), and the generation of a result. In Soar 7, this situation led to non-contemporaneous constraints in the chunk that generates **3**.

Behavior-level view of the Goal Dependency Set

This section discusses what the GDS does, and how that impacts production knowledge design and implementation.

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.[3] I-support is illustrated in Figure E.2. A copy of **A** in the subgoal, $\mathbf{A}_s$, is retracted automatically when **A**

[3]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.

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.2, the change from **A** to **A'** is a 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.[4] 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.4, 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.4, the persistent subgoal item **3** is dependent

[4]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.

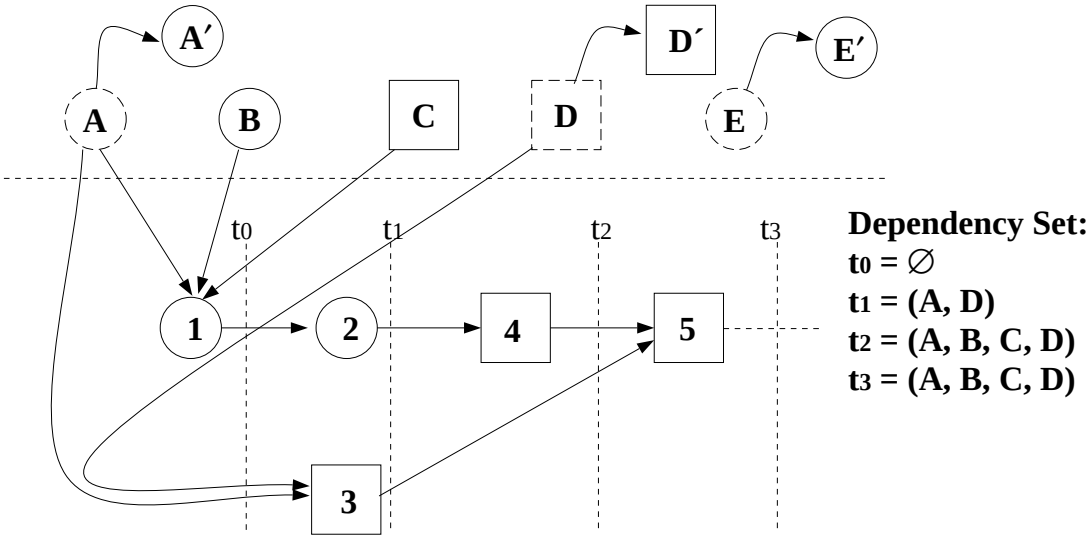

Figure E.4: The Dependency Set in Soar 8.

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.[5]

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 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.4, 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.

[5]In addition, superstate WMEs can also include context slot preferences, which are represented in the architecture as working memory elements.

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 the GDS via elaborate_gds(), a procedure in decide.c that mimics the chunking backtrace function. The algorithm is shown in Figure E.5. 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.6 illustrates the relationship. Each GDS consists of a pointer to its goal and a pointer to a WME DLL list. The gds_next and gds_prev pointers on WME 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.

```
PROC create_new_assertion(...)
    Whenever a new o-supported element is asserted, the GDS is updated
    to include any new context dependencies.
    ...
    A_inst ← 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 ← append(G_GDS, elaborate_GDS(A))
    ...
END

PROC elaborate_GDS(assertion A)
    S ← {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 ← append(S, elaborate_GDS(c))
                (compute GDS dependencies for c and add to goal's GDS)
④           c_inspected ← 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.5: The algorithm for determining members of the GDS.

INSERT DIAGRAM HERE

Figure E.6: The GDS and WME data structures

.

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.)

Colophon

This document was produced on a Sun workstation using LaTeX2$_\epsilon$. Illustrations were created using idraw.

Index

Summary of Soar Aliases and Functions

Predefined Aliases

There are a number of Soar "commands" that are shorthand for other Soar commands:

Summary of Soar Functions

The following table lists the commands in Soar. See the referenced page number for a complete description of each command.