

The Soar User's Manual

Version 9.6.0

John E. Laird, Clare Bates Congdon,
Mazin Assanie, Nate Derbinsky and Joseph Xu

Additional contributions by:

Mitchell Bloch, Karen J. Coulter, Steven Jones,
Aaron Mininger, Preeti Ramaraj and Bryan Stearns

Division of Computer Science and Engineering
University of Michigan

Draft of: September 21, 2017

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

Copyright © 1998 - 2017, The Regents of the University of Michigan

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

Contents	vii
1 Introduction	1
1.1 Using this Manual	2
1.2 Contacting the Soar Group	3
1.3 Different Platforms and Operating Systems	4
2 The Soar Architecture	5
2.1 An Overview of Soar	5
2.1.1 Types of Procedural Knowledge in Soar	6
2.1.2 Problem-Solving Functions in Soar	7
2.1.3 An Example Task: The Blocks-World	7
2.1.4 Representation of States, Operators, and Goals	8
2.1.5 Proposing candidate operators	9
2.1.6 Comparing candidate operators: Preferences	9
2.1.7 Selecting a single operator: Decision	11
2.1.8 Applying the operator	11
2.1.9 Making inferences about the state	12
2.1.10 Problem Spaces	12
2.2 Working memory: The Current Situation	14
2.3 Production Memory:	
Long-term Procedural Knowledge	16
2.3.1 The structure of a production	17
2.3.2 Architectural roles of productions	18
2.3.3 Production Actions and Persistence	18
2.4 Preference Memory: Selection Knowledge	19
2.4.1 Preference Semantics	19
2.4.2 How preferences are evaluated to decide an operator	21
2.5 Soar’s Execution Cycle: Without Substates	24
2.6 Input and Output	25
2.7 Impasses and Substates	27
2.7.1 Impasse Types	28
2.7.2 Creating New States	28
2.7.3 Results	29
2.7.4 Justifications: Support for results	31
2.7.5 Chunking: Learning Procedural Knowledge	32

2.7.6	The calculation of o-support	32
2.7.7	Removal of Substates: Impasse Resolution	34
2.7.8	Soar’s Cycle: With Substates	36
2.7.9	Removal of Substates: The Goal Dependency Set	37
3	The Syntax of Soar Programs	43
3.1	Working Memory	43
3.1.1	Symbols	44
3.1.2	Objects	44
3.1.3	Timetags	45
3.1.4	Acceptable preferences in working memory	46
3.1.5	Working Memory as a Graph	46
3.1.6	Working Memory Activation	48
3.2	Preference Memory	48
3.3	Production Memory	48
3.3.1	Production Names	50
3.3.2	Documentation string (optional)	50
3.3.3	Production type (optional)	50
3.3.4	Comments (optional)	51
3.3.5	The condition side of productions (or LHS)	52
3.3.6	The action side of productions (or RHS)	67
3.3.7	Grammars for production syntax	82
3.4	Impasses in Working Memory and in Productions	84
3.4.1	Impasses in working memory	84
3.4.2	Testing for impasses in productions	86
3.5	Soar I/O: Input and Output in Soar	86
3.5.1	Overview of Soar I/O	86
3.5.2	Input and output in working memory	87
3.5.3	Input and output in production memory	89
4	Procedural Knowledge Learning	91
4.1	Chunking	91
4.2	Explanation-based Chunking	92
4.3	Overview of the EBC Algorithm	94
4.3.1	Identity	95
4.3.2	The Five Main Components of Explanation-Based Chunking	98
4.4	What EBC Does Prior to the Learning Episode	99
4.4.1	Identity Assignment and Propagation	99
4.4.2	Relevant Operator Selection Knowledge Tracking	101
4.5	What EBC Does During the Learning Episode	102
4.5.1	Calculating the Complete Set of Results	102
4.5.2	Backtracing and the Three Types of Analysis Performed	103
4.5.3	Rule Formation	105
4.6	Subtleties of EBC	108
4.6.1	Relationship Between Chunks and Justifications	108
4.6.2	Chunk Inhibition	108

4.6.3	Chunks Based on Chunks	109
4.6.4	Mixing Chunks and Justifications	109
4.6.5	Generality and Correctness of Learned Rules	109
4.6.6	Over-specialization and Over-generalization	110
4.6.7	Previous Results and Rule Repair	110
4.6.8	Missing Operator Selection Knowledge	111
4.6.9	Generalizing Over Operators Selected Probabilistically	111
4.6.10	Collapsed Negative Reasoning	112
4.6.11	Problem-Solving That Doesn't Test The Superstate	112
4.6.12	Disjunctive Context Conflation	113
4.6.13	Generalizing knowledge retrieved from semantic or episodic memory	113
4.6.14	Learning from Instruction	114
4.6.15	Determining Which OSK Preferences are Relevant	115
4.6.16	Generalizing Knowledge From Math and Other Right-Hand Side Functions	116
4.6.17	Situations in which a Chunk is Not Learned	117
4.7	Usage	118
4.7.1	Overview of the <code>chunk</code> command	118
4.7.2	Enabling Procedural Learning	119
4.7.3	Fine-tuning What Your Agent Learns	119
4.7.4	Examining What Was Learned	120
4.8	Explaining Learned Procedural Knowledge	123
4.9	Visualizing the Explanation	128
5	Reinforcement Learning	131
5.1	RL Rules	131
5.2	Reward Representation	133
5.3	Updating RL Rule Values	134
5.3.1	Gaps in Rule Coverage	136
5.3.2	RL and Substates	137
5.3.3	Eligibility Traces	138
5.3.4	$GQ(\lambda)$	139
5.4	Automatic Generation of RL Rules	139
5.4.1	The <code>gp</code> Command	139
5.4.2	Rule Templates	140
5.4.3	Chunking	141
6	Semantic Memory	143
6.1	Working Memory Structure	143
6.2	Knowledge Representation	144
6.2.1	Integrating Long-Term Identifiers with Soar	144
6.3	Storing Semantic Knowledge	145
6.3.1	<code>Store</code> command	145
6.3.2	<code>Store-new</code> command	146
6.3.3	User-Initiated Storage	146

6.3.4	Storage Location	147
6.4	Retrieving Semantic Knowledge	147
6.4.1	Non-Cue-Based Retrievals	148
6.4.2	Cue-Based Retrievals	148
6.4.3	Retrieval with Depth	152
6.5	Performance	153
6.5.1	Math queries	153
6.5.2	Performance Tweaking	153
7	Episodic Memory	155
7.1	Working Memory Structure	155
7.2	Episodic Storage	156
7.2.1	Episode Contents	156
7.2.2	Storage Location	156
7.3	Retrieving Episodes	157
7.3.1	Cue-Based Retrievals	158
7.3.2	Absolute Non-Cue-Based Retrieval	159
7.3.3	Relative Non-Cue-Based Retrieval	160
7.3.4	Retrieval Meta-Data	160
7.4	Performance	161
7.4.1	Performance Tweaking	162
8	Spatial Visual System	165
8.1	The scene graph	166
8.1.1	svs_viewer	168
8.2	Scene Graph Edit Language	168
8.2.1	Examples	169
8.3	Commands	169
8.3.1	add_node	170
8.3.2	copy_node	170
8.3.3	delete_node	171
8.3.4	set_transform	171
8.3.5	set_tag	172
8.3.6	delete_tag	172
8.3.7	extract and extract_once	172
8.4	Filters	173
8.4.1	Result lists	174
8.4.2	Filter List	174
8.4.3	Examples	176
8.5	Writing new filters	177
8.5.1	Filter subclasses	177
8.5.2	Generic Node Filters	179
8.6	Command line interface	181
9	The Soar User Interface	183
9.1	Basic Commands for Running Soar	184

9.1.1	soar	185
9.1.2	run	191
9.1.3	exit	193
9.1.4	help	193
9.1.5	decide	194
9.1.6	alias	198
9.2	Procedural Memory Commands	199
9.2.1	sp	200
9.2.2	gp	202
9.2.3	production	204
9.3	Short-term Memory Commands	214
9.3.1	print	215
9.3.2	wm	219
9.3.3	preferences	226
9.3.4	svs	229
9.4	Learning	231
9.4.1	chunk	232
9.4.2	rl	236
9.5	Long-term Declarative Memory	241
9.5.1	smem	241
9.5.2	epmem	251
9.6	Other Debugging Commands	257
9.6.1	trace	258
9.6.2	output	265
9.6.3	explain	269
9.6.4	visualize	275
9.6.5	stats	278
9.6.6	debug	281
9.7	File System I/O Commands	283
9.7.1	File System	284
9.7.2	load	285
9.7.3	save	289
9.7.4	echo	291
Index	293	
Summary of Soar Aliases, Variables, and Functions	299	

List of Figures

2.1	Soar is continually trying to select and apply operators.	5
2.2	The initial state and goal of the “blocks-world” task.	8
2.3	The initial state of the blocks world as working memory objects	9
2.4	The WM state in blocks world after the first operator is selected	10
2.5	Six proposed blocks world operators	10
2.6	The blocks-world problem space	13
2.7	An abstract view of production memory	16
2.8	The preference resolution process	22
2.9	A detailed illustration of Soar’s decision cycle.	26
2.10	A simplified version of the Soar algorithm.	27
2.11	A simplified illustration of a subgoal stack.	30
2.12	Simplified Representation of the context dependencies	39
2.13	The Dependency Set in Soar.	39
3.1	A semantic net illustration of four objects in working memory.	47
3.2	An example production from the example blocks-world task.	49
3.3	An example portion of the input link for the blocks-world task.	88
3.4	An example portion of the output link for the blocks-world task.	89
4.1	A Soar 9.4.0 chunk vs. an explanation-based chunk	92
4.2	A comparison of a working memory trace and an explanation trace	93
4.3	A visualization of an explanation trace	94
4.4	An explanation trace of two simple rules that matched in a substate	96
4.5	An explanation trace after identity analysis	97
4.6	The five main components of explanation-based chunking	98
4.7	The seven stages of rule formation	105
4.8	A colored visualization of an explanation trace	128
5.1	Example Soar substate operator trace.	137
6.1	Example long-term identifier with four augmentations.	144
7.1	Example episodic memory cache setting data.	163
8.1	SVS environment setup	165
8.2	SVS scene graph representation	167

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: version 9.6.0.

Our goals for Soar include that it ultimately 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 possible problem solving methods;
- interact with the outside world; and
- learn about all aspects of the tasks and its performance on those tasks.

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

1. The number of distinct architectural mechanisms should be minimized. Classically Soar had only a single representation of permanent knowledge (production rules), a single representation of temporary knowledge (objects with attributes and values), a single mechanism for generating goals (automatic subgoaling), and a single learning mechanism (chunking). It was only as Soar was applied to diverse tasks in complex environments that we found these mechanisms to be insufficient and added new long-term memories (semantic and episodic) and learning mechanisms (semantic, episodic, and reinforcement learning) to extend Soar agents with crucial new functionalities.
2. All decisions are made through the combination of relevant knowledge at run-time. In Soar, every decision is based on the current interpretation of sensory data and any relevant knowledge retrieved from permanent memory. Decisions are never precompiled into uninterruptible sequences.

1.1 Using this Manual

We expect that novice Soar users will read the manual in the order it is presented. Not all users will make use of the mechanisms described in chapters 4-8, but it is important to know that these capabilities exist.

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

Chapter 4 describes chunking, Soar’s mechanism to learn new procedural knowledge (productions).

Chapter 5 describes reinforcement learning (RL), a mechanism by which Soar’s procedural knowledge is tuned given task experience.

Chapter 6 and **Chapter 7** describe Soar’s long-term declarative memory systems, semantic and episodic.

Chapter 8 describes the Spatial Visual System (SVS), a mechanism by which Soar can convert complex perceptual input into practical semantic knowledge.

Chapter 9 describes the Soar user interface — how the user interacts with Soar. The chapter is a catalog of user-interface commands, grouped by functionality. The most accurate and up-to-date information on the syntax of the Soar User Interface is found online, at the Soar web site, at <https://github.com/SoarGroup/Soar/wiki/CommandIndex>.

Advanced users will refer most often to Chapter 9, flipping back to Chapters 2 and 3 to answer specific questions.

Chapters 2 and 3 make use of a Blocks World example agent. The Soar code for this agent can be downloaded at <https://web.eecs.umich.edu/soar/blocksworld.soar>.

Additional Back Matter

After these chapters is 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. The Soar project website (see link below) has additional help documents and resources.

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

1.2 Contacting the Soar Group

Resources on the Internet

The primary website for Soar is:

<http://soar.eecs.umich.edu/>

Look here for the latest Soar-related downloads, documentation, FAQs, and announcements, as well as links to information about specific Soar research projects and researchers.

Soar kernel development is hosted on GitHub at

<https://github.com/SoarGroup>

This site contains the public GitHub repository, a wiki describing the command-line interface, and an issue tracker where users can report bugs or suggest features.

To contact the Soar group or get help, or to receive notifications of significant developments in Soar, we recommend that you register with one or both of our email lists:

For questions about using Soar, you can use the **soar-help** list. For other discussion or to receive announcements, use the **soar-group** list.

Also, please do not hesitate to file bugs on our issue tracker:

<https://github.com/SoarGroup/Soar/issues>

To avoid redundant entries, please search for duplicate issues first.

For Those Without Internet Access

Mailing Address:

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

1.3 Different Platforms and Operating Systems

Soar runs on a wide variety of platforms, including Linux, Unix (although not heavily tested), Mac OS X, and Windows 10, 7, possibly 8 and Vista, XP, 2000 and NT). We currently test Soar on both 32-bit and 64-bit versions of Ubuntu Linux, OS X 10, and Windows 10.

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

Chapter 2

The Soar Architecture

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

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

2.1 An Overview of Soar

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

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

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

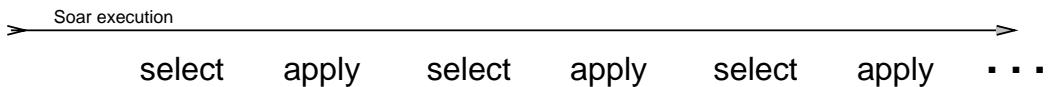


Figure 2.1: Soar is continually trying to select and apply operators.

situation and its long-term procedural 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.)

Long-term procedural knowledge is held in **production memory**. Procedural knowledge specifies how to respond to different situations in working memory, can be thought of as the program for Soar. The Soar architecture cannot solve any problems without the addition of long-term procedural 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 Types of Procedural Knowledge in Soar

Soar’s procedural knowledge can be categorized into four distinct types of knowledge:

1. *Inference Rules*

In Soar, we call these state elaborations. This knowledge provides monotonic inferences that can be made about the state in a given situation. The knowledge created by such rules are not persistent and exist only as long as the conditions of the rules are met.

2. *Operator Proposal Knowledge*

Knowledge about when a particular operator is appropriate for a situation. Note that multiple operators may be appropriate in a given context. So, Soar also needs knowledge to determine which of the candidates to choose:

3. *Operator Selection Knowledge*:

Knowledge about the desirability of an operator in a particular situation. Such knowledge can be either in terms of a single operator (e.g. never choose this operator in this situation) or relational (e.g. prefer this operator over another in this situation).

4. *Operator Application Rules*

Knowledge of how a specific selected operator modifies the state. This knowledge creates persistent changes to the state that remain even after the rule no longer matches or the operator is no longer selected.

Note that state elaborations can indirectly affect operator selection and application by creating knowledge that the proposal and application rules match on.

2.1.2 Problem-Solving Functions in Soar

These problem-solving functions are the primitives for generating behavior 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 on top 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.

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 knowledge 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 no new operators 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 2.7.

2.1.3 An Example Task: The Blocks-World

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

The Soar code for this task is available online at
<https://web.eecs.umich.edu/~soar/blocksworld.soar>.
You do not need to look at the code at this point.

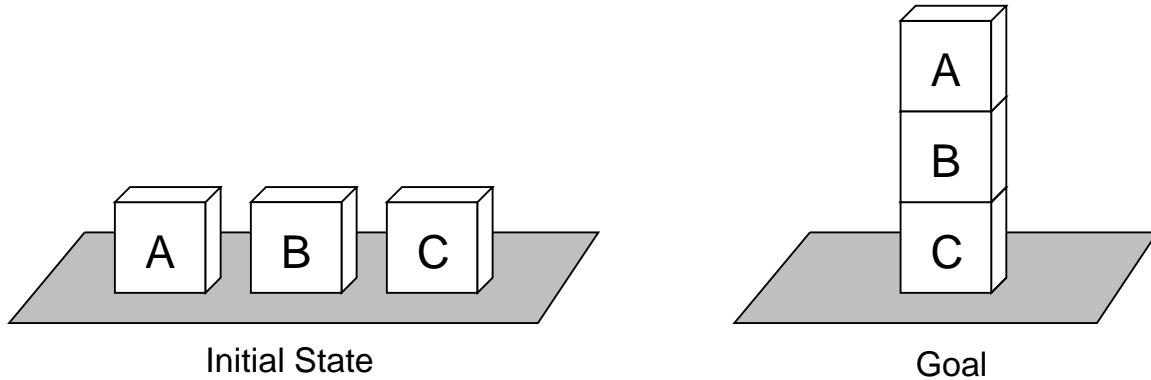


Figure 2.2: The initial state and goal of the “blocks-world” task.

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 top of block C, and block A on top of block B.

2.1.4 Representation of States, Operators, and Goals

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

A state can have only one selected operator at a time but it may also have a number of *potential* operators that are in consideration. These proposed operators should not be confused with the active, selected operator.

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

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

For example, the goal in our blocks-world task is represented implicitly in the provided Soar program. This is because a single production rule monitors the state for completion of the goal and halts Soar when the goal is achieved. (Syntax of Soar programs will be explained in Chapter 3.) If the goal was an explicit working memory structure, a rule could compare

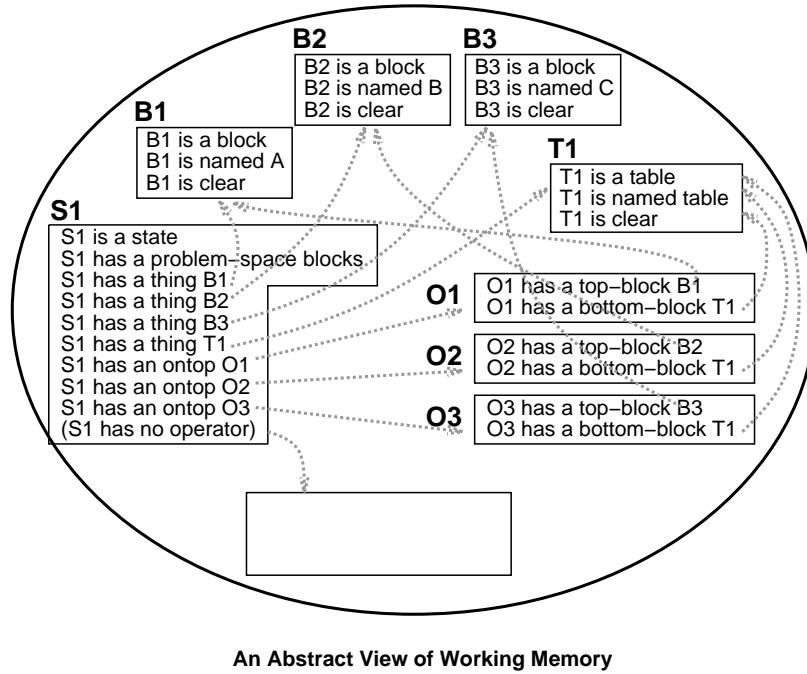


Figure 2.3: An abstract illustration of the initial state of the blocks world as working memory objects. At this stage of problem solving, no operators have been proposed or selected.

the configuration of blocks to that structure instead of having the goal embedded within the rule's programming.

2.1.5 Proposing candidate operators

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

2.1.6 Comparing candidate operators: Preferences

The second step Soar takes in selecting an operator is to evaluate or compare the candidate operators. In Soar, this is done via rules that test the proposed operators and the current state, and then create **preferences** (stored in *preference memory*). 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. Preferences are discussed in detail in section 2.4.2.

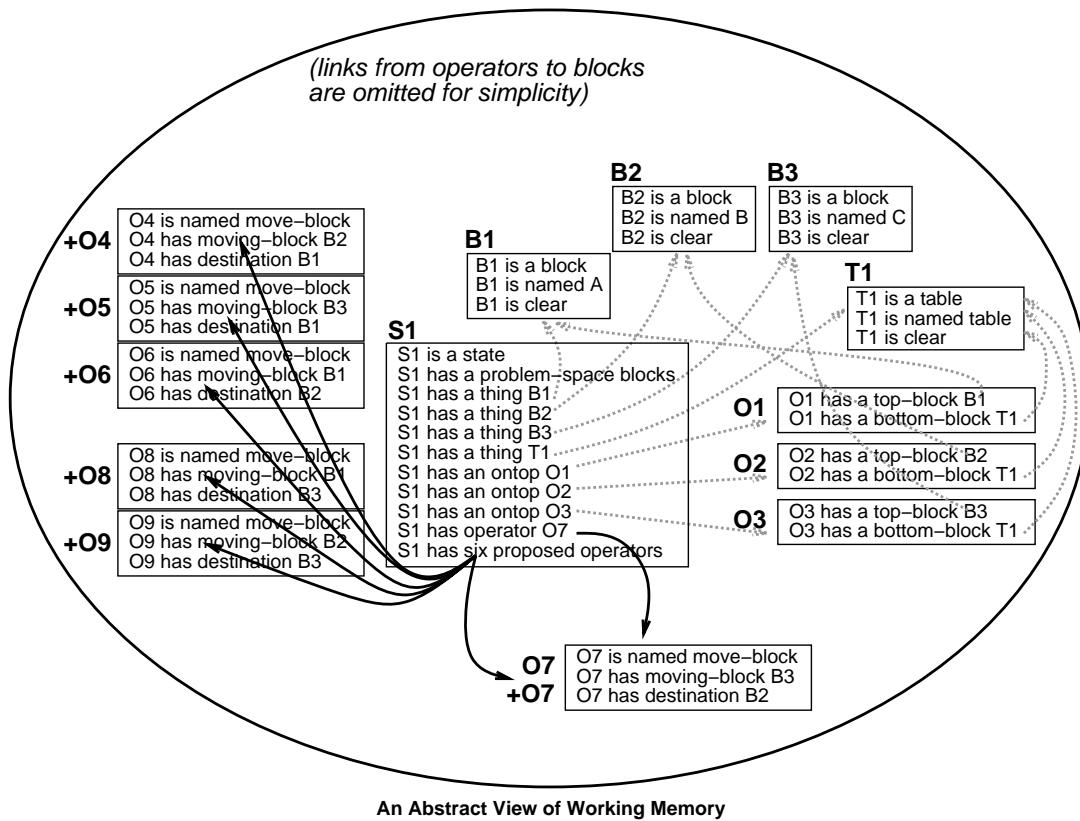


Figure 2.4: An abstract illustration of working memory in the blocks world after the first operator has been selected.

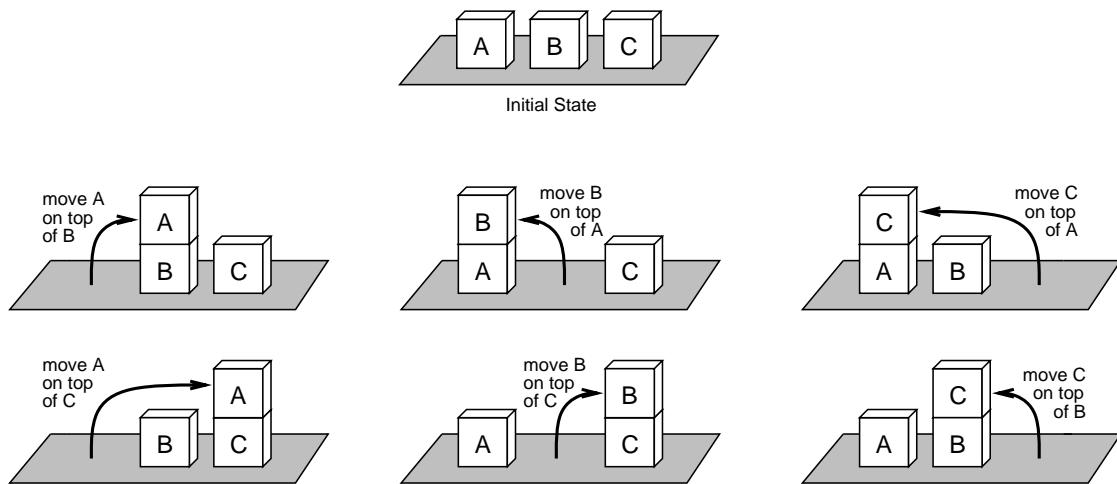


Figure 2.5: The six operators proposed for the initial state of the blocks world each move one block to a new location.

2.1.7 Selecting a single operator: Decision

Soar attempts to select a single operator as a decision, 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 2.7.

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

2.1.8 Applying the operator

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

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

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

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

The example blocks-world task described here does not interact with an external environment. Therefore, the Soar program directly makes changes to the state when operators are

applied. There are four changes that may need to be made when a block is moved in our task:

1. The block that is being moved is no longer where it was (it is no longer “on top” of the same thing).
2. The block that is being moved is in a new location (it is “on top” of a new thing).
3. The place that the block used to be in 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. In this case, the Soar program does not update all the “on top” and “clear” relations; the updated state description comes from the simulator.

2.1.9 Making inferences about the state

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

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

2.1.10 Problem Spaces

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

Soar programs are implicitly organized in terms of problem spaces because the conditions for proposing operators will restrict an operator to be considered only when it is relevant. The complete problem space for the blocks world is shown in Figure 2.6. Typically, when

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

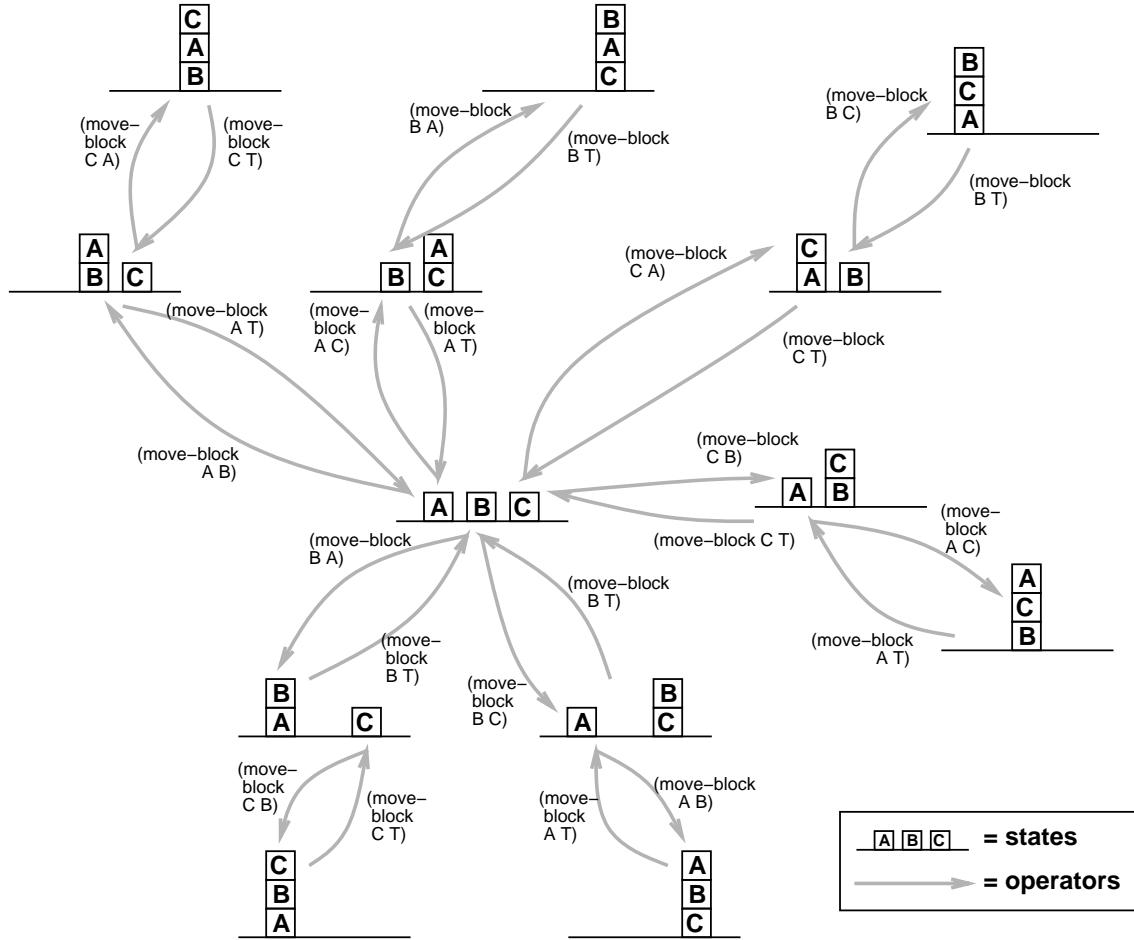


Figure 2.6: The problem space in the blocks-world includes all operators that move blocks from one location to another and all possible configurations of the three blocks.

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

2.2 Working memory: The Current Situation

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

Working memory contains elements called working memory elements, or WMEs for short. Each WME contains a very specific piece of information; for example, a WME might say that “B1 is a block”. Several WMEs 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 WMEs 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 WMEs that share this identifier are referred to as an *object* in working memory. Each WME describes a different *attribute* of the object, for example, its name or type or location; each *attribute* has a *value* associated with it, for example, the name is A, the type is block, and the position is on the table. Therefore, each WME is an identifier-attribute-value triple, and all WMEs 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 WMEs might describe the object T1: “T1 is a table”, “T1 is brown”, and “T1 is ontop of F1”. And still another collection of WMEs 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.

WMEs 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 (as a single *identifier-attribute-value* triple), while augmentation is a term that is used more often to refer to the description of an object. Working memory is illustrated at an abstract level in Figure 2.3 on page 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 06. When the value is an identifier, it refers to an object in working memory that may have additional substructure. In semantic net terms, if a value is a constant, then it is a terminal node with no links; if it is an identifier it is a nonterminal node.

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

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.

different values. When this happens, we say the attribute is a *multi-valued attribute*, which is often shortened to be *multi-attribute*.

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

There is no predefined relationship between objects in working memory and “real objects” in the outside world. Objects in working memory may refer to real objects, such as `block A`; features of an object, such as the color `red` or shape `cube`; a relation between objects, such as `ontop`; classes of objects, such as `blocks`; etc. The actual names of attributes and values have no meaning to the Soar architecture (aside from a few WMEs created by the architecture itself). For example, Soar doesn’t care whether the things in the blocks world are called “blocks” or “cubes” or “chandeliers”. It is up to the Soar programmer to pick suitable labels and to use them consistently.

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

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

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

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

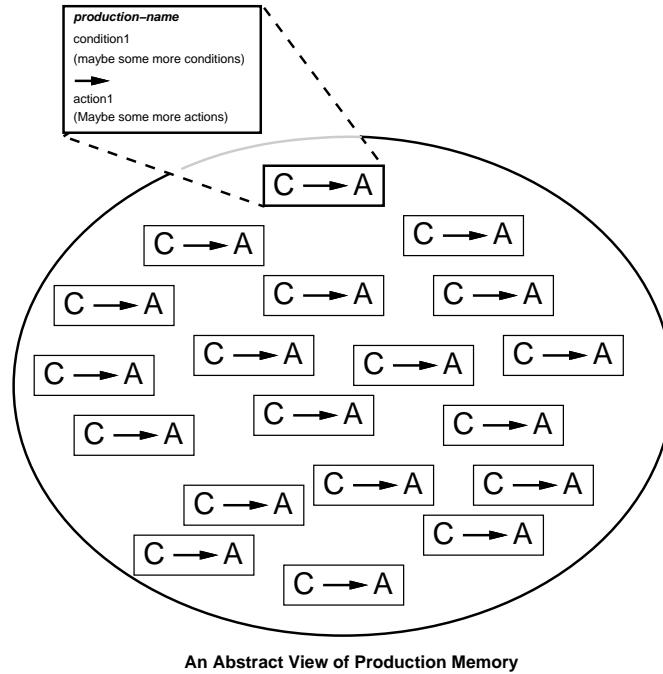


Figure 2.7: An abstract view of production memory. The productions are not related to one another.

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

Preferences are held in a separate *preference memory* where they cannot be tested by productions. There is one notable exception. Since a soar program may need to reason about candidate operators, **acceptable** preferences are made available in working memory as well. The acceptable preferences can then be tested by productions, which allows a Soar program to reason about candidates operators to determine which one should be selected. Preference memory and the different types of preferences will be discussed in Section 2.4.

2.3 Production Memory: Long-term Procedural Knowledge

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

2.3.1 The structure of a production

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

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

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

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

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

2.3.1.1 Variables in productions and multiple instantiations

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

When variables are bound to specific symbols in working memory elements by Soars matching process, Soar creates an *instantiation* of the production. This instantiation consists of the matched production along with a specific and consistent set of symbols that matched the variables. A production instantiation is consistent only if every occurrence of a variable is bound to the same value. Multiple instantiations of the same production can be created since the same production may match multiple times, each with different variable bindings. If blocks A and B are clear, the first production (without variables) will suggest one operator. However, consider a new proposal production that used variables to test the names of the block. Such a production will be instantiated twice and therefore suggest *two* operators: one operator to move block A on top of block B and a second operator to move block B on top of block A.

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

Just as the elements of working memory must be linked to a state in working memory, so

must the objects referred to in a production’s conditions. That is, one condition must test a state object *and* all other conditions must test that same state or objects that are linked to that state.

2.3.2 Architectural roles of productions

Soar productions can fulfill the following four roles, by retrieving different types of procedural knowledge, all described on page 6:

1. Operator proposal
2. Operator comparison
3. Operator application
4. State elaboration

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

2.3.3 Production Actions and Persistence

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

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

Thus, operator application productions do not retract their actions, even if they no longer match working memory. This is called *o-support* (for “operator support”). Working memory

elements that participate in the application of operators are maintained throughout the existence of the state in which the operator is applied, unless explicitly removed (or if they become unlinked). Working memory elements are removed by a *reject* action of a operator-application rule.

Whether a working memory element receives o-support or i-support is determined by the structure of the production instantiation that creates the working memory element. O-support is given only to working memory elements created by operator-application productions in the state where the operator was selected.

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.

During productions matching, all productions that have their conditions met fire, creating preferences which may add or remove 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 procedure 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”.

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 section 2.4.2.

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 (> *value*), **Worse (< *value*)** 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.

Unary Indifferent (=) A **unary indifferent** preference states that there is positive knowledge that a single value is as good or as bad a choice as other expected alternatives.

When two or more competing values both have **indifferent** preferences, by default, Soar chooses randomly from among the alternatives. (The **decide indifferent-selection** function can be used to change this behavior as described on page 195 in Chapter 9.)

Binary Indifferent (= *value*) A **binary indifferent** preference states that two values are mutually **indifferent** and it does not matter which of these values are selected. It behaves like a **unary indifferent** preference, except that the operator value given this preference is only made **indifferent** to the operator value given as the argument.

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, and behaves in that manner when competing with another

value having a unary indifferent preference. But when a set of competing operator values have `numeric-indifferent` preferences, the decision mechanism will choose an operator based on their numeric-indifferent values and the exploration policy. The available exploration policies and how they calculate selection probability are detailed in the documentation for the `indifferent-selection` command on page 195. When a single operator is given multiple numeric-indifferent preferences, they are either averaged or summed into a single value based on the setting of the `numeric-indifferent-mode` command (see page 195).

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

Require (!) A `require` preference states that the value *must* be selected if the goal is to be achieved. A `required` value is preferred over all others. Only a single operator value should be given a `require` preference at a time.

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 2.7.

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.

2.4.2 How preferences are evaluated to decide an operator

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 2.8. (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 is the set of current operator preferences, and the output consists of:

1. A subset of the candidate operators, which is either the empty set, a single, winning candidate, or a larger set of candidates that may be conflicting, tied, or indifferent.

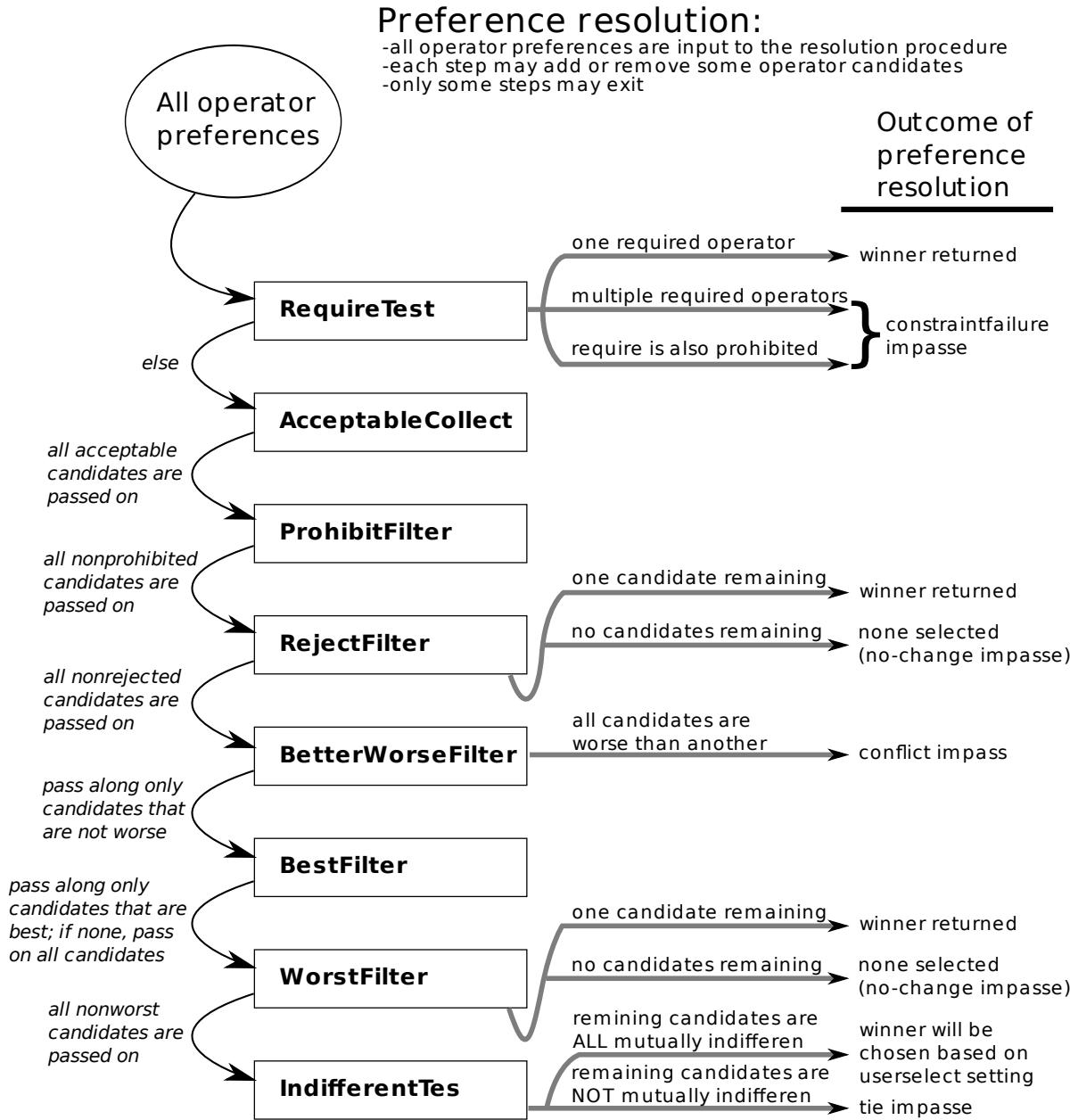


Figure 2.8: An illustration of the preference resolution process. There are eight steps; only five of these provide exits from the resolution process.

2. An 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 2.8 is described below:

RequireTest (!) This test checks for required candidates in preference memory and also constraint-failure impasses involving require preferences (see Section 2.7 on page 27).

- If there is exactly one candidate operator with a require preference and that candidate does not have a prohibit preference, then that candidate is the winner and preference semantics terminates.
- Otherwise — If there is more than one required candidate, then a constraint-failure impasse is recognized and preference semantics terminates by returning the set of required candidates.
- Otherwise — If there is a required candidate that is also prohibited, a constraint-failure impasse with the required/prohibited value is recognized and preference semantics terminates.
- Otherwise — There is no required candidate; 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.

Exit Point 1 :

- At this point, if the set of remaining candidates is empty, a no-change impasse is created with no operators being selected.
- If the set has one member, preference semantics terminates and this set is returned.
- Otherwise, the remaining candidates are passed to the BetterWorseFilter.

BetterWorseFilter (>), (<) This filter removes any candidates that are worse than another candidate.

Exit Point 2 :

- If the set of remaining candidates is empty, a conflict impasse is created returning the set of all candidates passed into this filter, i.e. all of the conflicted operators.
- If the set of remaining candidates has one member, preference semantics terminates and this set is returned.
- Otherwise, the remaining candidates are passed to the BestFilter.

BestFilter (>) If some remaining candidate has a best preference, this filter removes any candidates that do not have a best preference. If there are no best preferences for any of the current candidates, the filter has no effect. The remaining candidates are passed to the WorstFilter.

Exit Point 3 :

- At this point, if the set of remaining candidates is empty, a no-change impasse is created with no operators being selected.
- If the set has one member, preference semantics terminates and this set is returned.
- Otherwise, the remaining candidates are passed to the WorstFilter.

WorstFilter (<) This filter removes any candidates that have a worst preference. If all remaining candidates have worst preferences or there are no worst preferences, this filter has no effect.

Exit Point 4 :

- At this point, if the set of remaining candidates is empty, a no-change impasse is created with no operators being selected.
- If the set has one member, preference semantics terminates and this set is returned.
- Otherwise, the remaining candidates are passed to the IndifferentFilter.

IndifferentFilter (=) This operation traverses the remaining candidates and marks each candidate for which one of the following is true:

- the candidate has a unary indifferent preference
- the candidate has a numeric indifferent preference

This filter then checks every candidate that is not one of the above two types to see if it has a binary indifferent preference with every other candidate. If one of the candidates fails this test, then the procedure signals a tie impasse and returns the complete set of candidates that were passed into the IndifferentFilter. Otherwise, the candidates are mutually indifferent, in which case an operator is chosen according to the method set by the `decide indifferent-selection` command, described on page 195.

2.5 Soar’s Execution Cycle: Without Substates

The execution of a Soar program proceeds through a number of **decision 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.

An important aspect of productions in Soar to keep in mind is that all productions will always fire whenever their conditions are met, and retract whenever their conditions are no longer met. The exact details of this process are shown in Figure 2.9. The *Proposal* and *Application* phases described above are both composed of as many **elaboration cycles** as are necessary to reach quiescence. In each elaboration cycle, all matching productions fire and the working memory changes or operator preferences described through their actions are made. After each elaboration cycle, if the working memory changes just made change the set of matching productions, another cycle ensues. This repeats until the set of matching rules remains unchanged, a situation called **quiescence**.

After quiescence is reached in the *Proposal* phase, the *Decision* phase ensues, which is the architectural selection of a single operator, if possible. Once an operator is selected, the *Apply* phase ensues, which is practically the same as the *Proposal* phase, except that any productions that apply the chosen operator (they test for the selection of that operator in their conditions) will now match and fire.

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. In other words, if, during the *Apply* phase, the production(s) that proposed the selected operator retract, that *Apply* phase will immediately end.

2.6 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.

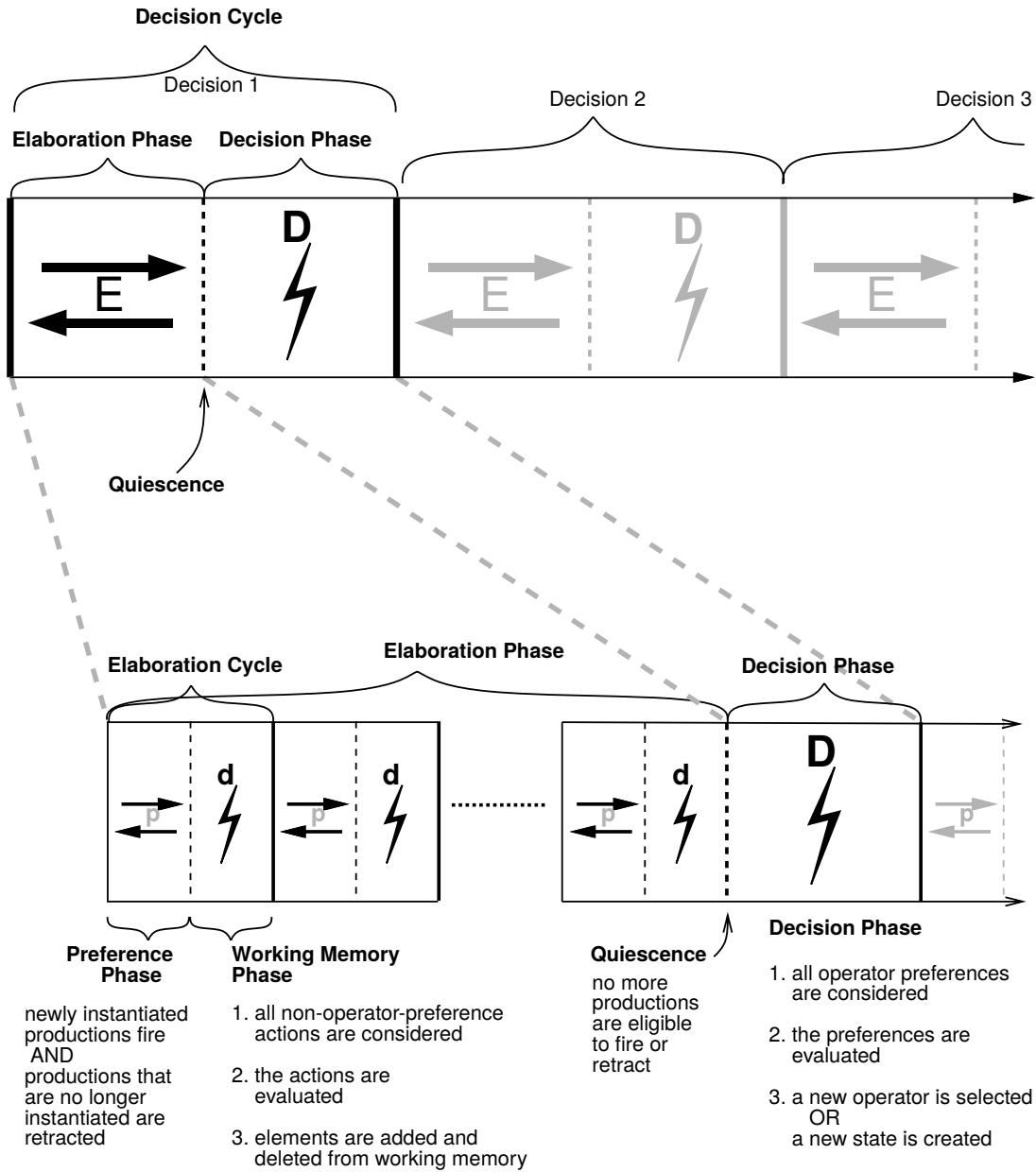


Figure 2.9: A detailed illustration of Soar’s decision cycle.

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. See Section 3.5 for more information.

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

2.7 Impasses and Substates

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

2.7.1 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 that have `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 four problem-solving functions (described in section 2.1.2) was incomplete.

2.7.2 Creating New States

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

The initial state in the subgoal contains a complete description of the cause of the impasse, such as the operators that could not be decided among (or that there were no operators proposed) and the state that the impasse arose in. From the perspective of the new state, the latter is called the **superstate**. Thus, the superstate is part of the substructure of each state, represented by the Soar architecture using the **superstate** attribute. (The initial state, created in the 0th decision cycle, contains a **superstate** attribute with the value of **nil** — the top-level state has no superstate.)

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

In the substate, operators can be selected and applied as Soar attempts to solve the subgoal. (The operators proposed for solving the subgoal may be similar to the operators in the superstate, or they may be entirely different.) While problem solving in the subgoal, additional impasses may be encountered, leading to new subgoals. Thus, it is possible for Soar to have a *stack* of subgoals, represented as states: Each state has a single superstate (except the initial state) and each state may have at most one substate. Newly created subgoals are considered to be added to the bottom of the stack; the first state is therefore called the *top-level state*.³ See Figure 2.11 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.

2.7.3 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 (the lowest-level state) and is not considered to have been created in the other states. The architecture automatically detects if a preference or working memory element created in a substate is also linked to a superstate.

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

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

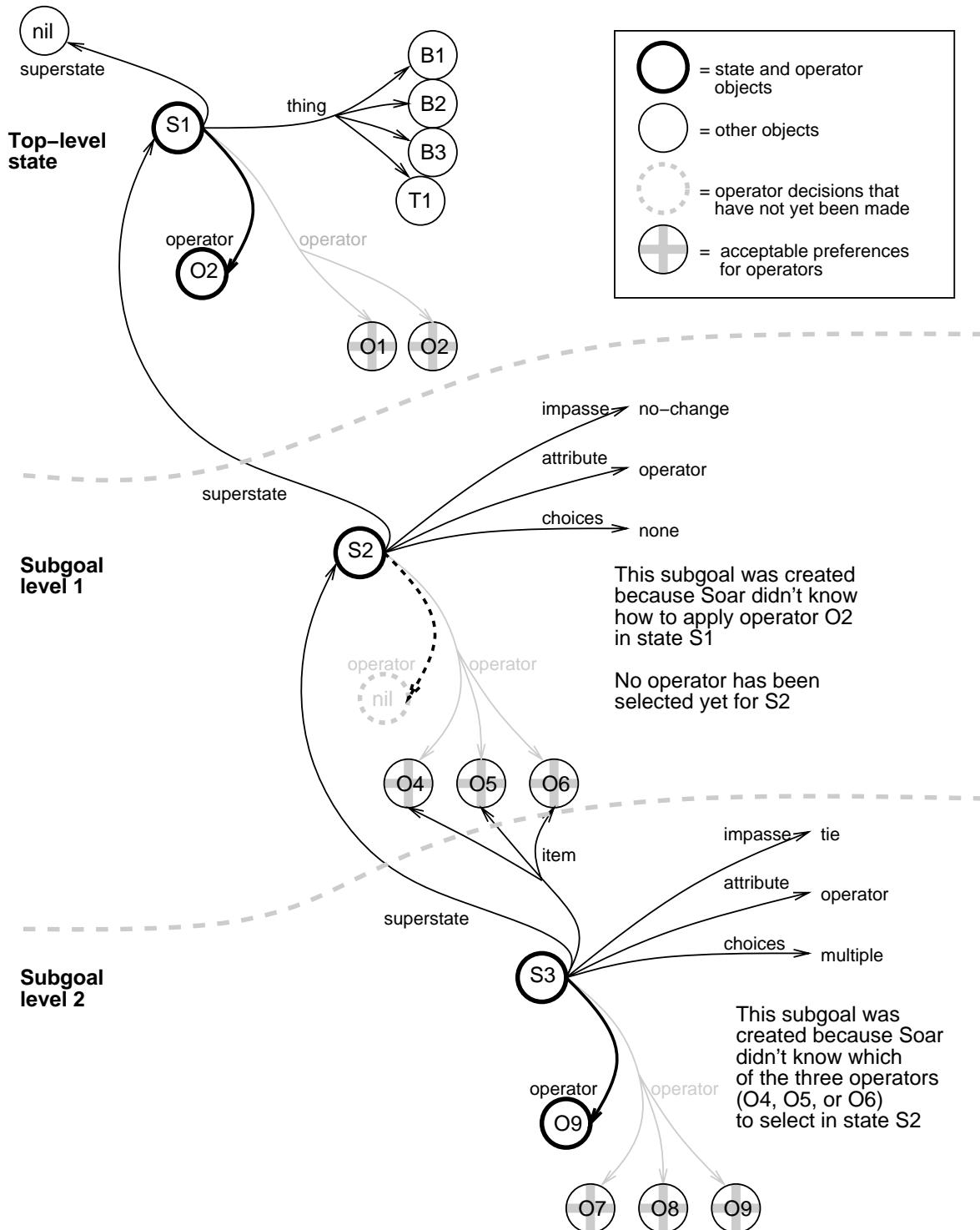


Figure 2.11: A simplified illustration of a subgoal stack.

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 operator then become results because the operator's identifier is now linked to the superstate.

2.7.4 Justifications: Support for results

Recall from Section 2.3.3 that WMEs with *i-support* disappear as soon as the production that created them retract,⁴ whereas WMEs with *o-support* (created through applying an operator) persist in working memory until deliberately removed.

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.

Thus, when the substate disappears, the generated justification serves as the production that supports any subgoal results.

Soar determines i-support or o-support for the justification and its actions 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

⁴ Technically, an i-supported 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 instantiated productions and the retraction of only one would not lead to the retraction of the WME.

justification. If such a result loses i-support from the justification, it will be retracted if there is no other support.

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 Sections 4.5.2.1 and 4.6.10 for details).

2.7.5 Chunking: Learning Procedural Knowledge

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

One of Soar’s learning mechanisms is called **chunking** (See chapter 4 for more information); 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 and justifications are very similar in that they both summarize substate results. They are, in fact, generated by the architecture using the same result dependency trace mechanisms. However, there are some important distinctions:

1. Justifications disappear as soon as its conditions no longer match.
2. Chunks contain variables so that they may match working memory in other situations; justifications are similar to an instantiated chunk.

In other words, a chunk might be thought of as a permanent and potentially more generalized form of a justification. Since the result that solves the impasse problem is learned in a chunk, whenever the agent encounters the same situation again as that which resulted in the original impasse, it can simply fire the chunk to generate the same result previously derived, preempting the need for a substate and repeated deliberate problem solving.

2.7.6 The calculation of o-support

This section provides a more detailed description of when an action is given o-support by an instantiation.⁵ The content here is somewhat more advanced, and the reader unfamiliar

⁵ In the past, Soar had various experimental support mode settings. Since version 9.6, the support mode used is what was previously called `mode 4`.

with rule syntax (explained in Chapter 3) may wish to skip this section and return at a later point.

Support is given by the production; that is, all working memory changes generated by the actions of a single instantiated production will have the same support (an action that is not given o-support will have i-support). The conditions and actions of a production rule will here be referred to using the shorthand of LHS and RHS (for Left-Hand Side and Right-Hand Side), respectively.

A production must meet the following two requirements to have o-supported actions:

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

```
(<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 this production,

```
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, this production,

```
sp {operator*superstate*application
    (state <s> ^superstate <s1>)
        ^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.

An extension of condition 1 is that operator augmentations should always receive i-support (augmentations define the proposed operator). Soar has been written to recognize augmentations directly off the operator

(ie, `(<o> ^augmentation value)`), and to attempt to give them i-support. However, what

should be done about a production that simultaneously tests an operator, doesn't propose an operator, adds an operator augmentation, and adds a non-operator augmentation? For example:

```
sp {operator*augmentation*application
  (state <s> ^task test-support
   ^operator <o>)
  -->
  (<o> ^new augmentation)
  (<s> ^new augmentation)}
```

In such cases, both receive i-support. Soar will print a warning on firing this production, because this is considered bad coding style.

2.7.7 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. A substate can resolve a constraint-failure impasse through actions that cause all but one of the conflicting preferences to retract.

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, change the state so the operator proposal no longer matches, or cause other operators to be 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 these cases, the impasse is eliminated, but not “resolved”, and Soar does not learn in this situation.

For example, in the blocks-world domain, an agent might deliberate in a substate to determine whether it should move block *A* onto block *C* or block *B* onto block *C* in its current situation. If a child suddenly throws block *A* out a window, this problem solving becomes irrelevant, and the impasse is eliminated.

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 surrounding 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 *Goal Dependency Set* (GDS) 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 (in other words, any superstate knowledge used to derive persistent substate knowledge). Whenever such an o-supported WME is

created, Soar records which superstate WMEs were tested, directly or indirectly, to create it. Whenever any of the WMEs in the dependency set of a substate change, the substate is regenerated. (See Sections 9.3.1.2 and 9.6.1.1 for how to examine GDS information through the user-interface.)

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 2.11, 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 09 will also be removed, as will the acceptable preferences for 07, 08, and 09, 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 04, 05, and 06 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).

2.7.8 Soar’s Cycle: With Substates

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

The main change when there are multiple substates is that at each phase of the decision cycle, 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.

2.7.9 Removal of Substates: The Goal Dependency Set

This subsection describes the Goal Dependency Set (GDS) with discussions on the motivation for the GDS and behavioral consequences of the GDS from a developer/modeler’s point of view. It goes into greater detail than might be beneficial for someone becoming familiar with the general operation of Soar for the first time. Readers may skip this section and return later if desired.

2.7.9.1 Why the GDS was needed

As a symbol system, Soar attempts to approximate a true knowledge level but will necessarily always fall short. We can informally think of the way in which Soar falls short as its peculiar “psychology.” Those interested in using Soar to model human cognition 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. 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 thesis⁶ 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) WMEs in a substate and its “context”. The context consists of all the WMEs in any superstates above the local goal/state.⁷ In Soar, any action (application) of an operator receives an o-support 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 prior to GDS, especially when trying to create a large-scale system, is that the knowledge developer must always think about which dependencies can be “ignored” and which may affect 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

⁶ Robert E. Wray. *Ensuring Reasoning Consistency in Hierarchical Architectures*. PhD thesis, University of Michigan, 1998.

⁷ This subsection will primarily use “state,” not “goal.” While these terms are often used nearly-interchangeably in the context of Soar, states refer to the set of WMEs comprising knowledge related to a peculiar level of goal. The *Goal* Dependency Set is the set of state elements upon which a goal depends.

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 in isolation 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 that agents 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.

2.7.9.2 Behavior-level view of the Goal Dependency Set

The following discussion covers what the GDS does, and how that impacts production knowledge design and implementation.

Operation of the Goal Dependency Set: Consider i-support. The persistence of an i-supported (“instantiation supported”) WME depends upon the creating 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 that support for the WME. I-support is illustrated in Figure 2.12. A copy of **A** in the subgoal, \mathbf{A}_s , is retracted automatically when **A** changes to **A'**. The substate WME persists only as long as it remains justified by **A**.

In the broadest sense, we can say that some feature $\langle b \rangle$ is “dependent” upon another element $\langle a \rangle$ if $\langle a \rangle$ was used in the creation of $\langle b \rangle$, i.e., if $\langle a \rangle$ was tested in the production instantiation that created $\langle b \rangle$. Further, a dependent change with respect to feature $\langle b \rangle$ is a change to any of its instantiating features. This applies to both i-supported and o-supported WMEs. In Figure 2.12, the change from **A** to **A'** is a dependent change for feature **1** because **A** was used to create **1**.

When **A** changes, the persistent WME **1** may be no longer consistent with its context (e.g., **A'**). The specific solution to this problem through GDS is inspired by the dependency analysis portion of the justification/chunking algorithm (see Chapter 4). Whenever an o-supported WME is created in the local state, the superstate dependencies of that new feature

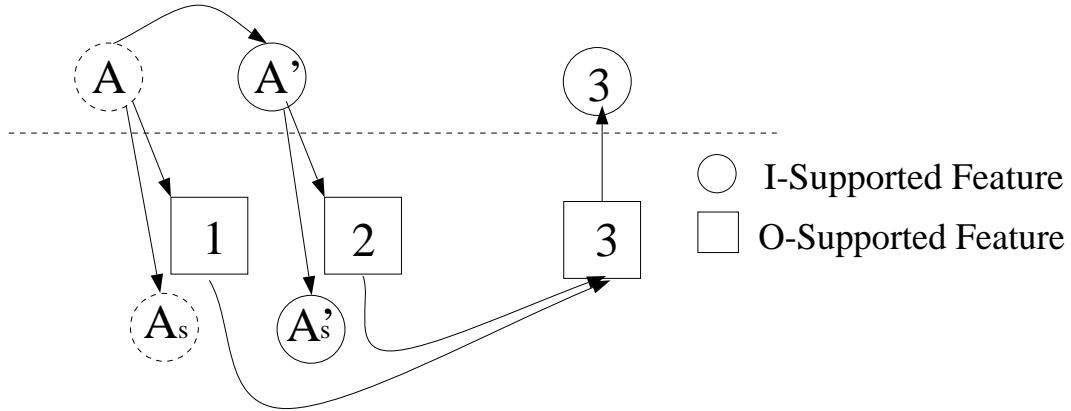


Figure 2.12: Simplified Representation of the context dependencies (above the line), local o-supported WMEs (below the line), and the generation of a result. Prior to GDS, this situation led to non-contemporaneous constraints in the chunk that generates **3**.

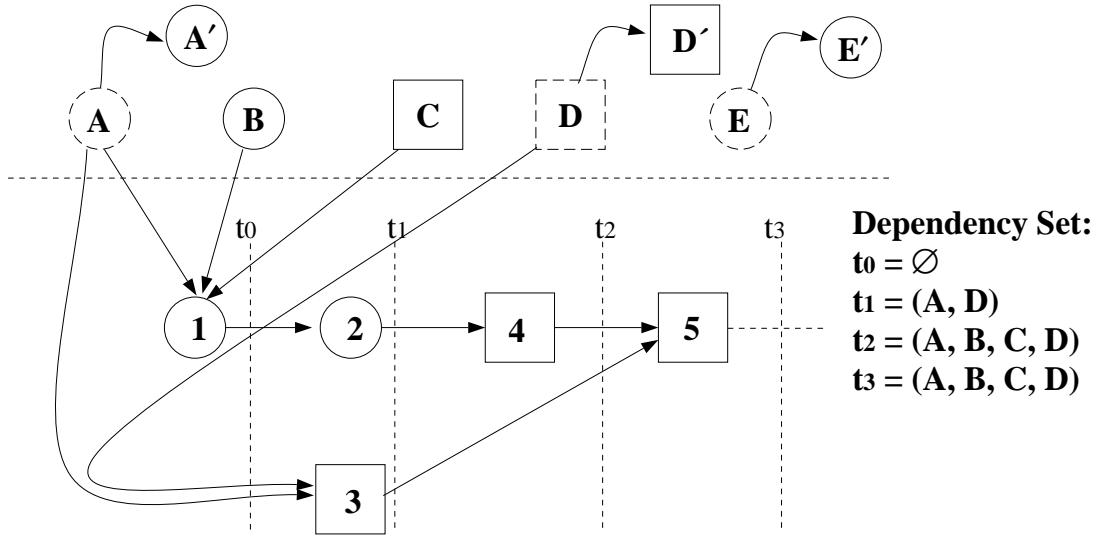


Figure 2.13: The Dependency Set in Soar.

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. *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 2.13, 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.

1. **Elements in the superstate:** WMEs in the superstate are added directly to the goal's dependency set. In Figure 2.13, the persistent subgoal item **3** is dependent upon **A** and **D**. These superstate WMEs are added to the subgoal's dependency set when **3** is added to working memory at time t_1 . It does not matter that **A** is i-supported and **D** o-supported.
2. **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.
3. **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 2.13, 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.

At any time after t_1 , either the **D** to **D'** or **A** to **A'** transition would cause the removal of the entire subgoal. The **E** to **E'** transition causes no retraction because **E** is not in the goal's dependency set.

The role of the GDS in agent design: The GDS places some design time constraints on operator implementation. These constraints are:

- Operator actions that are used to remember a previous state/situation should be asserted in the top state.
- All operator elaborations should be i-supported.
- Any operator with local actions should be designed to be re-entrant.

Because any dependencies for o-supported subgoal WMEs 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 structures may need to reflect hypothetical states. These are “assumptions” because a hypothetical inference cannot always be grounded in the current context. In problem solvers with truth maintenance, only assumptions are persistent.
2. **Reasoning non-monotonically:** Sometimes the result of an inference changes one of the structures 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 derive some belief in a new world state but the expense of performing the necessary computation makes this derivation undesirable. 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. 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 .

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. Further discussion of Soar I/O can be found on the Soar website.

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 tuple 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 first position always holds an internal identifier symbol, generated by the Soar architecture as it runs. The attribute and value positions can hold either identifiers or constants. The term *identifier* is used to refer both to the first position of a WME, as well as to the symbols that occupy that position. If a WME’s attribute or value is an identifier, there is at least one WME that has that identifier symbol in its first 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 022.

(The Soar user interface will also allow users to specify identifiers using lowercase letters in a case-insensitive manner, 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: |0dd-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 supplementary blocks-world program provided [online](#), after one elaboration cycle, you can look at the top-level state object by using the `print` command:

```
soar> print s1
(S1 ^io I1 ^ontop 02 ^ontop 03 ^ontop 01 ^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 `print --internal` command can be used:

```
soar> print --internal S1
(3: S1 ^io I1)
(10: S1 ^ontop 02)
(9: S1 ^ontop 03)
(11: S1 ^ontop 01)
(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 operators 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 linked above, after the first operator has been selected, you can again look at the top-level state using the `print --internal` command:

```
soar> print --internal s1
(3: S1 ^io I1)
(9: S1 ^ontop 03)
(10: S1 ^ontop 02)
(11: S1 ^ontop 01)
(48: S1 ^operator 04 +)
(49: S1 ^operator 05 +)
(50: S1 ^operator 06 +)
(51: S1 ^operator 07 +)
(54: S1 ^operator 07)
(52: S1 ^operator 08 +)
(53: S1 ^operator 09 +)
(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 (**04** through **09**). These have plus signs following the value to denote that they are acceptable preferences. The state has exactly one operator, **07**. This state corresponds to the illustration of working memory in Figure 2.4.

3.1.5 Working Memory as a Graph

Not only is working memory a set, it is also a graph structure where the identifiers are nodes, attributes are links, and constants are terminal nodes. Working memory is not an arbitrary graph, but a graph rooted in the states (e.g. **S1**). Therefore, all WMEs are *linked* either

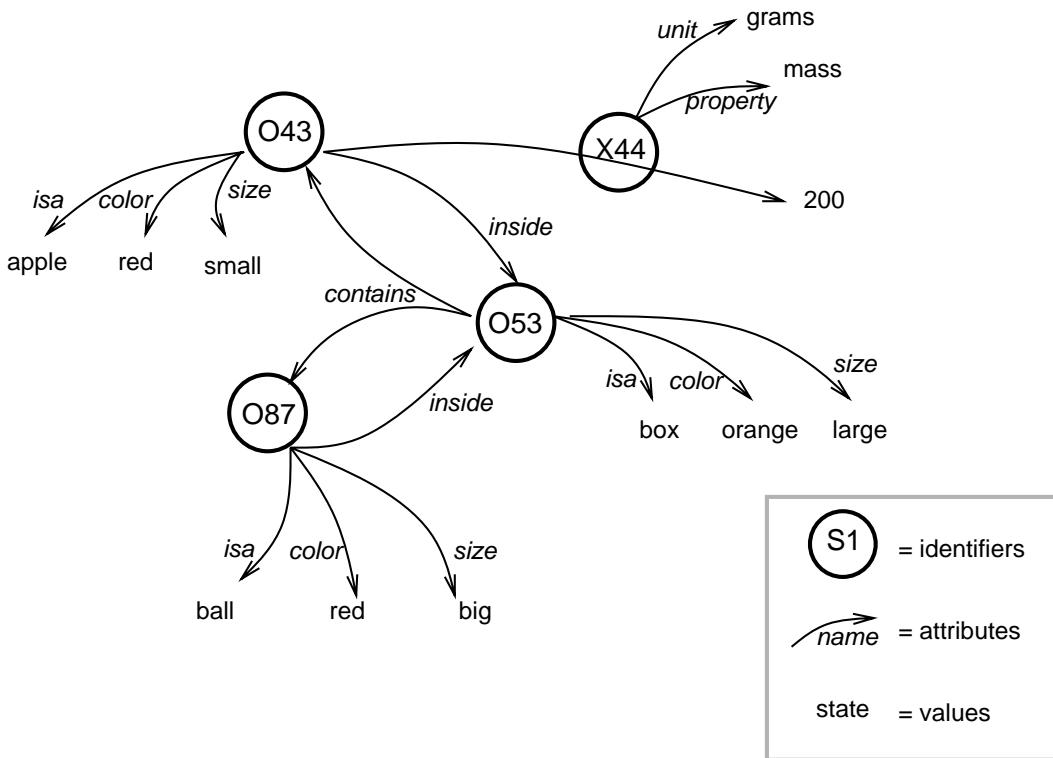


Figure 3.1: A semantic net illustration of four objects in working memory.

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

```
(043 ^isa apple ^color red ^inside 053 ^size small ^X44 200)
(087 ^isa ball ^color red ^inside 053 ^size big)
(053 ^isa box ^size large ^color orange ^contains 043 087)
(X44 ^unit grams ^property mass)
```

In this example, object 043 and object 087 are both linked to object 053 through (053 ^contains 043) and (053 ^contains 087), respectively (the `contains` attribute is a multi-valued attribute). Likewise, object 053 is linked to object 043 through (043 ^inside 053) and linked to object 087 through (087 ^inside 053). Object X44 is linked to object 043 through (043 ^X44 200).

Links are transitive so that 053 is linked to X44 (because 053 is linked to 043 and 043 is linked to X44). However, since links are not symmetric, X44 is not linked to 053.

3.1.6 Working Memory Activation

WMEs have a form of base level activation associated with them that is not accessible to the agent, but that is used by the architecture. **Working Memory Activation** (WMA) is subsymbolic metadata associated with a given element and represents its usage. A WME has been *used* if it has been matched in a rule that fired. WMA is not recorded or maintained when disabled, which is the default. See Section 9.3.2 for working memory settings and options for enabling WMA.

Simply enabling WMA has no impact on any agent's behavior outside of a small additional computational cost. However, working memory activation is used for other features. Primarily, it is necessary for allowing the forgetting of memory elements from working memory. When working memory forgetting is turned on, those working memory elements with activation below a given threshold are removed from working memory. This allows agents to maintain a bounded working memory size without explicit memory size management. It also has a role in determining spreading activation values, discussed in section 6.4.2.1.

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 entered in by a user (typed in while Soar is running or loaded from a file) or generated by chunking while Soar is running. Productions (both user-defined productions and chunks) may be examined using the `print`

```

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

```

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

command, described in Section 9.3.1 on page 215.

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

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

```

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

```

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

Conventions for indenting productions

Productions in this manual are formatted using conventions designed to improve their readability. These conventions are not part of the required syntax. First, the name of the production immediately follows the first curly bracket after the `sp`. All conditions are aligned

with the first letter after the first curly brace, and attributes of an object are all aligned. The arrow is indented to align with the conditions and actions and the closing curly brace follows the last action.

3.3.1 Production Names

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

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

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

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

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

3.3.2 Documentation string (optional)

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

3.3.3 Production type (optional)

A production may also include an optional *production type*, which may specify that the production should be considered a default production (`:default`) or a chunk (`:chunk`), or may specify that a production should be given o-support (`:o-support`) or i-support (`:i-support`). Users are discouraged from using these types.

Another flag (`:template`) can be used to specify that a production should be used to generate new reinforcement learning rules. See Section 5.4.2 on page 140 for details.

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 9.2.1 on Page 200.

These types are described in Section 9.2.1, which begins on Page 200.

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.

load file 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. (Each instance binds the rule to specific WMEs.)

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

3.3.5.1 Conditions

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

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

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

Although all of the attribute tests in the example condition 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 symbols in WMEs 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 is 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 non-pipe (|) character in between.

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

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

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

3.3.5.3 Predicates for values

A test for an identifier, attribute, or value in a condition (whether constant or variable) can be modified by a preceding predicate. There are six general 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
> <valuem>	> > <valueym>
< 1	1 >
<=> <y>	= 10

There are also four special predicates that can be used to test Long-Term Identifier (LTI) links held by working memory identifiers: @, !@, @+, @-

Predicate	Semantics of Predicate
@	Same LTI. Matches when the two values are working memory identifiers linked to the same LTI.
!@	Different LTI. Matches when the values are not both identifiers linked to the same LTI.
@+	Matches if the value is an identifier linked to some LTI.
@-	Matches if the value is not an identifier linked to some LTI.

See Section 6.2 for more information on long-term semantic memory and LTIs.

Example Productions

```
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”.

```
sp {example*lti*predicates
  (state <s> ^existing-item { @+ <orig-sti> }
    ^smem.result.retrieved { @ <orig-sti> <result-sti> })
  -->
  ... }
```

In this production, <orig-sti>, is tested for whether it is linked to some LTI. It is also compared against <result-sti> (a working memory element retrieved from long-term memory and known to be linked to an LTI) to see if the two elements point to the same long-term memory. Note the the @+ in this example is actually unnecessary, since the { @ <orig-sti> <result-sti> } test will fail to match if either value tested is not linked to an LTI.

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 >>	<< <var> 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> >= }	{ <x> < <a> + }
{ <x> > <y> }	{ > > }
{ <> <x> <y> }	{ <a> }
{ <y> <> <x> }	
{ << 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 ``.

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

The third and fourth examples are equivalent. They state that 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 fifth example, the value must be one of `A`, `B`, or `C`, and the second conjunctive test binds the value to variable `<x>`.

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

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

Note that it is illegal syntax for a condition to test the equality of two variables, as demonstrated in the last illegal conjunction above. Any such test can instead be coded in simpler terms by only using one variable in the places where either would be referenced throughout the rule.

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. For example, the following rule’s actions will cause it to no longer match, which will cause the action to retract, which will cause the rule to match, and so on:

```
sp {example*infinite-loop
  (state <s> ^car <c>
    -^road )
  -->
  (<s> ^road |route-66|) }
```

Also note that syntactically it is invalid for the first condition of a rule to be a negated condition. For example, the following production would fail to load:

```
sp {example*invalid-negated-first-condition
  (state <s> -^road <r>
    ^car <c>)
  -->
  ... }
```

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 tests 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 accord-

ing to some attribute, `<a>`, and that block `<t>` and `<t2>` both have this attribute (whatever it is), and have the same value for the attribute.

Predicates in attributes

Predicates may be used with attributes, as in:

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

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

Disjunctions of attributes

Disjunctions may also be used with attributes, as in:

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

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

Conjunctive tests for attributes

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

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

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

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

test includes a variable used to bind to the attribute name. Thus,

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

is interpreted to mean:

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

3.3.5.11 Attribute-path notation

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

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

For example, using attribute path notation, the production:

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

could be written as:

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

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

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

notation.

Negations and attribute path notation

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

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

could be rewritten as:

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

Multi-valued attributes and attribute path notation

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

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

Multi-attributes and attribute-path notation

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

```
sp {blocks-world*propose*move-block*dont-do-this
  (state <s> ^problem-space blocks
```

```

^clear.block <block1>
^clear.block { <> <block1> <block2> }
^ontop.top-block <block1>
^ontop.bottom-block <> <block2>)

(<block1> ^type block)
-->

...
}

```

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

```

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

```

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

Negated multi-valued attributes and attribute-path notation

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

For example,

```

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

```

gets expanded to:

```
sp {blocks*negated-conjunction-example
```

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

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

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

which expands to:

```
sp {blocks*negated-conjunction-example
  (state <s> ^name top-state)
  -{(<s> ^ontop <o*2>)
    (<o*2> ^bottom-object <b*2>)
    (<b*2> ^name a)}
  -{(<s> ^ontop <o*1>)
    (<o*1> ^bottom-object <b*1>)
    (<b*1> ^name table)}
-->
  (<s> ^nothing-ontop-a-or-table true +) }
```

Notes on attribute-path notation

- Attributes specified in attribute-path notation may not start with a digit. For example, if you type `^foo.3.bar`, Soar thinks the `.3` is a floating-point number. (Attributes that don't appear in path notation can begin with a number.)
- Attribute-path notation may be used to any depth.
- Attribute-path notation may be combined with structured values, described in Section 3.3.5.12.

3.3.5.12 Structured-value notation

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

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

Using structured-value notation, the production in Figure 3.2 (on page 49) may also be written as:

```
sp {blocks-world*propose*move-block
  (state <s> ^problem-space blocks
    ^thing <thing1>
    ^thing {<> <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 is not referenced again, so this is not helpful in this instance):

```
sp {blocks-world*propose*move-block
  (state <s> ^problem-space blocks
    ^thing <thing1>
    ^thing {<> <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 can be combined with negations and with multi-attributes.
- Structured-value notation can 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 it 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 removal should be used with an action that will be o-supported. If removal is attempted with i-support, the working memory element will reappear if the removal 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 ($<\text{s1}> \ ^\text{operator} <\text{o1}> > <\text{o2}>$).

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. ($\langle s \rangle \ ^\text{operator} \langle o_1 \rangle$) is assumed to mean ($\langle s \rangle \ ^\text{operator} \langle o_1 \rangle \ +$).

Note however that the + is only implicit if no other preferences are specified for that operator. Specifying a preference that is not the acceptable preference does not also imply an acceptable preference. For example, ($\langle s \rangle \ ^\text{operator} \langle o_1 \rangle \ >$) by itself cannot lead to $\langle o_1 \rangle$ being selected, since it does not have an acceptable preference.

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⟩ < ⟨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.

Commas are only allowed in rule syntax for this sort of use, in the RHS. They can be used to separate actions, and if used when no disambiguation is needed will have no effect other than syntactic sugar.

As another example, ($\langle s \rangle \ ^\text{operator} \langle o_1 \rangle \langle o_2 \rangle \ > \langle o_3 \rangle$) would be interpreted as

```
(⟨s⟩ ^operator ⟨o1⟩ +
    ^operator ⟨o2⟩ > ⟨o3⟩)
```

But ($\langle s \rangle \ ^\text{operator} \langle o_1 \rangle \langle o_2 \rangle \ >, \langle o_3 \rangle$) would be interpreted as

```
(⟨s⟩ ^operator ⟨o1⟩ +
    ^operator ⟨o2⟩ >
    ^operator ⟨o3⟩ +)
```

3.3.6.6 Right-hand side Functions

The fourth type of action that can occur in productions is called a *right-hand side function*. Right-hand 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.

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 the agent is to be restarted (see the `interrupt` RHS action below.)

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

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

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

Soar execution may also be stopped immediately before a production fires, using the `:interrupt` directive. This functionality is called a matchtime interrupt and is very useful for debugging. See Section 9.2.1 on Page 200 for more information.

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

wait — Executing this function causes the current Soar thread to sleep for the given integer number of milliseconds.

```
sp {
  ...
-->
  (wait 1000) }
```

Note that use of this function is discouraged.

Text input and output

These functions are provided as production actions 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>) }
```

log — This function is equivalent to the `write` function, except that it specifies the “trace channel” for output. It takes two arguments. First is an integer corresponding to the channel level for output, second is the message to print.

See section 9.6.1 for information about trace channel levels.

```
sp {
  ...
-->
  (log 3 |This only prints when trace is set to 3 or higher!|) }
```

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. These functions can each take any number of arguments, and will return the result of sequentially operating on each argument. The **-** symbol is also a unary function which, given a single argument, returns the product of the argument and **-1**. The **/** symbol is also a unary function which, given a single argument, returns the reciprocal of the argument ($1/x$).

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

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

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

abs, atan2, sqrt, sin, cos — These 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>)) }
```

min, max — These symbols provide n-ary mathematical functions (they each take a list of symbols as arguments). These symbols work similarly to C functions. They take either integer or real-number arguments, and return a real-number value if any of their arguments are real-numbers. Otherwise they return integers.

```
sp {
...
-->
(<s> ^max (max <x> 3.14 <z>
    ^min (min <a> <b> 42 <c>)) }
```

int — Converts a single symbol to an integer constant. This function expects either an integer constant, symbolic constant, or floating point constant. The symbolic constant must be a string which can be interpreted as a single integer. The floating point constant is truncated to only the integer portion. This function essentially operates as a type casting function.

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

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

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

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

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

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

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

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.

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)
```

compute-heading — This function takes four real-valued arguments of the form (x_1, y_1, x_2, y_2) , and returns the direction (in degrees) from (x_1, y_1) to (x_2, y_2) , rounded to the nearest integer.

For example:

```
sp {
  ...
  -->
  (<s> ^heading (compute-heading 0 0.5 32.5 28)) }
```

After this rule fires, working memory would look like:

```
(S1 ^heading 48).
```

compute-range — This function takes four real-valued arguments of the form (x_1, y_1, x_2, y_2) , and returns the distance from (x_1, y_1) to (x_2, y_2) , rounded to the nearest integer.

For example:

```
sp {
  ...
  -->
  (<s> ^distance (compute-range 0 0.5 32.5 28)) }
```

After this rule fires, working memory would look like:

```
(S1 ^distance 42).
```

concat — Given an arbitrary number of symbols, this function concatenates them together into a single constant symbol.

For example:

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

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

deep-copy — This function returns a copy of the given symbol along with linked copies of all descendant symbols. In other terms, a full copy is made of the working memory subgraph that can be reached when starting from the given symbol. All copied identifiers are created as new IDs, and all copied values remain the same.

For example:

```
sp {
  (state <s> ^tree <t>)
  (<t> ^branch1 foo ^branch2 <b>)
  (<b> ^branch3 <t>)
  -->
  (<s> ^tree-copy (deep-copy <t>)) }
```

After this rule fires, the following structure would exist:

```
(S1 ^tree T1 ^tree-copy D1)
  (T1 ^branch1 foo ^branch2 B1)
    (B1 ^branch3 T1)
  (D1 ^branch1 foo ^branch2 B2)
    (B2 ^branch3 D1)
```

dc — This function takes no arguments, and returns the integer number of the current decision cycle.

For example:

```
sp {example
  (state <s> ^type state)
  -->
  (<s> ^dc-count (dc) )}
```

@ **(get)** — This function returns the LTI number of the given ID. If the given ID is not linked to an LTI, it does nothing.

For example:

```
sp {example
  (state <s> ^stm <l1>)
  -->
  (<s> ^lti-num (@ <l1>) )}
```

After this rule fires, the (`S1 ^lti-num`) WME will have an integer value such as 42.

link-stm-to-ltm — This function takes two arguments. It links the first given symbol to the LTI indicated by the second integer value.

For example:

```
sp {example
  (state <s> ^stm <l1>)
  -->
  (link-stm-to-ltm <l1> 42) )
```

After this rule fires, the WME (`S1 ^stm <l1>`) will be linked to @42.

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)
```

rand-float — This function takes an optional positive real-valued argument. If no argument (or a negative argument) is given, it returns a random real-valued number in the range [0.0, 1.0]. Otherwise, given a value n , it returns a number in the range [0.0, n].

For example:

```
sp {
  ...
  -->
  (<s> ^fate (rand-float 1000)) }
```

After this rule fires, working memory might look like:

```
(S1 ^fate 275.481802).
```

rand-int — This function takes an optional positive integer argument. If no argument (or a negative argument) is given, it returns a random integer number in the range $[-2^{31}, 2^{31}]$. Otherwise, given a value n , it returns a number in the range $[0, n]$.

For example:

```
sp {
  ...
  -->
  (<s> ^fate (rand-int 1000)) }
```

After this rule fires, working memory might look like:

```
(S1 ^fate 13).
```

round-off — This function returns the first given value rounded to the nearest multiple of the second given value. Values must be integers or real-numbers.

For example:

```
sp {
  (state <s> ^pi <pi>
  -->
  (<s> ^pie (round-off <pi> 0.1)) }
```

After this rule fires, working memory might look like:
 $(S1 \ ^pi \ 3.14159 \ ^pie \ 3.1)$.

round-off-heading — This function is the same as `round-off`, but additionally shifts the returned value by multiples of 360 such that $-360 \leq value \leq 360$.

For example:

```
sp {
  (state <s> ^heading <dir>
  -->
  (<s> ^true-heading (round-off-heading <dir> 0.5)) }
```

After this rule fires, working memory might look like:
 $(S1 \ ^heading \ 526.432 \ ^true-heading \ 166.5)$.

size — This function returns an integer symbol whose value is the count of WME augmentations on a given ID argument. Providing a non-ID argument results in an error.

For example:

```
sp {
  (state <s> ^numbers <n>)
  (<n> ^1 1 ^10 10 ^100 100)
  -->
  (<s> ^augs (size <n>)) }
```

After this rule fires, the value of $S1 \ ^augs$ would be 3.

Note that some architecturally-maintained IDs such as $(<s> \ ^epmem)$ and $(<s> \ ^io)$ are not counted by the `size` function.

strlen — This function returns an integer symbol whose value is the size of the given string symbol.

For example:

```
sp {
  (state <s> ^io.input-link.message <m>)
  ...
  -->
  (<s> ^message-len (strlen <m>)) }
```

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
```

trim — This function takes a single string symbol argument and returns the same string with leading and trailing whitespace removed.

For example:

```
sp {
  (state <s> ^message <m>)
-->
  (<s> ^trimmed (trim <m>)) }
```

User-defined functions and interface commands as RHS actions

Any function which has a certain function signature may be registered with the Kernel (e.g. using SML) and called as a RHS function. The function must have the following signature:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.3.6.7 Controlling chunking

Chunking is described in Chapter 4.

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

The `dont-learn` and `force-learn` RHS actions are to be used with specific settings for the `chunk` command (see page 232.) Using the `chunk` command, learning may be set to one of `always`, `never`, `flagged`, or `unflagged`; chunking must be set to `flagged` for the `force-learn` RHS action to have any effect and chunking must be set to `unflagged` for the `dont-learn` RHS action to have any effect.

dont-learn — When chunking is set to unflagged, by default chunks can be formed in all states; the **dont-learn** RHS action will cause chunking 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 **chunk** is set to **unflagged**, and has no effect when other settings for **chunk** are used.

force-learn — When learning is set to flagged, by default chunks are not formed in any state; the **force-learn** RHS action will cause chunking 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 **chunk** is set to **flagged**, and has no effect when other settings for **chunk** are used.

3.3.7 Grammars for production syntax

This subsection 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 subsection.)

This information is provided for advanced Soar users, for example, those who need to write their own parsers. Note that some terms (e.g. **<sym_constant>**) are undefined; as such, this grammar should only be used as a starting point.

3.3.7.1 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)
```

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>.

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> <preferenceSpecifier>*

<preferenceSpecifier> ::= <unary-preference> [",,,"]
| <unary-or-binary-preference> [",,,"]
| <unary-or-binary-preference> <rhs_value> [",,,"]

```

```

<unary-pref>      ::= "+" | "-" | "!" | "~"
<unary-or-binary-pref>  ::= ">" | "=" | "<"

```

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 2.7:

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

All impasses lead to the creation of a new substate in working memory, and appear as objects within that substate. These objects 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 2.7 on page 27.)

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 constraint-failure or no-change impasses).

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

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

- ^item For multi-choice and constraint-failure impasses, this contains all values involved in the tie, conflict, or constraint-failure. If the set of items that tie or conflict changes during the impasse, the architecture removes or adds the appropriate item augmentations without terminating the existing impasse.
- ^item-count For multi-choice and constraint-failure impasses, this contains the number of values listed under the item augmentation above.
- ^non-numeric For tie impasses, this contains all operators that do not have numeric indifferent preferences associated with them. If the set of items that tie changes during the impasse, the architecture removes or adds the appropriate non-numeric augmentations without terminating the existing impasse.
- ^non-numeric-count For tie impasses, this contains the number of operators listed under the non-numeric augmentation above.
- ^quiescence States are the only objects with quiescence t, which is an explicit statement that quiescence (exhaustion of the elaboration cycle) was reached in the superstate. If problem solving in the subgoal is contingent on quiescence having been reached, the substate should test this flag. The side-effect is that no chunk will be built if it depended on that test. See Section 4.6.11 on page 112 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/default`s 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 09 010 011 ^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 02)
```

3.4.2 Testing for impasses in productions

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

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

```
sp {default*top-goal*halt*operator*failure
    "Halt if no operator can be selected for the top goal."
    :default
    (state <ss> ^impasse constraint-failure ^superstate <s>)
    (<s> ^superstate nil)
-->
    (write (crlf) |No operator can be selected for top goal.| )
    (write (crlf) |Soar will halt now. Goodnight.| )
    (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. Interfacing with a Soar agent through input and output can be done using the *Soar Markup Language* (SML). The details of designing an external process that uses SML to create the input and respond to output from Soar are beyond the scope of this manual, but they are described [online](#) on the Soar website. 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 works 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 structure.

3.5.2 Input and output in working memory

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

Suppose a blocks-world task is implemented using a robot to move actual blocks around, with a camera creating input to Soar and a robotic arm executing command outputs.

The camera image might be analyzed by a separate vision program; this program could have as its output the locations of blocks on an xy plane. The Soar input function could take the output from the vision program and create the following working memory elements on 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)
```

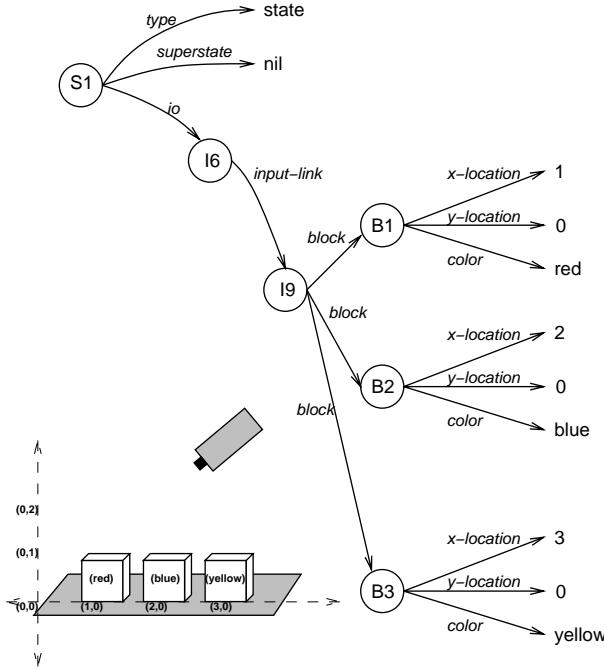


Figure 3.3: An example portion of the input link for the blocks-world task.

```
(B1 ^y-location 0)
(B1 ^color red)
(B2 ^x-location 2)
(B2 ^y-location 0)
(B2 ^color blue)
(B3 ^x-location 3)
(B3 ^y-location 0)
(B3 ^color yellow)
```

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

Then, during the Apply Phase of the execution cycle, Soar productions could respond to an operator, such as “move the red block ontop of the blue block” by creating a structure on the output link, such as:

```
(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)
```

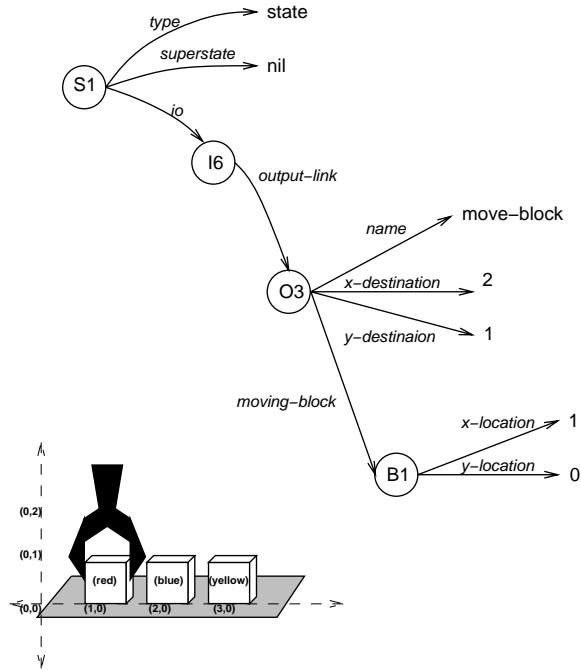


Figure 3.4: An example portion of the output link for the blocks-world task.

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

Procedural Knowledge Learning

4.1 Chunking

Chunking is Soar’s experience-based mechanism for learning new procedural knowledge. Chunking utilizes Soar’s impasse-driven model of problem decomposition into sub-goals to create new productions dynamically during task execution. These new productions, called **chunks**, summarize the substate problem-solving that occurred which led to new knowledge in a superstate. Whenever a rule fires and creates such new superstate knowledge, which are called **results**, Soar learns a new rule and immediately adds it to production memory. In future similar situations, the new chunk will fire and create the appropriate results in a single step, which eliminates the need to spawn another subgoal to perform similar problem-solving. In other words, rather than contemplating and figuring out what to do, the agent immediately knows what to do.

Chunking can effect both speed-up and transfer learning. A chunk can effect speed-up learning because it compresses all of the problem-solving needed to produce a result into a single step. For some real-world agents, hundreds of rule firings can be compressed into a single rule firing. A chunk can effect transfer learning because it generalizes the problem-solving in such a way that it can apply to other situations that are similar but have not yet been experienced by the agent.

Chunks are created whenever one subgoal creates a result in a superstate; since most Soar programs are continuously sub-goaling and returning results to higher-level states, chunks are typically created continuously as Soar runs. Note that Soar builds the chunk as soon as the result is created, rather than waiting until the impasse is resolved.

While chunking is a core capability of Soar, procedural learning is disabled by default. See section [4.7](#) for more information about enabling and using chunking.

```

sp {chunk-94*process-column*apply
  (state <s1> ^operator <o1>
    ^arithmetic-problem <a1>
    ^one-fact 1
    ^top-state <s1>
    ^arithmetic <a2>
    ^arithmetic <a3>)

  (<o1> ^name process-column)
  (<a1> ^operation subtraction
    ^current-column <c1>)
  (<c1> -^new-digit1 <n1>
    ^digit1 0
    ^digit2 7
    ^next-column <n2>)
  (<n2> ^digit1 0
    ^new-digit1 9
    ^next-column <n3>)
  (<n3> ^digit1 5
    ^new-digit1 4)
  (<a2> ^subtraction-facts <s2>
    ^subtraction-facts <s3>
    ^subtraction-facts <s4>)
  (<a3> ^add10-facts <a4>)

  (<a4> ^digit1 0
    ^digit-10 10)

  (<s2> ^digit1 10 ^digit2 1
    ^result 9)
  (<s3> ^digit1 5 ^digit2 1
    ^result 4)
  (<s4> ^digit1 10 ^digit2 7
    ^result 3)
-->
  (<c1> ^result 3) }

sp {chunk-96*process-column*apply
  (state <s1> ^operator <o1>
    ^arithmetic-problem <a1>
    ^one-fact <o2>
    ^one-fact <o3>
    ^top-state <t1>
    ^arithmetic <a2>
    ^arithmetic <a3>)

  (<o1> ^name process-column)
  (<a1> ^operation subtraction
    ^current-column <c1>)
  (<c1> -^new-digit1 <n1>
    ^digit1 { <d2> < <d1> }
    ^digit2 <d1>
    ^next-column <n2>)
  (<n2> ^digit1 { <d3> < <o3> }
    ^new-digit1 <n3>
    ^next-column <n4>)
  (<n4> ^digit1 { <d4> >= <o2> }
    ^new-digit1 <n5>)
  (<a2> ^subtraction-facts <s2>
    ^subtraction-facts <s3>
    ^subtraction-facts <s4>)
  (<a3> ^add10-facts <a4>
    ^add10-facts <a5>)
  (<a4> ^digit1 <d2>
    ^digit-10 { <d5> >= <d1> })
  (<a5> ^digit1 <d3>
    ^digit-10 { <d6> >= <o3> })
  (<s2> ^digit1 <d6> ^digit2 <o3>
    ^result <n3>)
  (<s3> ^digit1 <d4> ^digit2 <o2>
    ^result <n5>)
  (<s4> ^digit1 <d5> ^digit2 <d1>
    ^result <r1>)
-->
  (<c1> ^result <r1>)}

```

Figure 4.1: A Soar 9.4.0 chunk (left) vs. an explanation-based chunk (right) in the arithmetic demo agent

4.2 Explanation-based Chunking

Explanation-based chunking improves on previous versions of chunking by learning rules that are qualitatively more general and expressive. In fact, any element of a learned rule can now be variablized, and learned rules now have the full expressive power of hand-written rules.

Figure 4.1 shows an example of an explanation-based chunk and how it differs from a chunk learned from the original algorithm. It is interesting to note that in Soar 9.4, the arithmetic agent learns 1263 rules like the one on the left-side of the figure. In Soar 9.6, the same agent

only learns 8 rules like the one on the right because they are so much more general.

To achieve this generality, chunking needs information about why rules matched in a substate and how those rules interacted. This allows it to determine what is generalizable and what limits there are on those generalizations. Unfortunately, the information necessary to determine this information was not readily available in prior versions of Soar which only recorded a trace of all WMEs that were tested in the substate. This trace, which we call the *working memory trace* possesses limited explanatory information, which limited chunking to learning very specific rules in which only Soar identifiers were variablized and all other elements tested the exact values found in the working memory trace.

To remedy this limitation and produce more general chunks, EBC instead analyzes two traces simultaneously: the working memory trace and a corresponding trace of the handwritten rules that matched in the substate. This new network of rule matches is called the explanation trace:

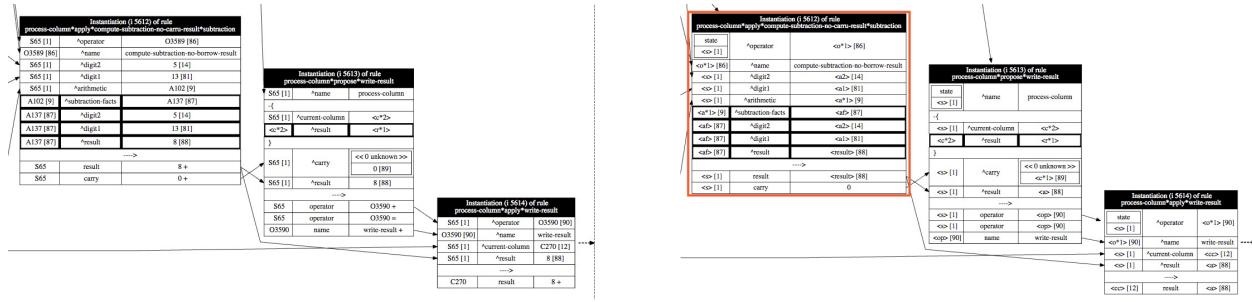


Figure 4.2: A close-up of a trace showing differences between a working memory trace (left) and an explanation trace (right). The working memory trace only contains the literal values of the WMEs that matched. The explanation trace, on the other hand, contains variables and various constraints on the values those variables can hold.

Note that this trace is generated dynamically as rules match. Whenever a rule matches during agent execution, Soar creates an internal record of the rule that fired, which is called a rule **instantiation**. (Each box in the explanation traces of this chapter represents an instantiation that was created during task execution within a particular substate.) The instantiation contains both instance information about what matched (the working memory elements) and explanatory information about why they matched (the rules and actions in the original rules that contains variables, constraint tests, RHS actions, etc.).

Note that WMEs that were automatically created by the architecture have special instantiations that explain why they were created. For example, an architectural instantiation is created for each `^item` attribute automatically created in operator tie impasse substates; the explanation causes the `^item` augmentation to be dependent on the operator in the superstate that led to it, which means that chunks learned which tested that `^item` augmentation will cause the chunk to also be dependent on the operator in the superstate.

Similarly, architectural instantiations are created for structures recalled by semantic and

episodic memory in the substate.

All of the instantiations that were created in a substate form the *instantiation graph* of that substate. As chunking **backtraces** through the instantiation graph, it determines the subset of instantiations that contributed to a result. This set of instantiations and the connections between them composes the explanation trace for a learning episode. (So, the explanation trace is a subgraph of the instantiation graph.)

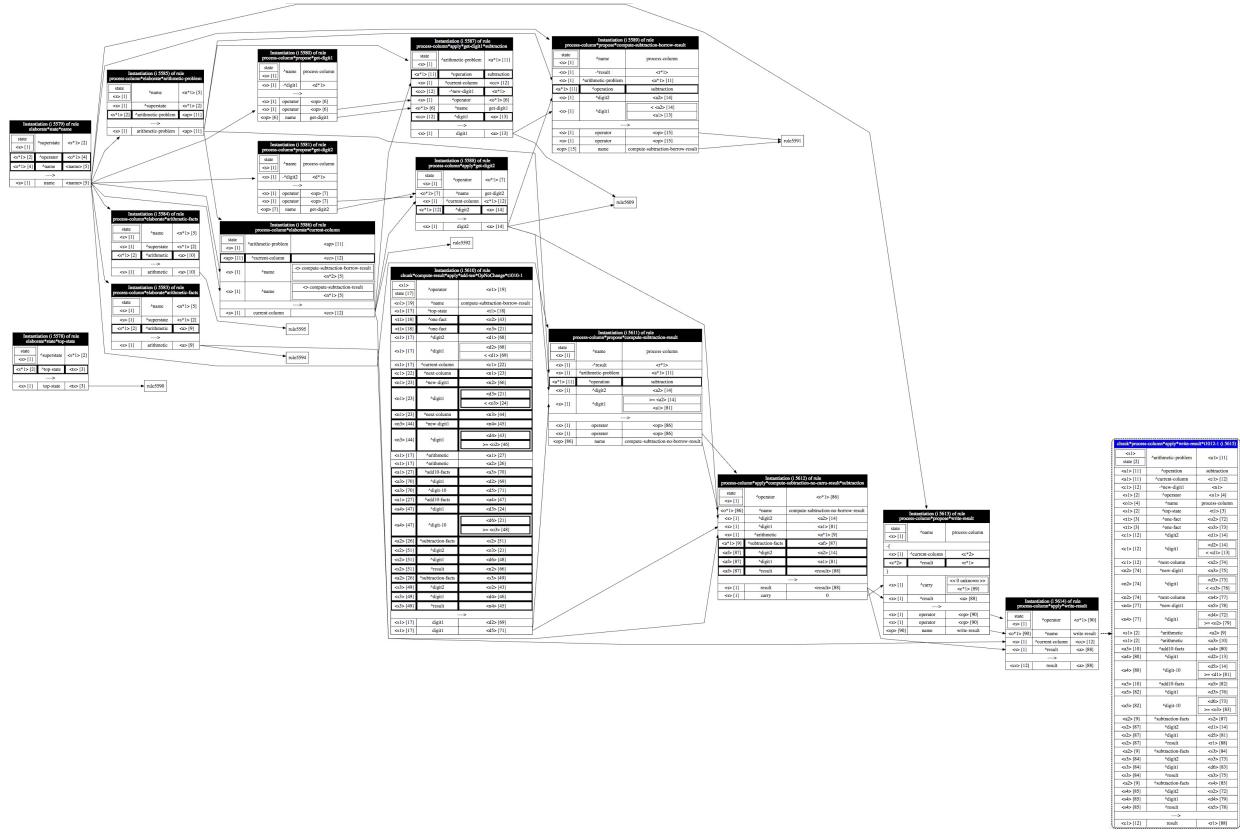


Figure 4.3: A visualization of the explanation trace of a chunk learned by the arithmetic agent. Each box represents a rule that fired in the substate. Arrows show dependencies between rules that create working memory elements and conditions that test those working memory elements.

EBC uses the explanation trace to determine (1) how variables were used during a problem-solving episode and (2) what constraints on those variables had to be met in order for the substate rules to match. EBC then uses the results of this analysis to create more expressive and general rules, which can contain the full gamut of tests that hand-written rules can and can have any element variablized.

4.3 Overview of the EBC Algorithm

Basic concepts:

- Every condition and action in the explanation trace has three *elements*:
 - ★ For conditions, the three elements refer to the symbol in the positive equality test for the identifier, attribute and value of the condition. For example, the last condition of rule 2 in Figure 4.4 has `<s>` as the identifier element, number as the attribute element, and `<y>` as the value element.
 - ★ For actions, the three elements refer to the identifier, attribute and value of the WME being created.
- An element is either a variable, like `<s>` or a literal constant, like `23`, `3.3` or `someString`
- .

4.3.1 Identity

Before we can discuss the algorithm, we must first define one of its central concepts: *identity*.

- **An identity is the set of all variables in a trace that refer to the same underlying object.**
 - ★ So we can say that two *variables* are said to *share an identity* if they both refer to the same underlying object.
- **The NULL identity is a special identity that indicates an element which cannot be generalized and must contain a specific value.**
 - ★ All elements in the original rule that reference specific constant values are trivially assigned the NULL identity.
 - ★ A variable's identity can also be *mapped to the NULL identity*. When this happens, we say the identity has been **literalized**.

EBC traverses an explanation trace of the problem-solving that occurred in the substate to determine which variables in different rule instances refer to the same underlying object. There are two ways that an explanation trace can show a shared identity:

1. Variables that have the same name and are in the same rule firing will share an identity
This is the trivial case. The basic semantics of rules implies that the same variable in a rule references the same underlying object.
2. If a RHS action of one rule creates a WME and a LHS condition of another rules tests that same WME, then all variables in the condition and actions will possess the same identity as their counterpart's corresponding element.
The interaction between the two rules indicates a shared identity between their corresponding variables.

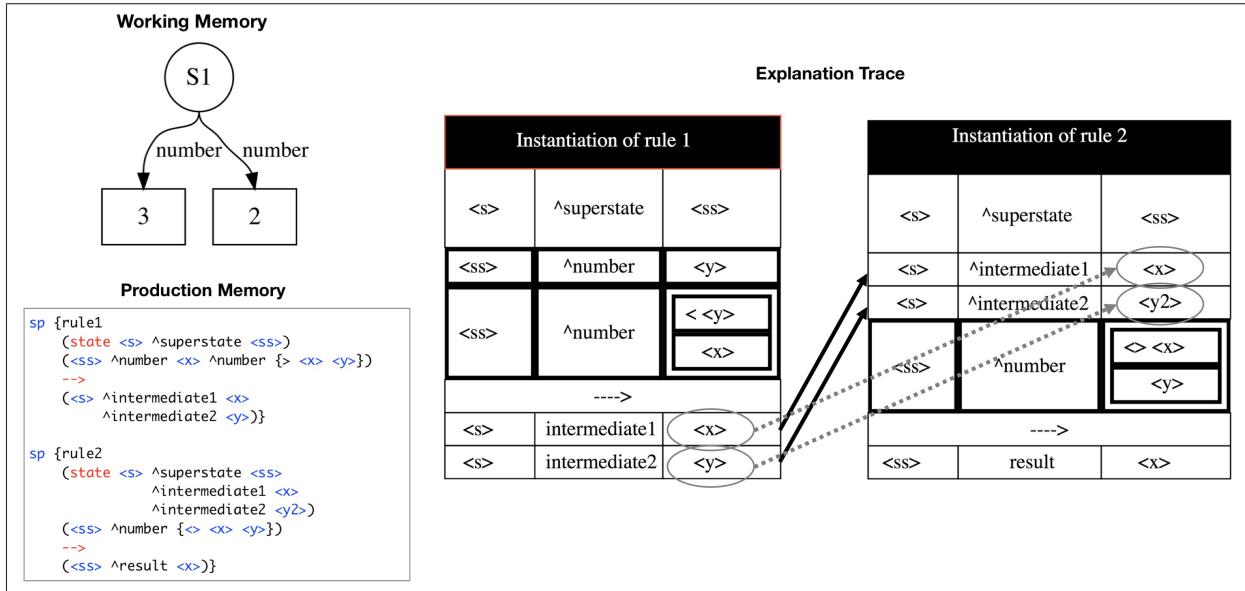


Figure 4.4: An explanation trace of two simple rules that matched in a substate

To get a better picture of what a shared identity is, consider the two simple rules and the explanation trace of how they matched in a substate as shown in Figure 4.4. The connection between rule 2 and rule 1 will *unify* the identities of $\langle s \rangle$, $\langle x \rangle$ and $\langle y \rangle$ in rule 1 with the identities of $\langle s \rangle$, $\langle x \rangle$ and $\langle y2 \rangle$ in rule 2. So, the $\langle x \rangle$ in rule 2 shares the same identity as the $\langle x \rangle$ in rule 1. Similarly, the $\langle y2 \rangle$ in rule 2 shares the same identity as $\langle y \rangle$ in rule 1. In contrast, the $\langle y \rangle$ in rule 2 does NOT share the same identity as the $\langle y \rangle$ in rule 1.

It doesn't matter that the $\langle y \rangle$ in rule 1 uses the same variable name as the $\langle y \rangle$ in rule 2. It also doesn't matter that both conditions with $\langle y \rangle$ happen to match the same working memory element, ($S1 \ ^number\ 3$). In terms of sharing an identity, the only thing that matters is how the rules interact, namely whether there's a connection between elements in the condition of one rule and elements in the actions of another rule.

All literal values, for example all of the attribute in Figure 4.4 (superstate, number, intermediate1, etc.) are considered members of the *NULL identity*.

Variable identities can also be mapped to the NULL identity, which means that any elements in the final rule that share that identity will not be variablized. When this happens, we say that the identity has been *literalized*. There are two ways that a rule interaction can effect an identity literalization:

1. If a RHS action of one rule creates a WME element using a constant, literal value in an element and a LHS condition tests that element, then the identity of the condition's variables is literalized and mapped to the NULL identity.

Because the variable in the condition matched a rule that will always create the same constant, literal value, the condition's variable must have that same value. Otherwise, it would not have matched.

2. If a RHS action of one rule creates a WME element using a variable and a LHS condition tests that that element is a specific value, then the identity of the action's variables is literalized and mapped to the NULL identity.

Because the condition requires that the rule that created the matched WME to have a specific constant, literal value, the action's variable must have that same value. Otherwise, it would not have created something that matched the condition.

Identities are the basis of nearly every mechanism in explanation-based chunking. EBC's identity analysis algorithm, which is a fairly complicated process, determines all shared identities in an explanation trace. Figure 4.5 shows an explanation trace after identity analysis has been performed. Elements that share an identity in the figure are colored the same.

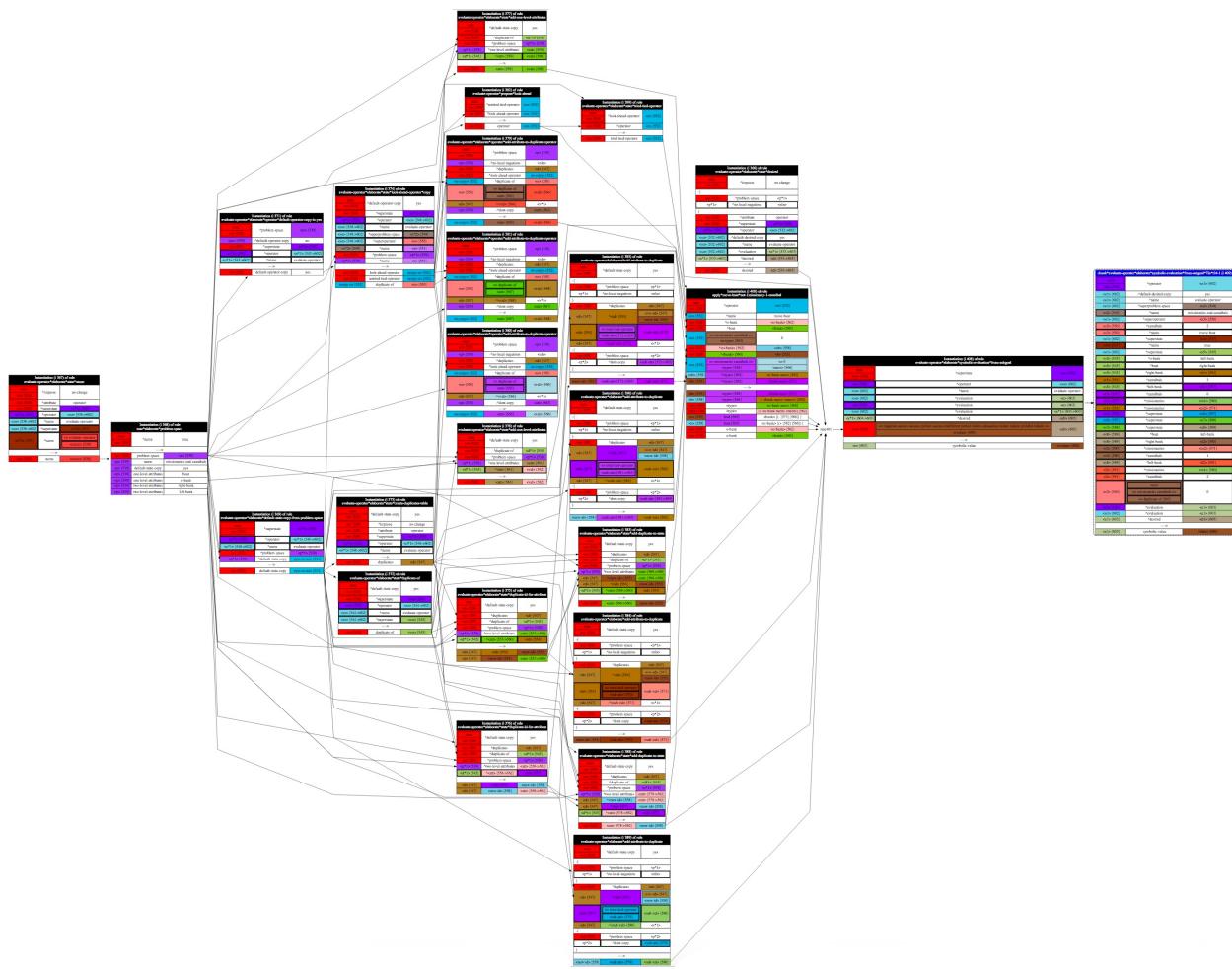


Figure 4.5: An explanation trace after identity analysis

While it's not readable in this figure, note that each identity is assigned a numeric ID. Both the explainer and the visualizer annotate elements of an explanation with the identity ID in square brackets. These numbers are simply syntactic sugar to ease debugging and make

Explanation-Based Chunking Algorithm			
Relevant OSK Tracking	Identity Analysis		Rule Formation
	Identity Assignment and Propagation	Identity Graph Manipulation	Conditions and Action Creation Constraint Enforcement Identity-based Generalization
Operator Decision	Rule Matches and Fires	Constraint Tracking	Condition Merging Condition Polishing Rule Repair and Validation Condition Re-ordering
		Operationality Analysis	Rule Formation
		Backtracing	
			Chunking
During Problem-Solving			

Figure 4.6: Note that the two rows on the bottom indicate when each component occurs during Soar's processing.

traces easier to understand. Underneath the hood, every test in a condition has a pointer to more complicated identity data structure that will be discussed in more detail in Section 4.4.1 on the identity graph.

4.3.2 The Five Main Components of Explanation-Based Chunking

1. Identity analysis

This component determines which variables in an explanation trace share the same identity. It also determines which identities are ineligible for variablization because they were tested against literal values in some rules.

Note that this component has two distinct mechanisms that occur at very different times. The first mechanism, identity propagation, occurs constantly while problem-solving in the substate. The second mechanism, identity graph manipulation, occurs during the learning episode.

2. Relevant operator selection knowledge tracking]

This component also occurs before the learning episode. Whenever an operator is selected, it analyzes what rule firings contributed necessary operator selection preferences and caches them in all rule instances that tests that operator.

3. Constraint tracking

This component keeps track of every value or relational constraint (e.g. $\text{<} \text{>} \text{ <} \text{x} \text{>} \text{, } \text{>} \text{=}$ 3.14, $\text{<<} \text{disjunction of constants} \text{>>}$) placed on the various variables that share an identity. It is used by the rule formation component to make sure that the learned rule only fires when all constraints required are met.

4. Operationality analysis

This component determines which conditions in an explanation trace tested working memory elements in a superstate. The rule formation component will use these conditions as a basis for the left-hand side of the chunk. While it does have a few key new differences, this is the one step that is similar to previous versions of chunking.

5. Rule Formation

The above four components performed the analysis that EBC needs to form a general but correct rule. This final component uses the results of that analysis to actually build the new rule. This is a complex component that has seven different stages. If a valid rule is created, Soar immediately adds the rule to production memory.

The following sections will describe each component in more detail.

4.4 What EBC Does Prior to the Learning Episode

While most of the work that explanation-based chunking performs occurs during the learning episode, i.e. after a rule in a substate fires and Soar detects that a result will be created, some critical aspects of the analysis it performs also occur prior to the learning episode, during problem-solving in the substate. The two points when that happens is when a rule fires in a substate and when an operator is selected in a substate.

4.4.1 Identity Assignment and Propagation

Each instantiation describes the working memory elements that matched each condition and the working memory elements and preferences that are created by each action. With the introduction of EBC, all instantiations now also store the underlying explanation behind each condition and action as defined by the original rule: which elements in conditions are variables and which ones are literal constants, which variables are the same variables, what constraints must be met on the values of each variable and any relationships between variables.

EBC uses this underlying logic to determine the identities of objects used during the problem-solving. Identities are not simply IDs. Each identity is a declarative object that describes a set of variables across multiple rule firings and the various properties they hold.

When an instantiation is created, EBC assigns all elements of every condition and action to an identity, creating new identities as necessary. Identities are created and propagated using the following rules:

1. If the same variable appears in multiple places in the same rule, it must be assigned the same identity.
2. The NULL Identity is assigned to any element with a literal value in the original rule.

3. A new identity is created and assigned for:

- (a) All right-hand side action elements that produce a new Soar identifier in the substate

These are also known as unbound RHS variables.

- (b) All variable elements of conditions that matched superstate WMEs

It is important to note that if two conditions both match the same superstate WME, each condition is considered independent. This means that each condition is assigned new identities for each of its elements and will produce its own condition in the final learned rule. This is a key way that EBC differs from previous versions of chunking.

4. An existing identity is propagated for:

- (a) Any condition element that matched a substate WME with existing identities

Each element is assigned the identity found in the corresponding element of the action of the rule that created that WME. This propagates identities forward through the explanation trace, which allows us to represent that the variable in the condition refers to the same object as the variable in the action of the other rule.

- (b) Any element that matches special working memory elements called **singletons** are assigned the same identity.

Singletons are working memory elements that are guaranteed to only have a single possible value in a state. The most important singleton is the local `^superstate` singleton, which is an architecturally created WME that links the substate to the superstate, for example `(S2 ^superstate S1)`. Since we know that it's impossible for there to be two superstate features in a state, all conditions that test that singleton WME will be assigned the same identities.

While there are a variety of built-in singletons for architecturally-created WMEs, users can also specify their own domain-specific singletons to eliminate unnecessary generality when learning. See section [4.7.3.2](#) for more information about user singletons. The full list of architecturally-created singletons can be found in the chunk command's help entry in section [9.4.1](#).

Note that rule 1 may conflict with other rules. For example, if a variable appears in two different conditions, then two different identities may propagate into each one of them. In such cases, rule 1 is always enforced and propagation is ignored. During the second phase of identity analysis, which occurs during the actual learning episode, EBC will re-examine all of the condition-action pairs as it performs a backward traversal of the explanation trace and fix the missing propagations. It does this by creating and manipulating an identity graph that can correctly incorporate all identity relationships.

4.4.2 Relevant Operator Selection Knowledge Tracking

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

In practice, however, it may make sense to design an agent so that search control does affect the correctness of search. Here are just 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 not be correct because important parts of the superstate that were tested by operator evaluation rules do not appear as conditions. The chunks would not accurately summarize the processing in that problem state. The tracking of **Relevant Operator Selection Knowledge** (ROSK) is a way to address this issue.

Relevant operator selection knowledge is the set of necessary operator evaluation preferences that led to the selection of an operator in a subgoal. As previously described, whenever Soar learns a rule, it recursively backtraces through rule instances to determine which conditions to include in the final chunk or justification. With the ROSK, not only does Soar backtrace through each rule instance that created a matched working memory element, but it also backtraces through every rule instance that created preferences in the ROSK for any operator that gave those matched WMEs o-support. By backtracing through that additional set of preferences at each step of the backtrace, an agent will create more specific chunks that incorporate the goal-attainment knowledge encoded in the operator evaluation rules.

Specifically, this component does two things:

1. When an operator is selected, it analyzes the operator preferences that led to the decision, and caches any operator selection knowledge that played a necessary role in the selection.

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

How Soar determines which of those preferences to include in the ROSK is determined by the preference semantics it uses to choose an operator. During the decision phase, operator preferences are evaluated in a sequence of seven steps or filters, in an effort to select a single operator, as described in Section 2.4.2. Each step, or filter, handles a specific type of preference. As the preference semantics are applied at each step to incrementally filter the candidate operators to a potential selected operator, EBC incrementally adds operator preferences to the ROSK based on the preferences that were instrumental in applying each filter. A more detailed explanation of the logic used at each step can be found in Section 4.6.15.

2. When an o-supported rule matches, EBC caches the operator's ROSK in the instantiation of that rule.

Since that selection knowledge was necessary to select the operator needed for the rule to match, chunking must backtrace through that knowledge. The operability analysis component uses the cached ROSK to do this and incorporate the necessary operator selection reasoning knowledge into the learned rule. For some types of agent designs, including operator selection knowledge is needed to ensure correctness.

4.5 What EBC Does During the Learning Episode

All of the previously discussed steps occurred during problem-solving in the substate as rules matched and operators were selected. It is worth noting that the analysis performed prior to the learning episode is persistent and can be shared across learning episodes. In other words, EBC can repeatedly re-use that analysis if it learns multiple chunks in the same substate.

Every time a rule fires in a substate, Soar checks to see if any of the working memory elements created by the rule qualify as results. This is when the actual learning episode begins.

4.5.1 Calculating the Complete Set of Results

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.

4.5.2 Backtracing and the Three Types of Analysis Performed

When learning a new rule, EBC performs a dependency analysis of the productions that fired in a substate – a process called backtracing. Backtracing works as follows. For each instantiated production that creates a subgoal result, backtracing examines the explanation trace to determine which working memory elements matched each condition. If the working memory element is local to the substate, then backtracing recursively examines the instantiation that created that condition’s matched working memory element. Thus, backtracing traces backwards through all rules that fired and created working memory elements that were used to produce a result.

If an instantiation being backtraced through tested a selected operator, EBC will backtrace through each instantiation that created a preference in that operator’s relevant operator selection knowledge set. This behavior is off by default and can be enabled with `chunk add-osk on` (See Section 9.4.1.5.)

Multiple components of EBC perform their work during backtracing: operability analysis, identity analysis and constraint tracking. The following sections will discuss what aspects of the agent’s problem-solving are analyzed during backtracing.

4.5.2.1 Operability Analysis

The traditional core function of chunking’s backtracing is to determine which conditions in the working memory trace tested working memory elements accessible to the superstate. These conditions will form the left-hand side of the rule.

The determination of which conditions to include is analogous to the concept of *operability* in explanation-based techniques. In classic EBL literature, operability is typically defined as nodes in the explanation trace that are “efficiently calculable”. In terms of Soar’s problem-state computational model, operability can be defined as any condition that tests knowledge linked to a superstate.

As EBC is backtracing through rules that fired in a substate, it collects all of these operational conditions. Once the entire explanation trace is traversed, the operability analysis will have determined exactly what superstate knowledge was tested during the process of creating a result, which it then uses as the basis for the left-hand side of the newly learned rule.

Note: Soar 9.6.0’s explanation-based approach has led to one key change to Soar’s operability analysis. In previous versions of chunking, chunking would never add two conditions to a chunk that matched the same superstate working memory element. This made sense because chunking was based on a generalization of the working memory trace. More than one condition that tested the same WME would be redundant. Explanation-based chunking, though, learns based on the reasoning within the original hand-written rules. Since the

reasoning behind each of the two conditions may be different even if they matched the same WME, EBC must always add both conditions. (Note that there are some exceptions. See Section 4.7.3.2 on superstate singletons and user singletons.)

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 super-state, 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.

4.5.2.2 Identity Analysis

The first phase of identity analysis, forward identity propagation, occurred as rules fired and instantiations were recorded. Unfortunately, forward propagation alone will not produce correct identities. We previously gave one reason why this is the case – conditions may have conflicting identities propagated forward – but there are other, more complicated reasons as well that are beyond the scope of this document. What is important to know is that a second phase of identity analysis will be performed during backtracing that will refine and correct the limitations of the initial forward propagation of identity. This second phase achieves these corrections by building an identity graph, which represent the identities involved during problem-solving, and manipulating it as it backtraces through the explanation trace.

The Identity Graph

The identity graph initially contains a node for each identity used in the explanation trace. Each node can have multiple edges that point to children identities and a single directed *join edge* that initially points back to itself. As the agent backtraces through the explanation trace, EBC will manipulate the identity graph based on the condition-action pairs it encounters.

1. Joining identities

If a condition matches an action with a conflicting identity, EBC performs a join operation between the two identities. This chooses one identity as the *joined identity* and points the join edges of the other identity and any previously joined identities to the new *joined identity*.

Note that any time EBC uses an element's identity, it is actually using the *joined identity*.

2. Literalizing identities

If a condition/action with a variable element matches an action/condition with a literal

element, EBC marks the identity as *literalized*. This means that any conditions in the final chunk that have elements with that identity will be considered to have the NULL identity, just like constants, and will not be variablized. Instead, the matched value will be used for that element.

4.5.2.3 Constraint Tracking

Our definition of operability is very clear and allows us to almost trivially determine which conditions we should include in a learned rule, but it does have one shortcoming: non-operational conditions, which are ones that don't test working memory elements in the superstate, can transitively place constraints on the values of variables in operational conditions that *will* appear in a chunk. If our learning algorithm does not include these constraints, the learned rule can apply to situations where the previous substate reasoning could not have occurred, which means that the learned rule is over-general.

To handle this limitation, ***EBC keeps track of all constraints found in non-operational conditions that it encounters while backtracing*** in the following manner:

- It stores constraints on the value a single identity, for example $\geq 0, < 23$.
- It stores relational constraints between two identities, for example $> \langle \text{min} \rangle, < \langle \text{max} \rangle$ or $\neq \langle \text{other} \rangle$.
- EBC stores all of these constraints based on the underlying identities, not the variables used. For example, if a variable $\langle \text{foo} \rangle$ had the constraint $\neq \langle \text{other} \rangle$, EBC would record that the variables that share the identity of $\langle \text{foo} \rangle$ cannot have the same value as variables that share the identity of $\langle \text{other} \rangle$.

4.5.3 Rule Formation

There are seven distinct, sequential stages to rule formation. The following sections will give a brief overview of each one.

4.5.3.1 Condition and Action Creation

This stage creates the basis for the left-hand and right-hand side of the rule. To create the initial conditions of the chunk, it copies all conditions in the explanation trace that were flagged as operational during backtracing. These initial conditions contain literal values for each element. To form the actions of the chunk, it creates copies of the actions that produced each of the result and all children of those results that came along for the ride.

Rule Formation
Conditions and Action Creation
Constraint Enforcement
Identity-based Generalization
Condition Merging
Condition Polishing
Rule Repair and Validation
Condition Re-ordering

Figure 4.7:

4.5.3.2 Enforcement of Constraints

This stage adds all constraints on non-operational conditions that were collected during backtracing. As previously described, each constraint is indexed in terms of the identity it constrains. So, *if the identity being constrained exists in one of the conditions of the learned rule*, EBC will enforce the constraint by adding a new test to that condition.

One situation in which attaching a constraint can be tricky occurs *when the constrained identity has been literalized but the constraint itself refers to an identity that has not been literalized*, for example $\{ > \langle x \rangle 3 \}$. While that constraint references a condition element that can only match a value of 3, the relationship between 3 and the identity of $\langle x \rangle$ must still hold (assuming $\langle x \rangle$ appears in a different element somewhere else in the rule.) Since these constraints still need to be enforced to ensure a correct rule, EBC will invert the constraint and attach it to a variable in another condition. In this example, it would add a < 3 to some other condition with an element that had $\langle x \rangle$'s identity.

4.5.3.3 Identity-Based Variabilization

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. At this point in the algorithm, all of the real work needed to determine the most general but correct variabilization has already been performed by the identity analysis component.

So, this step simply needs to replace all elements with non-NUL identities with variables, making sure that elements with the same joined identity are assigned the same variable. This step also makes sure to skip and elements with identities that have been flagged as *literalized*.

4.5.3.4 Merging Redundant Conditions

Any two conditions in the learned rule that share the same identities in all three elements can be combined. In such cases, it is logically impossible for those two conditions to match two different WMEs and cause the same rules to match in the substate. (If the two conditions were to match two different WMEs, at least one of the other rules in the explanation trace that had unified the two conditions would not have matched.) As a result, EBC can safely merge those two conditions without losing generality.

4.5.3.5 Polishing Conditions

EBC polishes the conditions of the learned rule by pruning unnecessary constraints on literalized elements and replacing multiple disjunction constraints with a single simplified disjunction.

1. **Merging disjunctions:** If an element in a condition has two disjunction tests, the constraints will be merged into a single disjunction that contains only the shared values. $\{ \langle\langle a b c \rangle\rangle \langle\langle b c d \rangle\rangle \langle x \rangle \}$ becomes $\{ \langle\langle b c \rangle\rangle \langle x \rangle \}$, because it is impossible for $\langle x \rangle$ to be either a or b . This will also eliminate any duplicate disjunctions.
2. **Throwing out unnecessary constraints:** If an element in a condition has been literalized but also has a literal constraint on its value, then the constraint is unnecessary and will be thrown out. For example, $\langle s \rangle \sim \text{value} \{ < 33 23 \}$ becomes $\langle s \rangle \sim \text{value} 23$.

4.5.3.6 Validating Rule and Repairing Unconnected Conditions

At this point, the rule is essentially formed. Chunking must now make sure that the learned rule is fully operational and can be legally added to production memory. A fully operational rule does not have any conditions or actions that are not linked to a goal state specified in the rule.

If an unconnected action or condition is found, EBC will attempt to repair the rule by adding new conditions that provide a link from a state that is already tested somewhere else in the rule to the unconnected condition or action.

To repair the rule, EBC performs a search through working memory to find the shortest path of working memory elements that lead from a state identifier in the rule to a WME with the identifier in the unconnected condition or action. A new condition is then added for every WME in that found path, which is then variablized.

Note that there may be multiple paths from a state to the unconnected identifier. EBC does a breadth-first search, so it will find one with the shortest distance.

4.5.3.7 Re-ordering Conditions

Since the efficiency of the Rete matcher depends heavily upon the order of a production's conditions, the chunking mechanism attempts to sort the chunk's conditions into the most favorable order. At each stage, the condition-ordering algorithm tries to determine which eligible condition, if placed next, will lead to the fewest number of partial instantiations when the chunk is matched. A condition that matches an object with a multi-valued attribute will lead to multiple partial instantiations, so it is generally more efficient to place these conditions later in the ordering. This is the same process that internally reorders the conditions in user-defined productions, as mentioned briefly in Section 2.3.1.

4.6 Subtleties of EBC

4.6.1 Relationship Between Chunks and Justifications

Chunks are closely related to another type of rule called a *justification*. Justifications are also created when a substate creates a result for a superstate, the difference being that justifications are only built when learning is off. These justifications are needed to decide whether the working memory elements in the result should get i-support or o-support in the superstate. To do that, Soar needs to determine whether any rules involved in the creation of the result tested the selected operator in the superstate, which is exactly the same type of analysis that chunking does.

As a result, Soar uses a limited version of the chunking algorithm to do that. It analyzes the substate problem-solving and learns a new, temporary rule, a “justification”, which is added to production memory. If this temporary rule tests an operator in the superstate, it gives the result o-support. (Note that when learning is on, a justification is not needed since the chunk will provide the correct support.)

Justifications use all the components described in the following sections and are even affected by the current chunk settings.¹ You can even print justifications out like other rules. The only differences between chunks and justifications are:

1. Every condition and action in a justification contain the literal values that matched. Justifications contain no variables.²
2. Justifications don't contain any of the value constraints that a chunk would have.
3. Justifications get removed from production memory as soon as their conditions no longer match.

4.6.2 Chunk Inhibition

If a newly learned chunk was immediately added to production memory, it would immediately match with the same working memory elements that participated in its creation. This can be problematic if the production's actions create new working memory elements. Consider the case where a substate proposes a new operator, which causes a chunk to be learned that also proposes a new operator. The chunk would immediately fire and create a preference for another new operator, which duplicates the operator preference that was the original result of the subgoal.

To prevent this, Soar uses **inhibition**. This means that each production that is built during chunking is considered to have already fired with an instantiation based on the exact set of

¹ Even though they don't contain variables, justifications can be over-general because they don't incorporate enough knowledge, for example, operator selection knowledge.

² Justifications can have variables in the negated conditions and negated conjunctions of conditions. They just don't have any variables in its positive conditions.

working memory elements used to create it.

Note that inhibition does not prevent a newly learned chunk from immediately matching other working memory elements that are present and creating a new instantiation.

4.6.3 Chunks Based on Chunks

When a problem has been decomposed into more than one substate, a single result can produce multiple chunks. This process is called *bottom-up chunking*. The first chunk is produced in the substate where the problem-solving that produced the result occurred. The next chunk is based on the implicit match of the first chunk in one of the higher level problem-spaces. If that match is lower than the state that the result is being returned to, Soar will backtrace through the chunk match and learn a second chunk (relative to the substate that the chunk matched in). This process continues until it learns a chunk that only creates working memory elements in the same state that it matched in.

4.6.4 Mixing Chunks and Justifications

If an agent is using the `only` or `except` setting, then justifications will be built in states where learning is disabled and chunks will be built in states where learning is enabled. In these situations, justifications also serve another purpose: they provide an explanation of the results for future learning episodes in states that do have learning on. EBC does this by retaining all of the extra information that chunks have but justifications do not, namely those extra tests and how things would have been variablized. This allows EBC to learn chunks from justifications as readily as it can from hand-written rules and other chunks.

When mixing justifications and chunks, users may want to set the explainer to record the learning episodes behind justifications. This allows one to examine the reasoning behind a justification just like you would a chunk, which may be important if that justification later participates in the formation a chunk. See Section 9.6.3 for more information about the explainer's settings.

4.6.5 Generality and Correctness of Learned Rules

Chunking is intended to produce the most general rule that is also correct.

Generality is a measure of the space of similar situations that a rule can apply to. A more general rule can be applied to a larger space of similar situations. A rule is considered over-general if it can apply to situations in which the original problem-solving would have never occurred.

Correctness is a requirement that the learned rule produces the exact same results that the original problem-solving would have produced. In other words, if we inhibited a correct chunk so that it did not fire, the agent should subgoal, execute the same substate reasoning

that it previously performed when learning the chunk, and produce the same results that the learned chunk produces.

Note that an over-general rule is an incorrect rule, but not all incorrect rules are over-general.

4.6.6 Over-specialization and Over-generalization

Explanation-based chunking was pursued to address the main limitation of traditional chunking: *over-specialized rules* that were very specific and could not be applied to many other situations. Specifically, EBC’s identity-based variabilization and constraint tracking/enforcement has eliminated the core source of this issue.

The nature of EBC’s algorithm does add two new situations in which rules may become over-specialized. Section 4.6.16 discusses how variables used in certain RHS functions need to be literalized to maintain correctness, which can cause overspecialization. Section 4.6.7 discusses how testing or augmenting a previous result creates non-operational rules that require repair, a process which may sometimes over-specialize a rule. Note that this situation can easily be avoided and, even when it does occur, may not add much unnecessary specificity to learned rules.

While over-specialization may no longer be a common problem, it is still possible to get over-general rules. Several of the sources of correctness issues listed in the next section can produce over-general rules in certain situations.

4.6.7 Previous Results and Rule Repair

An agent may learn a slightly over-specialized rule when EBC repairs a rule that has unconnected conditions, which are conditions that have an identifier that is not linked to one of the states referenced in the rule. Such rules are illegal and cannot be added to Soar’s production memory.

Rules that require repair are caused by substate problem-solving that tests or augments a previous result. A previous result is a working memory element that was originally created locally in the substate but then later became a result when a rule fired and connected it to the superstate. (At which point a chunk must have been learned.). If another substate rules later matches or augments such a previous result WME *using a path relative to the local substate*, then EBC will have problems. It will know that the WME is in the superstate – so conditions that test the WME are considered operational and augmentations on that identifier are considered results – but it won’t know where in the superstate that working memory is located is and how it should be referenced in the learned rule, because the problem solving referenced the result relative to the local substate.

As described in Section 4.5.3.6, EBC repairs the rule by adding new *grounding conditions* that provide a link from a state, which is tested somewhere else in the rule, to the unconnected condition or action. It does this by searching through working memory to find the shortest path from a state to the identifier behind the unconnected element. It then variabilizes those

conditions appropriately.

Since the conditions are based purely on what happened to be in working memory at that point and nothing in the explanation dictated that particular path found during the search, the learned rule may be over-specialized. The chunk will only match future situations where the previous result can be found on that same path. Fortunately, new chunks can be learned to ameliorate this. If a similar situation is encountered in the future, but with a different path to the unconnected element, the chunk won't fire, because the added grounding conditions won't match, which should cause the agent to subgoal and learn a similar chunk with a different set of grounding conditions.

Note that if an agent designer expects that the path to the previous result found by the search will always exist, a repaired rule should match just as generally as an unrepaired rule.

But if this is not the case, an agent designer can avoid this situation by modifying the rules that test or augment the substructure of a previous result. *If those rules are modified so that they match the previous results by referencing them relative to the superstate than the local substate, EBC will be able to create a valid rule without any repair.*

To detect when this is happening, use the `chunk stats` command. (See section 9.4.1.2) It will tell you if any of an agent's learned rules required repair. If you instruct the explainer to record the chunk, you can also see whether a specific chunk was repaired by looking at the chunk's individual stats

4.6.8 Missing Operator Selection Knowledge

If an agent uses rules that create operator preferences to choose amongst multiple operators in the substate, it is possible that the reasoning behind those rules needs to be incorporated in any rule learned. This topic is discussed in greater detail in Section 4.4.2.

EBC will incorporate relevant operator selection knowledge if you enable the chunk setting `add-osk`, which is off by default. (See Section 9.4.1.5.)

4.6.9 Generalizing Over Operators Selected Probabilistically

If the problem-solving in a substate involves operators that were selected probabilistically, chunking will not be able to summarize the agent's reasoning into a correct rule. For a rule to be correct, it must always produce the same result that the substate would have produced if the learned rule was not in production memory. Since a different operator could have been selected which could have resulted in different problem-solving, the substate could easily produce different results than any chunk learned in that substate.

Future versions of chunking will provide an option to prevent rules from forming when a probabilistically-selected operator was chosen during problem-solving. Until then, agent engineers can disable learning in states that involve such reasoning.

4.6.10 Collapsed Negative Reasoning

Over-general chunks can be created when conditions in the explanation trace test for the absence of a working memory elements in the substate. Since there is no clear way for chunking to generate a set of conditions that describe when a given working memory element would not exist in a substate, chunking can't represent that aspect of the problem-solving.

Chunking can include negated tests if they test for *the absence of working memory elements in the superstate, though.* So, the agent engineer can avoid using negated conditions for local substate data by either (1) designing the problem-solving so that the data that is being tested in the negation is already in the superstate or (2) making the data a result by attaching it to the superstate. This increases the number of chunks learned, but a negated condition of knowledge in the superstate can be incorporated correctly into learned rules.

Note that there are agent design patterns where local negations are perfectly safe to ignore, so Soar allows local negations by default. In some agents, they are common enough that turning the filter on prevents any rules from being learned.

If you suspect that a rule may be over-general because of locally negated condition, you can verify whether such a condition was encountered during backtracing by using the `chunk stats` command and `explain stats` command. See Sections 9.4.1.2 and 9.6.3.8 for more information.

If such chunks are problematic, turning off chunking's correctness filter `allow-local-negations` will force Soar to reject chunks whose problem-solving involved a local negation.

4.6.11 Problem-Solving That Doesn't Test The Superstate

Over-general 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 over-general 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.

In these cases, EBC will learn a chunk with no conditions, which it will reject. But the superstate result is still created by the substate rule that matched. If a new rule is learned that uses that result, it will be over-general since the rule does not summarize the reasoning that led to the result, namely that all operators were exhaustively applied.

The current solution to this problem is a bit of a hack. Soar allows an agent to signal to the architecture that a test for a substate is being made by testing for the `~quiescence t` augmentation of the subgoal. If this special test is found in the explanation trace, EBC will not build a chunk. 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.

4.6.12 Disjunctive Context Conflation

An incorrect rule can be learned when multiple rules fire in a substate that test different structures in the superstate *but create the same WME in the substate*. For example, there may be a rule that can match the superstate in several different ways, each time elaborating the local state with a WME indicating that at least one of these qualifying superstate WMEs existed. In such a situation, the rule would fire multiple times, but the result of the rule firings will be collapsed into creating a single WME in the substate.

If this WME is then tested to create a result on the superstate, the chunk that is subsequently created can produce different behavior than the substate would have. In the original subgoal processing, multiple matches produced one substate WME, but that one substate WME only created a single result in the superstate. The chunk on the other hand will match multiple times for each of the items that previously created the substate WME. And then, each one of those matches will create its own distinct result in the superstate. Since this is different behavior than the original substate, this rule would be considered incorrect.

If it were possible, EBC should learn a *disjunctive conjunctive condition*, with each disjunction being the superstate conditions tested by each substate rule that had previously created the substate WME that was repeatedly asserted. This is why this potential source of incorrect rules is called *disjunctive context conflation*.

If this type of reasoning is needed, agents can move the conflating WME to the superstate. The rule learned would then produce only one result regardless of the number of rules that repeatedly created that WME.

4.6.13 Generalizing knowledge retrieved from semantic or episodic memory

Generalizing problem-solving based on knowledge recalled from an external memory system can be problematic for three main reasons.

1. Knowledge can change after the learning episode

Semantic knowledge can be modified by the agent. Different semantic knowledge can effect different problem-solving, in which case a rule based on the original problem-solving would be incorrect.

2. Justification for a memory recall is opaque to agent

EBC does not have access to the reasoning behind why a piece of knowledge was recalled from a memory system. For example, consider the case of a semantic memory that is recalled because it has the highest level of activation at a particular time. In a future situation, the same semantic memory may not be the most active, in which case

something else would be recalled and different problem-solving could occur. Because of that possibility, the original rule is not guaranteed to produce the same result and hence has the potential to be incorrect. (Note that this can also occur with episodic memory queries.)

3. Knowledge from semantic or episodic memory recalled directly into the substate is considered local

To understand why this is a problem, remember that a chunk's conditions are based on the conditions in the explanation trace that tested knowledge linked to a superstate. (See section 4.5.2.1 for more information.) If semantic or episodic memory is recalled directly into the substate, then any conditions that test that recalled knowledge is considered local to the substate and will not be included as a condition in the chunk. So, even though the substate reasoning required some piece of semantic knowledge to exist, the chunk will not require it. And, since the learned rule *is not* incorporating some of the reasoning and constraints that involved the recalled knowledge, the rule may be over-general.

To avoid this situation, an agent can retrieve the knowledge in a higher-level state rather than the substate in which the rule is learned.

4.6.14 Learning from Instruction

Note that some agent designs, for example an agent that learns by instruction, can take advantage of the fact that knowledge recalled from semantic or episodic memory directly into the substate is considered local. For such agents, a rule that is directly dependent on the instructions being in working memory would be useless. The agent would need to get the instruction every time it wanted to perform the task again, defeating the purpose of learning by instruction.

One technique that can be used to produce a more general rule which is not directly dependent on the instruction being in working memory is to first store the instructions in semantic or episodic memory. When the agent is in a substate that it wants to learn a rule based on the instructions, it recalls the instructions from semantic or episodic memory directly into the substate. Because that knowledge is not linked to the superstate, any rules learned in that substate will not be directly dependent on the existence of the instructions.

Since conditions that test the recalled knowledge are not incorporated into the learned rule, it is very easy to learn over-general chunks. To avoid this, any substate rules which test recalled knowledge must also test superstate structures that correspond to the recalled knowledge. Doing so removes the need for the instructions to exist while avoiding over-generality by ensuring that structures in the superstate corresponding to those instructions are still being tested. Those conditions that test superstate WMEs will be generalized and included in the chunk, but the undesired portion of the reason that they matched will not be, namely the fact that the superstate knowledge corresponded to recalled instructions.

4.6.15 Determining Which OSK Preferences are Relevant

The following outline describes the logic that happens at each step. For a more detailed description of the various filters (but not the ROSK) see Section 2.4.2 on page 21. Note that depending on the set of preferences being processed, impasses may occur at some of these stages, in which case, no operator is selected and the ROSK is emptied. Moreover, if the candidate set is reduced to zero or one, the decision process will exit with a finalized ROSK. For simplicity's sake, this explanation assumes that there are no impasses and the decision process continues.

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

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

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

Best Filter Add any best preferences for remaining candidates to the ROSK.

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

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

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

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

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

Note also that operator selection knowledge is not tracked and incorporated into chunks by default. The setting must be turned on via the chunk command's `add-osk` setting. See Section 9.4.1 on page 232 for more information.

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

4.6.16 Generalizing Knowledge From Math and Other Right-Hand Side Functions

Explanation-based chunking introduces the ability to learn more expressive rules whose actions perform arbitrary right-hand side functions with variablized arguments.

It is important to note that this ability is limited. EBC can only learn rules with generalized RHS functions in its actions when the rule that created the result contained a RHS function. In many cases, RHS functions will be used in the intermediate rule firings in the explanation trace. Not only will these intermediate RHS function not appear in the chunk, but any chunk learned based on their output will become more specific. This is one of the sources of over-specialization referenced in section 4.6.6 on over-specialization.

RHS function calls in intermediate rule firings are a challenge for EBC to deal with because the problem-solving may have placed constraints on the intermediate results that cannot be represented in a single Soar rule.

For example, consider the case of one rule that used a RHS function to add two numbers. Now consider another rule that matched the output of the RHS function, but only if it was less than 5. If the second rule matched, it would return the total as a result. How could we encode the reasoning of those two rules into one rule? Since Soar's production syntax does not allow using RHS function as constraints in conditions, there is no way to insure that the two numbers add up to something less than 5 in a single rule. This is why RHS functions in intermediate rule firings can cause over-specialization.

Because the chunk's conditions can't represent constraints on the output of intermediate RHS functions, EBC must literalize both the identities of the variables that appear as arguments to the intermediate RHS function, as well as the identities in any conditions that test the output of the RHS function. That fixes the value of the RHS function and guarantees that any constraints in conditions that test the output of that RHS function will be met. While this will make the learned rule more specific, it will also ensure that the rule is correct.

4.6.17 Situations in which a Chunk is Not Learned

Soar learns a chunk every time a subgoal produces a result, unless one of the following conditions is true:

1. **Chunking is off**

This corresponds to the command `chunk never`. See Section 9.4.1 on page 232 for details of `chunk` and how to turn chunking on or off.

2. **Chunking was only enabled for some states, and the subgoal in question is not one of them**

When chunking is enabled via the `only` or `except` command, the agent must specify which states learning either occurs in or doesn't occur in, respectively. For the `except` setting, Soar will learn rules in all states in which a `dont-learn` RHS production action was not executed. Similarly, for the `only` setting, Soar will learn rules in all states where a `force-learn` RHS production action was executed. See Section 3.3.6.7 on page 81 for more information.

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

3. **The chunk learned is a duplicate of another 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 problem-solving in the substate violated one of the enabled correctness guarantee filters**

During the development of explanation-based chunking, we have developed a list of possible causes of incorrect chunks. EBC's correctness guarantee filters detect when those situations occur and prevents a chunk from being learned.

For example, the `allow-local-negations` filter will prevent a rule from being formed if the problem-solving that led to the result was dependent on a condition that tested whether a subgoal WME doesn't exist. Since there is no practical way to determine why a piece of knowledge doesn't exist, testing a local negation can result in an over-general and incorrect chunk. See Section 4.7.3.1 on page 119 for more information.

Note that correctness filters have not yet been implemented for all the identified potential sources of correctness issues.

5. **The chunking option bottom-only is on and a chunk was already built in the bottom subgoal that generated the results**

With `bottom-only` chunking, 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 . See Section 9.4.1 on page 232 for details of `chunk` used with the `bottom-only` setting.

6. The problem-solving that led to the result contained a condition that tested the architecturally-created <state> ^quiescence t augmentation

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 4.6.11 on page 112). 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.

7. The problem-solving in the substate did not test any knowledge in the superstate

In these cases, the chunk learned does not have any conditions and is not a legal production. *Note that this creates an unusual persistence issue for any results that came out of the substate.* Since a justification or chunk was not learned, there is no rule in the superstate that can provide either i-support or o-support for the result that came out of the substate. Consequently, those result WMEs will be completely dependent on the rules that fired within the substate. So, when the substate is removed, those results will also be removed.

4.7 Usage

4.7.1 Overview of the chunk command

Chunk Commands and Settings	
? help	Print this help listing
timers	Timing statistics (no args to print stats)
stats	Print stats on learning that has occurred
----- Settings -----	
ALWAYS never only except	When Soar will learn new rules
bottom-only	Learn only from bottom substate
naming-style	Simple names or rule-based name
max-chunks	Max chunks that can be learned (per phase)
max-dupes	Max duplicate chunks (per rule, per phase)
----- Debugging -----	
interrupt	Stop Soar after learning from any rule
explain-interrupt	Stop Soar after learning explained rule
warning-interrupt	Stop Soar after detecting learning issue
----- Fine Tune -----	
singleton	Print all WME singletons
singleton <type> <attribute> <type>	Add a WME singleton pattern
singleton -r <type> <attribute> <type>	Remove a WME singleton pattern
----- EBC Mechanisms -----	
add-ltm-links	Recreate LTM links in original results
add-osk	Incorporate operator selection knowledge
merge	Merge redundant conditions
lhs-repair	Add grounding conds for unconnected LHS
rhs-repair	Add grounding conds for unconnected RHS
user-singletons	Use domain-specific singletons
----- Correctness Guarantee Filters -----	
allow-local-negations	Allow rules to form that...
allow-local-negations [ON off]	...used local negative reasoning
allow-opaque*	...used knowledge from a LTM recall

<code>allow-missing-osk*</code> <code>allow-uncertain-operators*</code>	<code>[ON off]</code> ...tested operators selected using OSK <code>[ON off]</code> ...tested probabilistic operators
--	---

* disabled

See Section [9.4.1](#) for more detailed information about the `chunk` command's settings.

4.7.2 Enabling Procedural Learning

By default, explanation-based chunking is off.

- To turn on chunking: `chunk always`
- To turn off chunking: `chunk never`

In real world agents, there may be certain problem spaces in which you don't want your agent to learn rules. Chunking has a mechanism to allow agents to dynamically specify the states in which rules are learned.

- To turn off chunking in all states except ones manually flagged on:
 - ★ Use `chunk only` setting.
 - ★ Design an agent rule that executes the RHS action `force-learn`, which only matches in states in which you want to learn rules.
- To turn on chunking in all states except ones manually flagged off:
 - ★ Use `chunk except` setting.
 - ★ Design an agent rule that executes the RHS action `dont-learn`, which only matches in states in which you don't want to learn rules.

Depending on your agent design, you may want to consider enabling the `add-osk` option. As of Soar 9.6.0, EBC does not incorporate operator selection knowledge into learned rules by default, since there is a performance cost and not all agents designs require its inclusion. You may want to enable this option if your agent has rules that test knowledge in the superstate to create operator preferences in the substate. See section [4.4.2](#) on page [101](#) for more information about learning and operator selection knowledge.

See Section [9.4.1](#) on page [232](#) for more information about using the `chunk` command to enable and disable procedural learning.

4.7.3 Fine-tuning What Your Agent Learns

4.7.3.1 Prohibiting known sources of correctness issues

It is theoretically possible to detect nearly all of the sources of correctness issues and prevent rules from forming when those situations are detected. In Soar 9.6.0, though, only one filter

is available, `allow-local-negations`. Future versions of Soar will include more correctness filters.

Note that it is still possible to detect that your agent may have encountered a known source of a correctness issue by looking at the output of the `chunk stats` command. It has specific statistics for some of the sources, while others can be gleaned indirectly. For example, if the stats show that some rules required repair, you know that your agent testing or augmenting a previous result in a substate.

4.7.3.2 Using singletons to simplify a rule's conditions

Unlike previous versions of chunking, EBC adds all conditions that tested superstate knowledge to a chunk, regardless of whether another condition already tested that working memory element. This means that EBC can sometimes produce learned rules with seemingly duplicate conditions. While these conditions are logically correct, they may be redundant because the nature of the domain may make it impossible for the two conditions to match different working memory elements. For example, in the blocks-world domain, the fact that there can be only one gripper in the world means that having multiple conditions testing for a gripper would be redundant.

Soar allows agents to specify such known domain characteristics, which EBC will then use to create better rules that don't have such unnecessary conditions. We call any working memory element that is guaranteed to only have a single possible value at any given time, a *singleton*. If EBC encounters two different conditions in the backtrace that both test the same superstate WME that matches a user singleton pattern, it will merge the two conditions. As described in Section 4b, there are several architectural singleton's that EBC already knows about. To specify patterns for domain-specific singletons, the `chunk singleton` command can be used.

See Section 9.4.1 for more information about the `chunk singleton` command.

4.7.4 Examining What Was Learned

4.7.4.1 Printing and Traces

Printing Rules:

- To print all chunks learned:
`print --chunks` or `print -c`
- To print all justifications learned (and still matching):
`print --justifications` or `print -j`
- To print a rule or justification:
`print <rule-name>`

For more information on print, see section [9.3.1](#) on page [215](#).

Trace Messages:

- To print when new rules are learned (just the name):
`trace --learning 1` or `trace -l 1`
- To print when new rules are learned (the full rule):
`trace --learning 2` or `trace -l 2`
- To print a trace of the conditions as they are collected during backtracing:
`trace --backtracing` or `trace -b`
- To print warnings about chunking issues detected while learning:
`trace --chunk-warnings` or `trace -C`
- To print when learned chunks match and fire:
`trace --backtracing` or `trace -b`

For more information on trace, see section [9.6.1](#) on page [258](#).

Note that the most detailed information about why a particular rule was learned can be acquired using the explain mechanism as described in section [9.6.3](#) on page [269](#). That is highly recommended over printing the backtracing trace messages.

4.7.4.2 Chunking Statistics

Chunking automatically compiles various statistics about the procedural rule learning that an agent performs. To access these stats, use the command `chunk stats` or `stats -l`

Explanation-Based Chunking Statistics	
Substates analyzed	0
Rules learned	0
Justifications learned	0
<hr/>	
Work Performed	
Number of rules fired	0
Number of rule firings analyzed during backtracing	0
Number of OSK rule firings analyzed during backtracing	0
Number of rule firings re-visited during backtracing	0
Conditions merged	0
Disjunction tests merged	0
- Redundant values	0

- Impossible values eliminated	0
Operational constraints	0
Non-operational constraints detected	0
Non-operational constraints enforced	0

Identity Analysis

Identities created	0
Distinct identities in learned rules	0
Identity propagations	0
Identity propagations blocked	0
Identity propagations from local singleton	0
Identities joined	0
- To unify two identities propagated into same variable	0
- To unify two conditions that tested a superstate singleton	0
- To connect an child result (result in rule had children WMEs)	0
Identities literalized	0
- Condition with variable matched a literal RHS element	0
- Condition with variable matched a RHS function	0
- Condition with literal value matched a RHS variable	0
- Variable used in a RHS function	0

Potential Generality Issues Detected

Rules repaired that had unconnected conditions or actions	0
Extra conditions added during repair	0

Potential Correctness Issues Detected

Chunk used negated reasoning about substate	0
Chunk tested knowledge retrieved from long-term memory	0
Justification used negated reasoning about substate	0
Justification tested knowledge retrieved from long-term memory	0

Learning Skipped or Unsuccessful

Ignored duplicate of existing rule	0
Skipped because problem-solving tested ^quiescence true	0
Skipped because no superstate knowledge tested	0
Skipped because MAX-CHUNKS exceeded in a decision cycle	0
Skipped because MAX-DUPES exceeded for rule this decision cycle	0

Note that similar statistics for a specific learned rule can be acquired using the explain mechanism as described in section 9.6.3 on page 269.

4.7.4.3 Interrupting Execution To Examine Learning

- To stop Soar after each successful learning episode:
`chunk interrupt on`
- To stop Soar after detecting any learning issue:
`chunk warning-interrupt on`
- To stop Soar after learning a rule that the explainer recorded:
`chunk explain-interrupt on`

For more information about how to record when a specific rule is learned, see section [9.6.3](#) on page [269](#) that describes the `explain` mechanism.

4.8 Explaining Learned Procedural Knowledge

While explanation-based chunking makes it easier for people to now incorporate learning into their agents, the complexity of the analysis it performs makes it far more difficult to understand how the learned rules were formed. The explainer is a new module that has been developed to help ameliorate this problem. The explainer allows you to interactively explore how rules were learned.

When requested, the explainer will make a very detailed record of everything that happened during a learning episode. Once a user specifies a recorded chunk to "discuss", they can browse all of the rule firings that contributed to the learned rule, one at a time. The explainer will present each of these rules with detailed information about the identity of the variables, whether it tested knowledge relevant to the superstate, and how it is connected to other rule firings in the substate. Rule firings are assigned IDs so that user can quickly choose a new rule to examine.

The explainer can also present several different screens that show more verbose analyses of how the chunk was created. Specifically, the user can ask for a description of (1) the chunk's initial formation, (2) the identities of variables and how they map to identity sets, (3) the constraints that the problem-solving placed on values that a particular identity can have, and (4) specific statistics about that chunk, such as whether correctness issues were detected or whether it required repair to make it fully operational.

Finally, the explainer will also create the data necessary to visualize all of the processing described in an image using the new 'visualize' command. These visualization are the easiest way to quickly understand how a rule was formed.

Note that, despite recording so much information, a lot of effort has been put into minimizing the cost of the explainer. When debugging, we often let it record all chunks and justifications formed because it is efficient enough to do so.

Use the `explain` command without any arguments to display a summary of which rule firings

the explainer is watching. It also shows which chunk or justification the user has specified is the current focus of its output, i.e. the chunk being discussed.

Tip: This is a good way to get a chunk id so that you don't have to type or paste in a chunk name.

```
=====
Explainer Summary
=====
Watch all chunk formations Yes
Explain justifications Nof
Number of specific rules watched 0

Chunks available for discussion:
chunkx2*apply2 (c 14)
chunk*apply*o (c 13)
chunkx2*apply2 (c 12)
chunk*apply*d (c 11)
chunkx2*apply2 (c 6)
chunk*apply* (c 15)
chunkx2*apply (c 8)
chunk*apply*c (c 5)
chunkx2*apply (c 10)
chunk*apply (c 1)

* Note: Printed the first 10 chunks. 'explain list' to see other 6 chunks.

Current chunk being discussed: chunk*apply*down-gripper(c 3)
```

`explain chunk [<chunk id> | <chunk name>]`

This command starts the explanation process by specifying which chunk's explanation trace you want to explore.

Tip: Use the alias `c` to quickly start discussing a chunk, for example:

```
soar % c 3
Now explaining chunk*apply*move-gripper-above*pass*top-state*OpNoChange*t6-1.
- Note that future explain commands are now relative
  to the problem-solving that led to that chunk.

Explanation Trace           Using variable identity IDs           Shortest Path to Result Instantiation
sp {chunk*apply*move-gripper-above*pass*top-state*OpNoChange*t6-1
1:   (<s1> ^top-state <s2>)          ([140] ^top-state [162])
2:   -{
3:     (<s1> ^operator <o1>)          ([140] ^operator [141])
4:     (<o1> ^name evaluate-operator)  ([141] ^name evaluate-operator)
5:   }
6:   (<s2> ^gripper <g1>)          ([162] ^gripper [156])
7:   (<g1> ^position up)            ([156] ^position up)
8:   (<g1> ^holding nothing)        ([156] ^holding nothing)
9:   (<g1> ^above <i1>)            ([156] ^above [157])
10:  (<i1> ^io <i2>)              ([162] ^io [163])
11:  (<i2> ^output-link <i1>)      ([163] ^output-link [164])
12:  (<i1> ^gripper <g2>)          ([164] ^gripper [165])
13:  (<s2> ^clear { <> <i1> <b1> })
14:    (<s1> ^operator <o1>)          ([162] ^clear { <> [161] [161] })
15:    (<o1> ^moving-block <b1>)    ([140] ^operator [149])
16:    (<b1> ^name pick-up)         ([149] ^moving-block [161])
17:    -->
18:    (<g2> ^name pick-up)          ([149] ^name pick-up)
19:  -->
20:  (<g2> ^command move-gripper-above +)  ([165] ^command move-gripper-above +)
21:  (<g2> ^destination <c1> +)       ([165] ^destination [161] +)
```

explain formation

Once you specify a rule to explain, this will be one of the first commands you issue. `explain formation` provides an explanation of the initial rule that fired which created a result. This is what is called the ‘base instantiation’ and is what led to the chunk being learned. Other rules may also be base instantiations if they previously created children of the base instantiation’s results. They also will be listed in the initial formation output.

```
soar % explain formation
-----
The formation of chunk 'chunk*apply*move-gripper-above*pass*top-state*OpNoChange*t6-1' (c 1)
-----
Initial base instantiation (i 31) that fired when apply*move-gripper-above*pass*top-state matched at level 3 at time 6:
Explanation trace of instantiation # 31          (match of rule apply*move-gripper-above*pass*top-state at level 3)
(produced chunk result)
Identities instead of variables      Operational    Creator
1:  (<s> ^operator <op>)          ([159] ^operator [160])      No        i 30 (pick-up*propose*move-gripper-above)
2:  (<op> ^name move-gripper-above)  ([160] ^name move-gripper-above)  No        i 30 (pick-up*propose*move-gripper-above)
3:  (<op> ^destination <des>)      ([160] ^destination [161])    No        i 30 (pick-up*propose*move-gripper-above)
4:  (<s> ^top-state <t*1>)       ([159] ^top-state [162])    No        i 27 (elaborate*state*top-state)
5:  (<t*1> ^io <i*1>)           ([162] ^io [163])        Yes      Higher-level Problem Space
6:  (<i*1> ^output-link <o*1>)   ([163] ^output-link [164])  Yes      Higher-level Problem Space
7:  (<o*1> ^gripper <gripper>)  ([164] ^gripper [165])     Yes      Higher-level Problem Space
-->
1:  (<gripper> ^command move-gripper-above +)  ([165] ^command move-gripper-above +)
2:  (<gripper> ^destination <des> +)      ([165] ^destination [161] +)
-----
This chunk summarizes the problem-solving involved in the following 5
rule firings:
i 27 (elaborate*state*top-state)
i 28 (elaborate*state*operator*name)
i 29 (pick-up*elaborate*desired)
i 30 (pick-up*propose*move-gripper-above)
i 31 (apply*move-gripper-above*pass*top-state)
```

explain instantiation <instantiation id>

This command prints a specific instantiation in the explanation trace. This lets you browse the instantiation graph one rule at a time. This is probably one of the most common things you will do while using the explainer.

Tip: Use the alias `i <instantiation id>` to quickly view an instantiation, for example:

```
soar % i 30
Explanation trace of instantiation # 30          (match of rule pick-up*propose*move-gripper-above at level 3)
- Shortest path to a result: i 30 -> i 31
Identities instead of variables      Operational    Creator
1:  (<s> ^name pick-up)          ([152] ^name pick-up)      No        i 28 (elaborate*state*operator*name)
2:  (<s> ^desired <d*1>)        ([152] ^desired [153])    No        i 29 (pick-up*elaborate*desired)
3:  (<d*1> ^moving-block <mblock>)  ([153] ^moving-block [154])  No        i 29 (pick-up*elaborate*desired)
4:  (<s> ^top-state <t*2>)       ([152] ^top-state [155])    No        i 27 (elaborate*state*top-state)
5:  (<t*2> ^clear <mblock>)      ([155] ^clear [154])      Yes      Higher-level Problem Space
6:  (<ts> ^gripper <g>)         ([155] ^gripper [156])    Yes      Higher-level Problem Space
7:  (<g> ^position up)          ([156] ^position up)      Yes      Higher-level Problem Space
8:  (<g> ^holding nothing)      ([156] ^holding nothing)  Yes      Higher-level Problem Space
9:  (<g> ^above { <> <mblock> <a*1> })  ([156] ^above { <> [154] [157] })  Yes      Higher-level Problem Space
-->
1:  (<s> ^operator <op1> +)      ([152] ^operator [158] +)
2:  (<op1> ^name move-gripper-above +)  ([158] ^name move-gripper-above +)
3:  (<op1> ^destination <mblock> +)    ([158] ^destination [154] +)
```

explain [explanation-trace | wm-trace]

In most cases, users spend most of their time browsing the explanation trace. This is where

chunking learns most of the subtle relationships that you are likely to be debugging. But users will also need to examine the working memory trace to see the specific values matched.

To switch between traces, you can use the `explain e` and the `explain w` commands.

Tip: Use the aliases `et` and `wt` to quickly switch between traces.

```
soar % explain w
Working memory trace of instantiation # 30      (match of rule pick-up*propose*move-gripper-above at level 3)
1:   (S9 ^name pick-up)                      No      i 28 (elaborate*state*operator*name)
2:   (S9 ^desired D6)                        No      i 29 (pick-up*elaborate*desired)
3:   (D6 ^moving-block B3)                   No      i 29 (pick-up*elaborate*desired)
4:   (S9 ^top-state S1)                     No      i 27 (elaborate*state*top-state)
5:   (S1 ^clear B3)                         Yes     Higher-level Problem Space
6:   (S1 ^gripper G2)                       Yes     Higher-level Problem Space
7:   (G2 ^position up)                      Yes     Higher-level Problem Space
8:   (G2 ^holding nothing)                  Yes     Higher-level Problem Space
9:   (G2 ^above { <> B3 T1 })              Yes     Higher-level Problem Space
-->
1:   (S9 ^operator O9) +
2:   (O9 ^name move-gripper-above) +
3:   (O9 ^destination B3) +
```

explain constraints

This feature lists all constraints found in non-operational constraints of the explanation trace. If these constraints were not met, the problem-solving would not have occurred.

This feature is not yet implemented. You can use `explain stats` to see if any transitive constraints were added to a particular chunk.

explain identity

`explain identity` will show the mappings from variable identities to identity sets. If available, the variable in a chunk that an identity set maps to will also be displayed.

By default, only identity sets that appear in the chunk will be displayed in the identity analysis. To see the identity set mappings for other sets, change the `only-chunk-identities` setting to `off`.

```
soar % explain identity
=====
-          Variabilization Identity to Identity Set Mappings      -
=====

--- NULL Identity Set ---

The following variable identities map to the null identity set and will
not be generalized: 282 301 138 291 355 336 227 309 328 318 128 218 345

--- How variable identities map to identity sets ---

Variabilization IDs      Identity      CVar      Mapping Type

Instantiation 36:
  125 -> 482           | IdSet 12 | <s>      | New identity set
  126 -> 493           | IdSet 11 | <o>      | New identity set
Instantiation 38:
Instantiation 41:
```

```

146 -> 482      | IdSet 12 | <s>      | New identity set
147 -> 493      | IdSet 11 | <o>      | New identity set
Instantiation 42:
151 -> 180      | IdSet 1  | <ss>     | New identity set
149 -> 482      | IdSet 12 | <s>      | New identity set
150 -> 493      | IdSet 11 | <o>      | New identity set
307 -> 180      | IdSet 1  | <ss>     | Added to identity set
187 -> 180      | IdSet 1  | <ss>     | Added to identity set
334 -> 180      | IdSet 1  | <ss>     | Added to identity set
173 -> 180      | IdSet 1  | <ss>     | Added to identity set
280 -> 180      | IdSet 1  | <ss>     | Added to identity set
Instantiation 53:
219 -> 489      | IdSet 15 | <b>      | New identity set
Instantiation 61:
Instantiation 65:
319 -> 492      | IdSet 20 | <t>      | New identity set

```

explain stats

Explain's **stat** command prints statistics about the specific chunk being discussed. This is a good way to see whether any generality or correctness issues were detected while learning that rule.

```
=====
Statistics for 'chunk*apply*move-gripper-above*pass*top-state*OpNoChange*t6-1' (c 1):
=====

Number of conditions          14
Number of actions              2
Base instantiation             i 31 (apply*move-gripper-above*pass*top-state)
=====

===== Generality and Correctness =====

Tested negation in local substate    No
LHS required repair                 No
RHS required repair                 No
Was unrepairable chunk              No
=====

===== Work Performed =====

Instantiations backtraced through   5
Instantiations skipped              6
Constraints collected              1
Constraints attached               0
Duplicates chunks later created    0
Conditions merged                  2
```

After-Action Reports The explainer has an option to create text files that contain statistics about the rules learned by an agent during a particular run. When enabled, the explainer will write out a file with the statistics when either Soar exits or a **soar init** is executed. This option is still considered experimental and in beta.

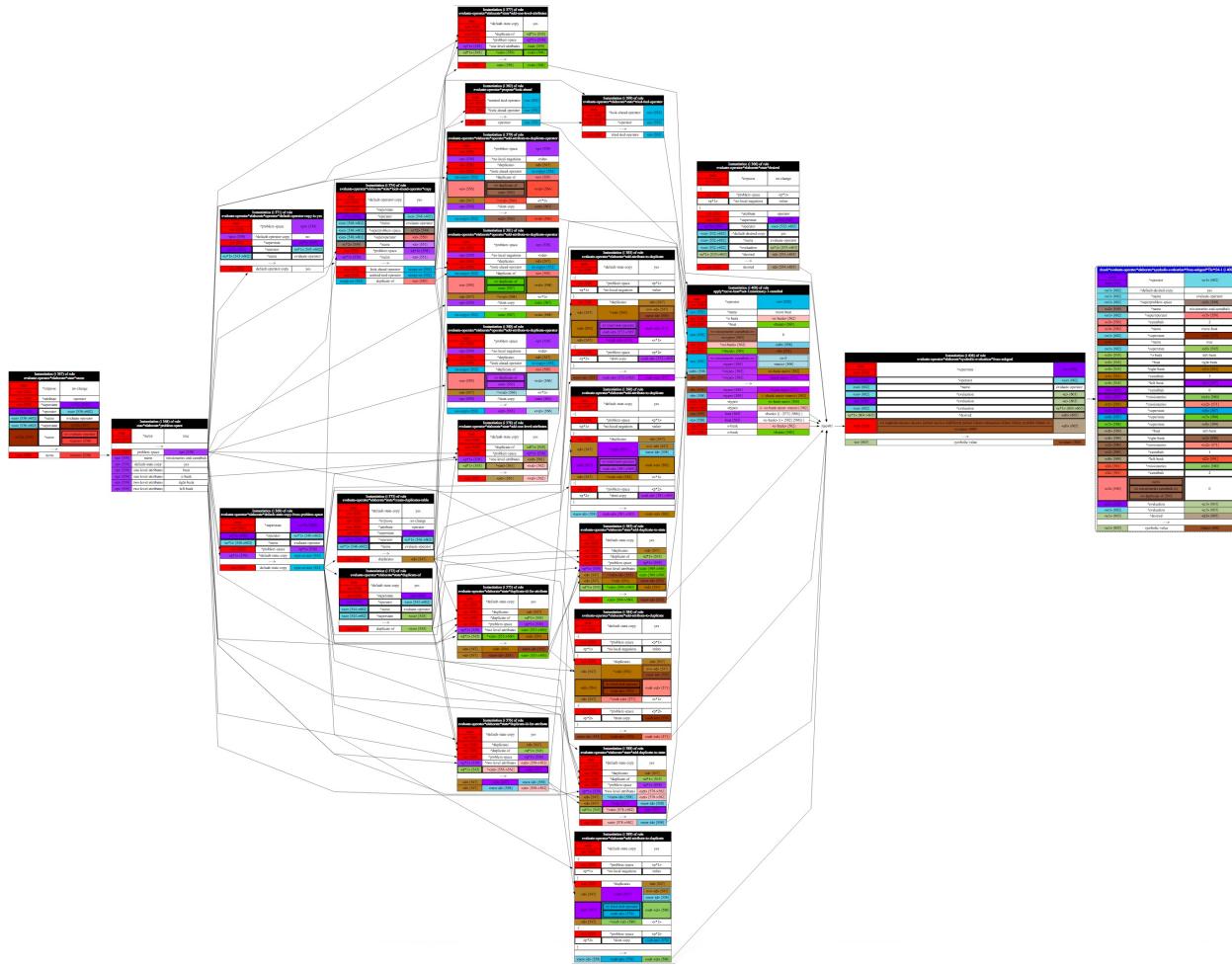


Figure 4.8: A colored visualization of an explanation trace

4.9 Visualizing the Explanation

The `visualize` command can generate two graphical representations of the analysis that chunking performed to learn a rule. While the explainer provides more data, these images are the easiest and most effective ways to quickly understand how a chunk was formed, especially for particularly complex chunks. The visualizer can create two types of chunking-related images:

1. An image that shows the entire instantiation graph at once and how it contributed to the learned rule.

Use the command `visualize ebc analysis` to create a very informative graph that shows all rules that fired in a substate with arrows that indicate dependencies between actions in one rule and conditions in others. In addition to all of the dependencies between the rules that fired, this visualization also shows which conditions in the instantiations tested knowledge in the superstate and hence became the basis for a condition in the final learned rule. Finally, the individual elements in the explanation

are color-coded to show which variables share the same identity.

2. An image that shows the graph of how variable identities were combined.

Use the `visualize identity graph` to create a graph that shows how identities were used to determine the variablization of a learned rule. This shows all identities found in the chunk and how the identity analysis joined them based on the problem-solving that occurred. This can be useful in determining why two elements were assigned the same variable.

Note that Soar will automatically attempt to launch a viewer to see the image generated. If you have an editor that can open graphviz files, you can have Soar launch that automatically as well. (Such editors allow you to move things around and lay out the components of the image exactly as you want them.) Your operating system chooses which program to launch based on the file type.

For the visualizer to work, you must have Graphviz and DOT installed, which are free third-party tools, and both must be available on your path. To date, the visualizer has only been tested on Mac and Linux. It is possible that certain systems may not allow Soar to launch an external program.

Chapter 5

Reinforcement Learning

Soar has a reinforcement learning (RL) mechanism that tunes operator selection knowledge based on a given reward function. This chapter describes the RL mechanism and how it is integrated with production memory, the decision cycle, and the state stack. We assume that the reader is familiar with basic reinforcement learning concepts and notation. If not, we recommend first reading *Reinforcement Learning: An Introduction* (1998) by Richard S. Sutton and Andrew G. Barto. The detailed behavior of the RL mechanism is determined by numerous parameters that can be controlled and configured via the `r1` command. Please refer to the documentation for that command in section [9.4.2](#) on page [236](#).

5.1 RL Rules

Soar’s RL mechanism learns Q-values for state-operator¹ pairs. Q-values are stored as numeric-indifferent preferences created by specially formulated productions called **RL rules**. RL rules are identified by syntax. A production is a RL rule if and only if its left hand side tests for a proposed operator, its right hand side creates a single numeric-indifferent preference, and it is not a template rule (see Section [5.4.2](#) for template rules). These constraints ease the technical requirements of identifying/updating RL rules and makes it easy for the agent programmer to add/maintain RL capabilities within an agent. We define an **RL operator** as an operator with numeric-indifferent preferences created by RL rules.

The following is an RL rule:

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

¹ In this context, the term “state” refers to the state of the task or environment, not a state identifier. For the rest of this chapter, bold capital letter names such as **S1** will refer to identifiers and italic lowercase names such as *s₁* will refer to task states.

```

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

```

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

The following are not RL rules:

```

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

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

sp {invalid*actions
  (state <s> ^operator <o> +)
-->
  (<s> ^operator <o> = 5)
  (write (crlf) |This is not an RL rule.|)
}

```

The first rule proposes multiple preferences for the proposed operator and thus does not comply with the rule format. The second rule does not comply because it does not provide a *constant* for the numeric-indifferent preference value. The third rule does not comply because it includes a *RHS function* action in addition to the numeric-indifferent preference action.

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

In the more general case, multiple RL rules can match a single state-operator pair, and a single RL rule can match multiple state-operator pairs. That is, in Soar, a state-operator pair corresponds to an operator in a specific working memory context, and multiple rules can

modify the preferences for a single operator, and a single rule can be instantiated multiple ways to modify preferences for multiple operators. For RL in Soar, all numeric-indifferent preferences for an operator are summed when calculating the operator's Q-value². In this context, RL rules can be interpreted more generally as binary features in a linear approximator of each state-operator pair's Q-value, and their numeric-indifferent preference values their weights. In other words,

$$Q(s, a) = w_1\phi_1(s, a) + w_2\phi_2(s, a) + \dots + w_n\phi_n(s, a)$$

where all RL rules in production memory are numbered $1 \dots n$, $Q(s, a)$ is the Q-value of the state-operator pair (s, a) , w_i is the numeric-indifferent preference value of RL rule i , $\phi_i(s, a) = 0$ if RL rule i does not match (s, a) , and $\phi_i(s, a) = 1$ if it does. This interpretation allows RL rules to simulate a number of popular function approximation schemes used in RL such as tile coding and sparse coding.

5.2 Reward Representation

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

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

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

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

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

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

There are two reasons for requiring the intermediate identifier. The first is so that multiple reward signals with the same value can exist simultaneously. Since working memory is a

² This is assuming the value of **numeric-indifferent-mode** is set to **sum**. In general, the RL mechanism only works correctly when this is the case, and we assume this case in the rest of the chapter. See page 195 for more information about this parameter.

set, multiple WMEs with identical values in all three positions (identifier, attribute, value) cannot exist simultaneously. Without an intermediate identifier, specifying two rewards with the same value would require a WME structure such as

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

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

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

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

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

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

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

5.3 Updating RL Rule Values

Soar's RL mechanism is integrated naturally with the decision cycle and performs online updates of RL rules. Whenever an RL operator is selected, the values of the corresponding RL rules will be updated. The update can be on-policy (Sarsa) or off-policy (Q-Learning),

as controlled by the **learning-policy** parameter of the **rl** command. (See page 236.) Let δ_t be the amount of change for the Q-value of an RL operator in a single update. For Sarsa, we have

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

where

- $Q(s_t, a_t)$ is the Q-value of the state and chosen operator in decision cycle t .
- $Q(s_{t+1}, a_{t+1})$ is the Q-value of the state and chosen RL operator in the next decision cycle.
- r_{t+1} is the total reward collected in the next decision cycle.
- α and γ are the settings of the **learning-rate** and **discount-rate** parameters of the **rl** command, respectively.

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

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

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

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

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

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

```
rl-1: (S1 ^operator 01 = 2.3)
rl-2: (S1 ^operator 01 = -1)
```

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

2. **O1** is selected and executed, so $Q(s_t, a_t) = Q(s_t, \mathbf{O1}) = 1.3$.
3. In decision cycle $t+1$, a total reward of 1.0 is collected on the **reward-link**, an operator **O2** is proposed, and another RL rule **rl-3** creates the following numeric-indifferent preference for it:

```
rl-3: (S1 ^operator 02 = 0.5)
```

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

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

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

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

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

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

5.3.1 Gaps in Rule Coverage

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

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

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

and the Q-Learning update is

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

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

Gap propagation can be disabled by setting the **temporal-extension** parameter of the **rl** command to **off**. When gap propagation is disabled, the RL rules preceding a gap are updated using $Q(s_{t+1}, a_{t+1}) = 0$. The **rl** setting of the **watch** command (see Section 9.6.1 on page 258) is useful in identifying gaps.

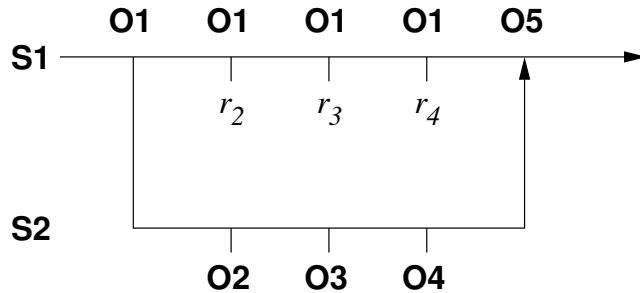


Figure 5.1: Example Soar substate operator trace.

5.3.2 RL and Substates

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

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

- At decision cycle 1, RL operator **O1** is selected in **S1** and causes an operator-no-change impasse for three decision cycles.
- In the substate **S2**, operators **O2**, **O3**, and **O4** are selected and applied sequentially.
- Meanwhile in **S1**, rewards r_2 , r_3 , and r_4 are put on the **reward-link** sequentially.
- Finally, the impasse is resolved by **O4**, the proposal for **O1** is retracted, and RL operator **O5** is selected in **S1**.

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

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

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

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

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

For impasses other than operator no-change, RL acts as if the impasse hadn't occurred. If **O1** is the last RL operator selected before the impasse, r_2 the reward received in the decision

cycle immediately following, and \mathbf{O}_n , the first operator selected after the impasse, then $\mathbf{O}1$ is updated with

$$\delta_1 = \alpha [r_2 + \gamma Q(s_n, \mathbf{O}_n) - Q(s_1, \mathbf{O}1)]$$

If an RL operator is selected in a substate immediately prior to the state's retraction, the RL rules will be updated based only on the reward signals present and not on the Q-values of future operators. This point is not covered in traditional RL theory. The retraction of a substate corresponds to a suspension of the RL task in that state rather than its termination, so the last update assumes the lack of information about future rewards rather than the discontinuation of future rewards. To handle this case, the numeric-indifferent preference value of each RL rule is stored as two separate values, the *expected current reward* (ECR) and *expected future reward* (EFR). The ECR is an estimate of the expected immediate reward signal for executing the corresponding RL operator. The EFR is an estimate of the time discounted Q-value of the next RL operator. Normal updates correspond to traditional RL theory (showing the Sarsa case for simplicity):

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

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

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

5.3.3 Eligibility Traces

The RL mechanism supports eligibility traces, which can improve the speed of learning by updating RL rules across multiple sequential steps.

The **eligibility-trace-decay-rate** and **eligibility-trace-tolerance** parameters control this mechanism. By setting **eligibility-trace-decay-rate** to 0 (default), eligibility traces are in effect disabled. When eligibility traces are enabled, the particular algorithm used is dependent upon the learning policy. For Sarsa, the eligibility trace implementation is *Sarsa*(λ). For Q-Learning, the eligibility trace implementation is *Watkin's* $Q(\lambda)$.

5.3.3.1 Exploration

The **decide indifferent-selection** command (page 195) determines how operators are selected based on their numeric-indifferent preferences. Although all the indifferent selection settings are valid regardless of how the numeric-indifferent preferences were arrived at, the

`epsilon-greedy` and `boltzmann` settings are specifically designed for use with RL and correspond to the two most common exploration strategies. In an effort to maintain backwards compatibility, the default exploration policy is `softmax`. As a result, one should change to `epsilon-greedy` or `boltzmann` when the reinforcement learning mechanism is enabled.

5.3.4 GQ(λ)

Sarsa(λ) and *Watkin's Q*(λ) help agents to solve the temporal credit assignment problem more quickly. However, if you wish to implement something akin to CMACs to generalize from experience, convergence is not guaranteed by these algorithms. *GQ*(λ) is a gradient descent algorithm designed to ensure convergence when learning off-policy. Soar's `learning-policy` can be set to `on-policy-gq-lambda` or `off-policy-gq-lambda` to increase the likelihood of convergence when learning under these conditions. If you should choose to use one of these algorithms, we recommend setting the `r1 step-size-parameter` to something small, such as 0.01 in order to ensure that the secondary set of weights used by *GQ*(λ) change slowly enough for efficient convergence.

5.4 Automatic Generation of RL Rules

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

5.4.1 The gp Command

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

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

For more information see the documentation for this command on page [202](#).

5.4.2 Rule Templates

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

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

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

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

```
(S1 ^value 3.2)
(S1 ^operator 01 +)
```

Then the following production will be added to production memory:

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

The variable **<v>** is replaced by 3.2 on both the LHS and the RHS, but **<s>** and **<o>** are not replaced because they matches against identifiers (**S1** and **01**). As with other RL rules, the value of 3.2 on the RHS of this rule may be updated later by reinforcement learning, whereas the value of 3.2 on the LHS will remain unchanged. If **<v>** had matched against a non-numeric constant, it will be replaced by that constant on the LHS, but the RHS

numeric-indifference preference value will be set to zero to make the new rule valid.

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

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

5.4.3 Chunking

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

Chapter 6

Semantic Memory

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

This chapter is organized as follows: semantic memory structures in working memory (6.1); representation of knowledge in semantic memory (6.2); storing semantic knowledge (6.3); retrieving semantic knowledge (6.4); and a discussion of performance (6.5). The detailed behavior of semantic memory is determined by numerous parameters that can be controlled and configured via the **smem** command. Please refer to the documentation for that command in Section 9.5.1 on page 241.

6.1 Working Memory Structure

Upon creation of a new state in working memory (see Section 2.7.1 on page 28; Section 3.4 on page 84), the architecture creates the following augmentations to facilitate agent interaction with semantic memory:

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

As rules augment the **command** structure in order to access/change semantic knowledge (6.3, 6.4), semantic memory augments the **result** structure in response. Production actions should not remove augmentations of the **result** structure directly, as semantic memory will maintain these WMEs.

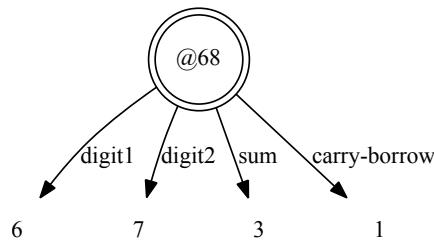


Figure 6.1: Example long-term identifier with four augmentations.

6.2 Knowledge Representation

The representation of knowledge in semantic memory is similar to that in working memory (see Section 2.2 on page 14) – both include graph structures that are composed of symbolic elements consisting of an identifier, an attribute, and a value. It is important to note, however, key differences:

- Currently semantic memory only supports attributes that are symbolic constants (string, integer, or decimal), but *not* attributes that are identifiers
- Whereas working memory is a single, connected, directed graph, semantic memory can be disconnected, consisting of multiple directed, connected sub-graphs

From Soar 9.6 onward, **Long-term identifiers** (LTIs) are defined as identifiers that exist in semantic memory *only*. Each LTI is permanently associated with a specific number that labels it (e.g. @5 or @7). Instances of an LTI can be loaded into working memory as regular *short-term* identifiers (STIs) linked with that specific LTI. For clarity, when printed, a short-term identifier associated with an LTI is followed with the label of that LTI. For example, if the working memory ID L7 is associated with the LTI named @29, printing that STI would appear as L7 (@29).

When presented in a figure, long-term identifiers will be indicated by a double-circle. For instance, Figure 6.1 depicts the long-term identifier @68, with four augmentations, representing the addition fact of $6 + 7 = 13$ (or, rather, 3, carry 1, in context of multi-column arithmetic).

6.2.1 Integrating Long-Term Identifiers with Soar

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

6.2.1.1 Working Memory

Long-term identifiers themselves never exist in working memory. Rather, instances of long term memories are loaded into working memory as STIs through queries or retrievals, and manipulated just like any other WMEs. Changes to any STI augmentations do not directly have any effect upon linked LTIs in semantic memory. Changes to LTIs themselves only occur though `store` commands on the command link or through command-line directives such as `smem --add` (see below).

Each time an agent loads an instance of a certain LTI from semantic memory into working memory using queries or retrievals, the instance created will always be a new unique STI. This means that if same long-term memory is retrieved multiple times in succession, each retrieval will result in a different STI instance, each linked to the same LTI. A benefit of this is that a retrieved long-term memory can be modified without compromising the ability to recall what the actual stored memory is.¹

6.2.1.2 Procedural Memory

Soar productions can use various conditions to test whether an STI is associated with an LTI or whether two STIs are linked to the same LTI (see Section 3.3.5.3 on page 53). LTI names (e.g. @6) may *not* appear in the action side of productions.

6.2.1.3 Episodic Memory

Episodic memory (see Section 7 on page 155) faithfully captures LTI-linked STIs, including the episode of transition. Retrieved episodes contain STIs as they existed during the episode, regardless of any changes to linked LTIs that transpired since the episode occurred.

6.3 Storing Semantic Knowledge

6.3.1 Store command

An agent stores a long-term identifier in semantic memory by creating a `^store` command: this is a WME whose identifier is the `command` link of a state's `smem` structure, the attribute is `store`, and the value is a short-term identifier.

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

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

¹ Before Soar 9.6, LTIs were themselves retrieved into working memory. This meant all augmentations to such IDs, whether from the original retrieval or added after retrieval, would always be merged under the same ID, unless `deep-copy` was used to make a duplicate short-term memory.

commands.

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

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

If the identifier used in the store command is not linked to any existing LTIs, a new LTI will be created in smem and the stored STI will be linked to it. If the identifier used in the store command is already linked to an LTI, the store will overwrite that long-term memory. For example, if an existing LTI @5 had augmentations **^A do ^B re ^C mi**, and a **store** command stored short-term identifier L35 which was linked to @5 but had only the augmentation **^D fa**, the LTI @5 would be changed to only have **^D fa**.

6.3.2 Store-new command

The **^store-new** command structure is just like the **^store** command, except that smem will always store the given memory as an entirely new structure, regardless of whether the given STI was linked to an existing LTI or not. Any STIs that don't already have links will get linked to the newly created LTIs. But if a stored STI was already linked to some LTI, Soar will not re-link it to the newly created LTI.

If this behavior is not desired, the agent can add a **^link-to-new-LTM yes** augmentation to override this behavior. One use for this setting is to allow chunking to backtrace through a stored memory in a manner that will be consistent with a later state of memory when the newly stored LTI is retrieved again.

6.3.3 User-Initiated Storage

Semantic memory provides agent designers the ability to store semantic knowledge via the **add** switch of the **smem** command (see Section 9.5.1 on page 241). The format of the command is nearly identical to the working memory manipulation components of the RHS of a production (i.e. no RHS-functions; see Section 3.3.6 on page 67). For instance:

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

Unlike agent storage, declarative storage is automatically recursive. Thus, this command instance will add a new long-term identifier (represented by the temporary 'arithmetic' vari-

able) with three augmentations. The value of each augmentation will each become an LTI with two constant attribute/value pairs. Manual storage can be arbitrarily complex and use standard dot-notation. The add command also supports hardcoded LTI ids such as @1 in place of variables.

6.3.4 Storage Location

Semantic memory uses SQLite to facilitate efficient and standardized storage and querying of knowledge. The semantic store can be maintained in memory or on disk (per the `database` and `path` parameters; see Section 9.5.1). If the store is located on disk, users can use any standard SQLite programs/components to access/query its contents. However, using a disk-based semantic store is very costly (performance is discussed in greater detail in Section 6.5 on page 153), and running in memory is recommended for most runs.

Note that changes to storage parameters, for example `database`, `path` and `append` will not have an effect until the database is used after an initialization. This happens either shortly after launch (on first use) or after a database initialization command is issued. To switch databases or database storage types while running, set your new parameters and then perform an `-init` command.

The `path` parameter specifies the file system path the database is stored in. When `path` is set to a valid file system path and `database` mode is set to *file*, then the SQLite database is written to that path.

The `append` parameter will determine whether all existing facts stored in a database on disk will be erased when semantic memory loads. Note that this affects `soar init` also. In other words, if the `append` setting is off, all semantic facts stored to disk will be lost when a `soar init` is performed. For semantic memory, `append` mode is `on` by default.

Note: As of version 9.3.3, Soar used a new schema for the semantic memory database. This means databases from 9.3.2 and below can no longer be loaded. A conversion utility is available in Soar 9.4 to convert from the old schema to the new one.

The `lazy-commit` parameter is a performance optimization. If set to `on` (default), disk databases will not reflect semantic memory changes until the Soar kernel shuts down. This improves performance by avoiding disk writes. The `optimization` parameter (see Section 6.5 on page 153) will have an affect on whether databases on disk can be opened while the Soar kernel is running.

6.4 Retrieving Semantic Knowledge

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

that state's `smem` structure.

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

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

Where the `<smem-c>` variable refers to the `command` structure of the state.

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

6.4.1 Non-Cue-Based Retrievals

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

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

If the value of the command is not an LTI-linked identifier, an error will result:

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

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

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

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

6.4.2 Cue-Based Retrievals

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

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

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

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

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

```

sp {smem*sample*query
  (state <s> ^smem.command <scmd>
    ^lti <lti>
    ^input-link.foo <bar>)
-->
  (<scmd> ^query <q>
  (<q> ^name <any-name>
    ^foo <bar>
    ^associate <lti>
    ^age 25)
}

```

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

- Attribute `name` with ANY value
- Attribute `foo` with value equal to that of variable `<bar>` at the time this rule fires
- Attribute `associate` with value that is the same long-term identifier as that linked to by the `<lti>` STI at the time this rule fires
- Attribute `age` with integer value 25

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

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

Otherwise, two WMEs are added:

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

The result `<retrieved-lti>` will be a new short-term identifier linked to the result LTI.

As with non-cue-based retrievals, all of the augmentations of the long-term identifier in semantic memory are added as new WMEs to working memory. If these augmentations include other LTIs in smem, they too are instantiated into new short-term identifiers in working memory.

It is possible that multiple long-term identifiers match the cue equally well. In this case, semantic memory will retrieve the long-term identifier that was most recently stored/retrieved. (More accurately, it will retrieve the LTI with the greatest *activation* value. See below.)

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

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

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

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

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

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

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

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

Conditional queries are of the format:

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

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

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

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

less A value less than the given argument

greater A value greater than the given argument

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

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

max The maximum value for the attribute

min The minimum value for the attribute

6.4.2.1 Activation

When an agent issues a cue-based retrieval and multiple LTIs match the cue, the LTI which semantic memory provides to working memory as the result is the LTI which not only matches the cue, but also has the highest **activation** value. Semantic memory has several activation methods available for this purpose.

The simplest activation methods are **recency** and **frequency** activation. Recency activation attaches a time-stamp to each LTI and records the time of last retrieval. Using recency activation, the LTI which matches the cue and was also most-recently retrieved is the one which is returned as the result for a query. Frequency activation attaches a counter to each LTI and records the number of retrievals for that LTI. Using frequency activation, the LTI which matches the cue and also was most frequently used is returned as the result of the query. By default, Soar uses recency activation.

Base-level activation can be thought of as a mixture of both recency and frequency. Soar makes use of the following equation (known as the Petrov approximation²) for calculating base-level activation:

$$BLA = \log \left[\sum_{i=1}^k t_i^{-d} + \frac{(n-k)(t_n^{1-d} - t_k^{1-d})}{(1-d)(t_n - t_k)} \right]$$

where n is the number of activation boosts, t_n is the time since the first boost, t_k is the time of the k th boost, d is the decay factor, and k is the number of recent activation boosts which are stored. (In Soar, k is hard-coded to 10.) To use base-level activation, use the following CLI command when sourcing an agent:

```
smem --set activation-mode base-level
```

Spreading activation is new to Soar 9.6.0 and provides a secondary type of activation beyond the previous methods. First, spreading activation requires that base-level activation is also being used. They are considered additive. This value does not represent recency or frequency of use, but rather context-relatedness. Spreading activation increases the activation of LTIs which are linked to by identifiers currently present in working memory.³ Such LTIs may be thought of as *spreading sources*.

Spreading activation values spread according to network structure. That is, spreading sources will add to the spreading activation values of any of their child LTIs, according to the directed graph structure within `smem` (not working memory). The amount of spread is controlled by the

spreading-continue-probability parameter. By default this value is set to 0.9. This would mean that 90% of an LTI's spreading activation value would be divided among its direct children (without subtracting from its own value). This value is multiplicative with depth. A “grandchild” LTI, connected at a distance of two from a source LTI, would receive spreading according to $0.9 \times 0.9 = 0.81$ of the source spreading activation value.

Spreading activation values are updated each decision cycle only as needed for specific `smem` retrievals. For efficiency, two limits exist for the amount of spread calculated. The **spreading-limit** parameter limits how many LTIs can receive spread from a given spreading source LTI. By default, this value is (300). Spread is distributed in a magnitude-first manner to all descendants of a source. (Without edge-weights, this simplifies to breadth-first.) Once the number of LTIs that have been given spread from a given source reaches the max value indicated by **spreading-limit**, no more is calculated for that source that update cycle, and the next spreading source's contributions are calculated. The maximum depth of descendants that can receive spread contributions from a source is similarly given by the **spreading-depth-limit** parameter. By default, this value is (10).

In order to use spreading activation, use the following command:

```
smem --set spreading on
```

²Petrov, Alexander A. “Computationally efficient approximation of the base-level learning equation in ACT-R.” *Proceedings of the seventh international conference on cognitive modeling*. 2006.

³ Specifically, linked to by STIs that have augmentations.

Also, spreading activation can make use of working memory activation for adjusting edge weights and for providing nonuniform initial magnitude of spreading for sources of spread. This functionality is optional. To enable the updating of edge-weights, use the command:

```
smem --set spreading-edge-updating on
```

and to enable working memory activation to modulate the magnitude of spread from sources, use the command:

```
smem --set spreading-wma-source on
```

For most use-cases, base-level activation is sufficient to provide an agent with relevant knowledge in response to a query. However, to provide an agent with more context-relevant results as opposed to results based only on historical usage, one must use spreading activation.

6.4.3 Retrieval with Depth

For either cue-based or non-cue-based retrieval, it is possible to retrieve a long-term identifier with additional depth. Using the **depth** parameter allows the agent to retrieve a greater amount of the memory structure than it would have by retrieving not only the long-term identifier's attributes and values, but also by recursively adding to working memory the attributes and values of that long-term identifier's children.

Depth is an additional command attribute, like query:

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

For instance, the following rule uses depth with a cue-based retrieval:

```
sp {smem*sample*query
  (state <s> ^smem.command <sc>
    ^input-link.foo <bar>)
-->
  (<sc> ^query <q>
    ^depth 2)
  (<q> ^name <any-name>
    ^foo <bar>
    ^associate <lti>
    ^age 25)
}
```

In the example above and without using depth, the long-term identifier referenced by

```
^associate <lti>
```

would not also have its attributes and values be retrieved. With a depth of 2 or more, that long-term identifier also has its attributes and values added to working memory.

Depth can incur a large cost depending on the specified depth and the structures stored in

semantic memory.

6.5 Performance

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

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

6.5.1 Math queries

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

6.5.2 Performance Tweaking

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

The next parameter is **thresh**. This has to do with the locality of storing/updating activation information with semantic augmentations. By default, all WME augmentations are incrementally sorted by activation, such that cue-based retrievals need not sort large number of candidate long-term identifiers on demand, and thus retrieval time is independent of cue selectivity. However, each activation update (such as after a retrieval) incurs an update cost linear in the number of augmentations. If the number of augmentations for a long-term identifier is large, this cost can dominate. Thus, the `thresh` parameter sets the upper bound of augmentations, after which activation is stored with the long-term identifier. This allows

the user to establish a balance between cost of updating augmentation activation and the number of long-term identifiers that must be pre-sorted during a cue-based retrieval. As long as the threshold is greater than the number of augmentations of most long-term identifiers, performance should be fine (as it will bound the effects of selectivity).

The next two parameters deal with the SQLite cache, which is a memory store used to speed operations like queries by keeping in memory structures like levels of index B+-trees. The first parameter, **page-size**, indicates the size, in bytes, of each cache page. The second parameter, **cache-size**, suggests to SQLite how many pages are available for the cache. Total cache size is the product of these two parameter settings. The cache memory is not pre-allocated, so short/small runs will not necessarily make use of this space. Generally speaking, a greater number of cache pages will benefit query time, as SQLite can keep necessary metadata in memory. However, some documented situations have shown improved performance from decreasing cache pages to increase memory locality. This is of greater concern when dealing with file-based databases, versus in-memory. The size of each page, however, may be important whether databases are disk- or memory-based. This setting can have far-reaching consequences, such as index B+-tree depth. While this setting can be dependent upon a particular situation, a good heuristic is that short, simple runs should use small values of the page size (1k, 2k, 4k), whereas longer, more complicated runs will benefit from larger values (8k, 16k, 32k, 64k). The episodic memory chapter (see Section 7.4 on page 161) has some further empirical evidence to assist in setting these parameters for very large stores.

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

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

Chapter 7

Episodic Memory

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

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

Please refer to the documentation for that command in Section 9.5.2 on page 251.

7.1 Working Memory Structure

Upon creation of a new state in working memory (see Section 2.7.1 on page 28; Section 3.4 on page 84), the architecture creates the following augmentations to facilitate agent interaction with episodic memory:

```
(<s> ^epmem <e>)
  (<e> ^command <e-c>)
  (<e> ^result <e-r>)
  (<e> ^present-id #)
```

As rules augment the **command** structure in order to retrieve episodes (7.3), episodic memory augments the **result** structure in response. Production actions should not remove augmentations of the **result** structure directly, as episodic memory will maintain these WMEs.

The value of the **present-id** augmentation is an integer and will update to expose to the agent the current episode number. This information is identical to what is available via the *time* statistic (see Section 9.5.2 on page 251) and the *present-id* retrieval meta-data (7.3.4).

7.2 Episodic Storage

Episodic memory records new episodes without deliberate action/consideration by the agent. The timing and frequency of recording new episodes is controlled by the `phase` and `trigger` parameters. The `phase` parameter sets the phase in the decision cycle (default: end of each decision cycle) during which episodic memory stores episodes and processes commands. The value of the `trigger` parameter indicates to the architecture the event that concludes an episode: adding a new augmentation to the output-link (default) or each decision cycle.

For debugging purposes, the `force` parameter allows the user to manually request that an episode be recorded (or not) during the current decision cycle. Behavior is as follows:

- The value of the `force` parameter is initialized to `off` every decision cycle.
- During the `phase` of episodic storage, episodic memory tests the value of the `force` parameter; if it has a value other than `off`, episodic memory follows the *forced* policy irrespective of the value of the `trigger` parameter.

7.2.1 Episode Contents

When episodic memory stores a new episode, it captures the entire top-state of working memory. There are currently two exceptions to this policy:

- Episodic memory only supports WMEs whose attribute is a constant. Behavior is currently undefined when attempting to store a WME that has an attribute that is an identifier.
- The `exclusions` parameter allows the user to specify a set of attributes for which Soar will not store WMEs. The storage process currently walks the top-state of working memory in a breadth-first manner, and any WME that is not reachable other than via an excluded WME will not be stored. By default, episodic memory excludes the `epmem` and `smem` structures, to prevent encoding of potentially large and/or frequently changing memory retrievals.

7.2.2 Storage Location

Episodic memory uses SQLite to facilitate efficient and standardized storage and querying of episodes. The episodic store can be maintained in memory or on disk (per the `database` and `path` parameters). If the store is located on disk, users can use any standard SQLite programs/components to access/query its contents. See the later discussion on performance (7.4) for additional parameters dealing with databases on disk.

Note that changes to storage parameters, for example `database`, `path` and `append` will not have an effect until the database is used after an initialization. This happens either shortly after launch (on first use) or after a database initialization command is issued. To

switch databases or database storage types while running, set your new parameters and then perform an `epmem --init` command.

The **path** parameter specifies the file system path the database is stored in. When **path** is set to a valid file system path and **database** mode is set to *file*, then the SQLite database is written to that path.

The **append** parameter will determine whether all existing facts stored in a database on disk will be erased when episodic memory loads. Note that this affects `init-soar` also. In other words, if the **append** setting is off, all episodes stored will be lost when an `init-soar` is performed. For episodic memory, **append** mode is **off** by default.

Note: As of version 9.3.3, Soar now uses a new schema for the episodic memory database. This means databases from 9.3.2 and below can no longer be loaded. A conversion utility will be available in Soar 9.4 to convert from the old schema to the new one.

7.3 Retrieving Episodes

An agent retrieves episodes by creating an appropriate command (we detail the types of commands below) on the `command` link of a state's `epmem` structure. At the end of the `phase` of each decision, after episodic storage, episodic memory processes each state's `epmem` command structure. Results, meta-data, and errors are placed on the `result` structure of that state's `epmem` structure.

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

```
<s> ^epmem.result.status bad-cmd
```

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

All retrieved episodes are recreated exactly as stored, except for any operators that have an acceptable preference, which are recreated with the attribute `operator*`. For example, if the original episode was:

```
(<s> ^operator <o1> +)
(<o1> ^name move)
```

A retrieval of the episode would become:

```
(<s> ^operator* <o1>)
(<o1> ^name move)
```

7.3.1 Cue-Based Retrievals

Cue-based retrieval commands are used to search for an episode in the store that best matches an agent-supplied cue, while adhering to optional modifiers. A cue is composed of WMEs that partially describe a top-state of working memory in the retrieved episode. All cue-based retrieval requests must contain a single **`^query`** cue and, optionally, a single **`^neg-query`** cue.

```
<s> ^epmem.command.query <required-cue>
<s> ^epmem.command.neg-query <optional-negative-cue>
```

A `^query` cue describes structures desired in the retrieved episode, whereas a `^neg-query` cue describes non-desired structures. For example, the following Soar production creates a `^query` cue consisting of a particular state name and a copy of a current value on the `input-link` structure:

```
sp {epmem*sample*query
    (state <s> ^epmem.command <ec>
        ^io.input-link.foo <bar>)
-->
    (<ec> ^query <q>)
    (<q> ^name my-state-name
        ^io.input-link.foo <bar>)
}
```

As detailed below, multiple prior episodes may equally match the structure and contents of an agent's cue. Nuxoll has produced initial evidence that in some tasks, retrieval quality improves when using *activation* of cue WMEs as a form of feature weighting. Thus, episodic memory supports integration with working memory activation (see Section 9.3.2.1 on page 220). For a theoretical discussion of the Soar implementation of working memory activation, consider reading *Comprehensive Working Memory Activation in Soar* (Nuxoll, A., Laird, J., James, M., ICCM 2004).

The cue-based retrieval process can be thought of conceptually as a nearest-neighbor search. First, all candidate episodes, defined as episodes containing at least one leaf WME (a cue WME with no sub-structure) in at least one cue, are identified. Two quantities are calculated for each candidate episode, with respect to the supplied cue(s): the cardinality of the match (defined as the number of matching leaf WMEs) and the activation of the match (defined as the sum of the activation values of each matching leaf WME). Note that each of these values is negated when applied to a negative query. To compute each candidate episode's match score, these quantities are combined with respect to the **`balance`** parameter as follows:

$$(balance) * (cardinality) + (1 - balance) * (activation)$$

Performing a graph match on each candidate episode, with respect to the structure of the cue, could be very computationally expensive, so episodic memory implements a two-stage matching process. An episode with perfect cardinality is considered a perfect *surface* match

and, per the **graph-match** parameter, is subjected to further *structural* matching. Whereas surface matching efficiently determines if all paths to leaf WMEs exist in a candidate episode, graph matching indicates whether or not the cue can be structurally unified with the candidate episode (paying special regard to the structural constraints imposed by shared identifiers). Cue-based matching will return the most recent structural match, or the most recent candidate episode with the greatest match score.

A special note should be made with respect to how short- vs. long-term identifiers (see Section 6.2 on page 144) are interpreted in a cue. Short-term identifiers are processed much as they are in working memory – transient structures. Cue matching will try to find any identifier in an episode (with respect to WME path from state) that can apply. Long-term identifiers, however, are treated as constants. Thus, when analyzing the cue, episodic memory will not consider long-term identifier augmentations, and will only match with the same long-term identifier (in the same context) in an episode.

The case-based retrieval process can be further controlled using optional modifiers:

- The **before** command requires that the retrieved episode come relatively before a supplied time:

```
<s> ^epmem.command.before time
```

- The **after** command requires that the retrieved episode come relatively after a supplied time:

```
<s> ^epmem.command.after time
```

- The **prohibit** command requires that the time of the retrieved episode is not equal to a supplied time:

```
<s> ^epmem.command.prohibit time
```

Multiple prohibit command WMEs may be issued as modifiers to a single CB retrieval.

If no episode satisfies the cue(s) and optional modifiers an error is returned:

```
<s> ^epmem.result.failure <query> <optional-neg-query>
```

If an episode is returned, there is additional meta-data supplied (7.3.4).

7.3.2 Absolute Non-Cue-Based Retrieval

At time of storage, each episode is attributed a unique *time*. This is the current value of **time** statistic and is provided as the *memory-id* meta-data item of retrieved episodes (7.3.4). An absolute non-cue-based retrieval is one that requests an episode by time. An agent issues an absolute non-cue-based retrieval by creating a WME on the **command** structure with attribute *retrieve* and value equal to the desired time:

```
<s> ^epmem.command.retrieve time
```

Supplying an invalid value for the `retrieve` command will result in an error.

The time of the first episode in an episodic store will have value 1 and each subsequent episode's time will increase by 1. Thus the desired time may be the mathematical result of operations performed on a known episode's time.

The current episodic memory implementation does not implement any episodic store dynamics, such as forgetting. Thus any integer time greater than 0 and less than the current value of the `time` statistic will be valid. However, if forgetting is implemented in future versions, no such guarantee will be made.

7.3.3 Relative Non-Cue-Based Retrieval

Episodic memory supports the ability for an agent to “play forward” episodes using relative non-cue-based retrievals.

Episodic memory stores the time of the last successful retrieval (non-cue-based or cue-based). Agents can indirectly make use of this information by issuing `next` or `previous` commands. Episodic memory executes these commands by attempting to retrieve the episode immediately proceeding/preceding the last successful retrieval (respectively). To issue one of these commands, the agent must create a new WME on the `command` link with the appropriate attribute (`next` or `previous`) and value of an arbitrary identifier:

```
<s> ^epmem.command.next <n>
<s> ^epmem.command.previous <p>
```

If no such episode exists then an error is returned.

Currently, if the time of the last successfully retrieved episode is known to the agent (as could be the case by accessing result meta-data), these commands are identical to performing an absolute non-cue-based retrieval after adding/subtracting 1 to the last time (respectively). However, if an episodic store dynamic like forgetting is implemented, these relative commands are guaranteed to return the next/previous valid episode (assuming one exists).

7.3.4 Retrieval Meta-Data

The following list details the WMEs that episodic memory creates in the `result` link of the `epmem` structure wherein a command was issued:

- **`retrieved <retrieval-root>`** If episodic memory retrieves an episode, that memory is placed here. This WME is an identifier that is treated as the root of the state that was used to create the episodic memory. If the `retrieve` command was issued with an invalid time, the value of this WME will be *no-memory*.
- **`success <query> <optional-neg-query>`** If the cue-based retrieval was successful, the WME will have the status as the attribute and the value of the identifier of the query (and neg-query, if applicable).

- **match-score** This WME is created whenever an episode is successfully retrieved from a cue-based retrieval command. The WME value is a decimal indicating the raw match score for that episode with respect to the cue(s).
- **cue-size** This WME is created whenever an episode is successfully retrieved from a cue-based retrieval command. The WME value is an integer indicating the number of leaf WMEs in the cue(s).
- **normalized-match-score** This WME is created whenever an episode is successfully retrieved from a cue-based retrieval command. The WME value is the decimal result of dividing the raw match score by the cue size. It can hypothetically be used as a measure of episodic memory’s relative confidence in the retrieval.
- **match-cardinality** This WME is created whenever an episode is successfully retrieved from a cue-based retrieval command. The WME value is an integer indicating the number of leaf WMEs matched in the `^query` cue minus those matched in the `^neg-query` cue.
- **memory-id** This WME is created whenever an episode is successfully retrieved from a cue-based retrieval command. The WME value is an integer indicating the time of the retrieved episode.
- **present-id** This WME is created whenever an episode is successfully retrieved from a cue-based retrieval command. The WME value is an integer indicating the current time, such as to provide a sense of “now” in episodic memory terms. By comparing this value to the `memory-id` value, the agent can gain a sense of the relative time that has passed since the retrieved episode was recorded.
- **graph-match** This WME is created whenever an episode is successfully retrieved from a cue-based retrieval command and the `graph-match` parameter was `on`. The value is an integer with value 1 if graph matching was executed successfully and 0 otherwise.
- **mapping <mapping-root>** This WME is created whenever an episode is successfully retrieved from a cue-based retrieval command, the `graph-match` parameter was `on`, and structural match was successful on the retrieved episode. This WME provides a mapping between identifiers in the cue and in the retrieved episode. For each identifier in the cue, there is a `node` WME as an augmentation to the `mapping` identifier. The `node` has a `cue` augmentation, whose value is an identifier in the cue, and a `retrieved` augmentation, whose value is an identifier in the retrieved episode. In a graph match it is possible to have multiple identifier mappings – this map represents the “first” unified mapping (with respect to episodic memory algorithms).

7.4 Performance

There are currently two sources of “unbounded” computation: graph matching and cue-based queries. Graph matching is combinatorial in the worst case. Thus, if an episode

presents a perfect surface match, but imperfect structural match (i.e. there is no way to unify the cue with the candidate episode), there is the potential for exhaustive search. Each identifier in the cue can be assigned one of any historically consistent identifiers (with respect to the sequence of attributes that leads to the identifier from the root), termed a literal. If the identifier is a multi-valued attribute, there will be more than one candidate literals and this situation can lead to a very expensive search process. Currently there are no heuristics in place to attempt to combat the expensive backtracking. Worst-case performance will be combinatorial in the total number of literals for each cue identifier (with respect to cue structure).

The cue-based query algorithm begins with the most recent candidate episode and will stop search as soon as a match is found (since this episode must be the most recent). Given this procedure, it is trivial to create a two-WME cue that forces a linear search of the episodic store. Episodic memory combats linear scan by only searching candidate episodes, i.e. only those that contain a change in at least one of the cue WMEs. However, a cue that has no match and contains WMEs relevant to all episodes will force inspection of all episodes. Thus, worst-case performance will be linear in the number of episodes.

7.4.1 Performance Tweaking

When using a database stored to disk, several parameters become crucial to performance. The first is **commit**, which controls the number of episodes that occur between writes to disk. If the total number of episodes (or a range) is known ahead of time, setting this value to a greater number will result in greatest performance (due to decreased I/O).

The next two parameters deal with the SQLite cache, which is a memory store used to speed operations like queries by keeping in memory structures like levels of index B+-trees. The first parameter, **page-size**, indicates the size, in bytes, of each cache page. The second parameter, **cache-size**, suggests to SQLite how many pages are available for the cache. Total cache size is the product of these two parameter settings. The cache memory is not pre-allocated, so short/small runs will not necessarily make use of this space. Generally speaking, a greater number of cache pages will benefit query time, as SQLite can keep necessary metadata in memory. However, some documented situations have shown improved performance from decreasing cache pages to increase memory locality. This is of greater concern when dealing with file-based databases, versus in-memory. The size of each page, however, may be important whether databases are disk- or memory-based. This setting can have far-reaching consequences, such as index B+-tree depth. While this setting can be dependent upon a particular situation, a good heuristic is that short, simple runs should use small values of the page size (1k, 2k, 4k), whereas longer, more complicated runs will benefit from larger values (8k, 16k, 32k, 64k). One known situation of concern is that as indexed tables accumulate many rows (~millions), insertion time of new rows can suffer an infrequent, but linearly increasing burst of computation. In episodic memory, this situation will typically arise with many episodes and/or many working memory changes. Increasing the page size will reduce the intensity of the spikes at the cost of increasing disk I/O and average/total time for episode storage. Thus, the settings of page size for long, complicated runs establishes the

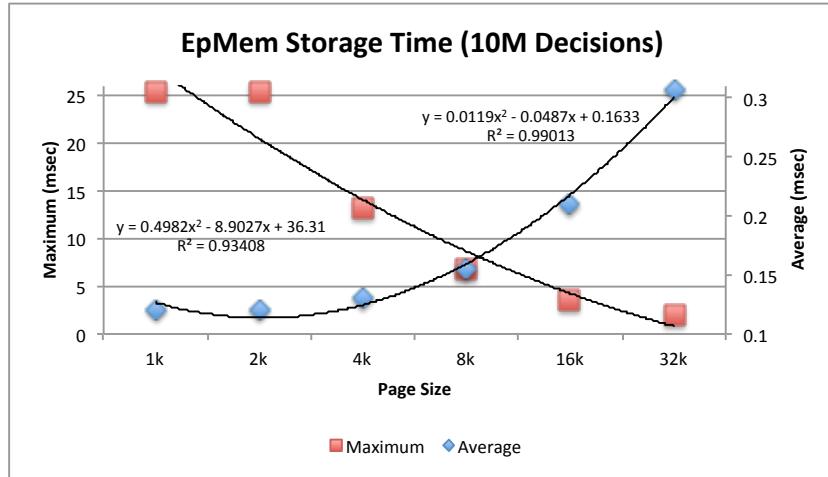


Figure 7.1: Example episodic memory cache setting data.

desired balance of reactivity (i.e. max computation) and average speed. To ground this discussion, the Figure 7.1 depicts maximum and average episodic storage time (the value of the epmem_storage timer, converted to milliseconds) with different page sizes after 10 million decisions (1 episode/decision) of a very basic agent (i.e. very few working memory changes per episode) running on a 2.8GHz Core i7 with Mac OS X 10.6.5. While only a single use case, the cross-point of these data forms the basis for the decision to default the parameter at 8192 bytes.

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

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

Chapter 8

Spatial Visual System

The Spatial Visual System (SVS) allows Soar to effectively represent and reason about continuous, three dimensional environments. SVS maintains an internal representation of the environment as a collection of discrete objects with simple geometric shapes, called the **scene graph**. The Soar agent can query for spatial relationships between the objects in the scene graph through a working memory interface similar to that of episodic and semantic memory. Figure 8.1 illustrates the typical use case for SVS by contrasting it with an agent that does not utilize it. The agent that does not use SVS (a. in the figure) relies on the environment to provide a symbolic representation of the continuous state. On the other hand, the agent that

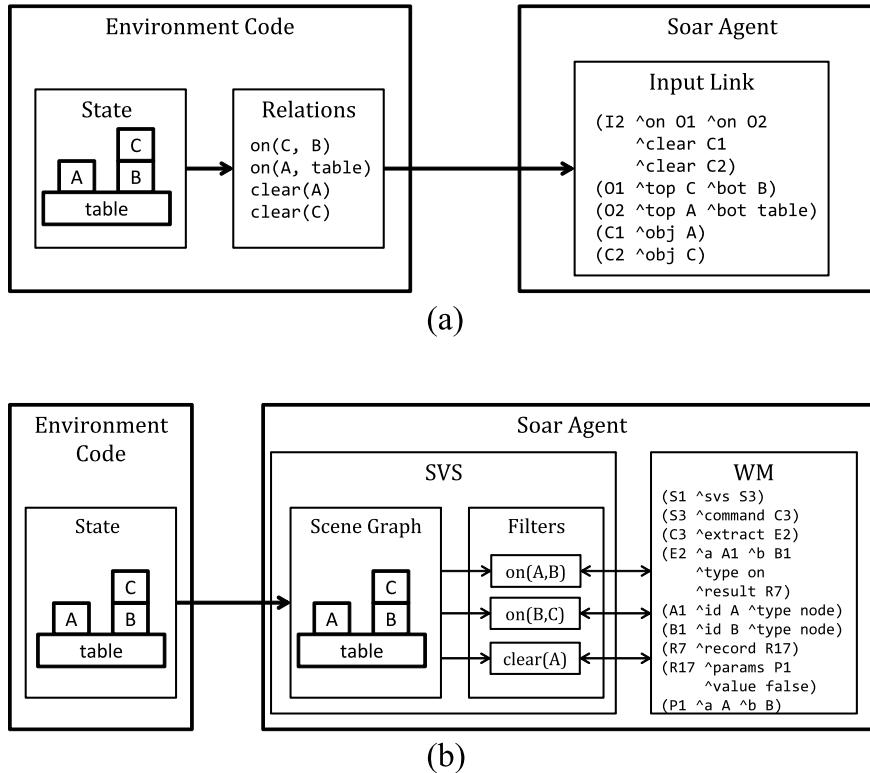


Figure 8.1: (a) Typical environment setup without using SVS. (b) Same environment using SVS.

uses SVS (b) accepts a continuous representation of the environment state directly, and then performs queries on the scene graph to extract a symbolic representation internally. This allows the agent to build more flexible symbolic representations without requiring modifications to the environment code. Furthermore, it allows the agent to manipulate internal copies of the scene graph and then extract spatial relationships from the modified states, which is useful for look-ahead search and action modeling. This type of imagery operation naturally captures and propagates the relationships implicit in spatial environments, and doesn't suffer from the frame problem that relational representations have.

8.1 The scene graph

The primary data structure of SVS is the scene graph. The scene graph is a tree in which the nodes represent objects in the scene and the edges represent “part-of” relationships between objects. An example scene graph consisting of a car and a pole is shown in Figure 8.2. The scene graph’s leaves are *geometry nodes* and its interior nodes are *group nodes*. Geometry nodes represent atomic objects that have intrinsic shape, such as the wheels and chassis in the example. Currently, the shapes supported by SVS are points, lines, convex polyhedrons, and spheres. Group nodes represent objects that are the aggregates of their child nodes, such as the car object in the example. The shape of a group node is the union of the shapes of its children. Structuring complex objects in this way allows Soar to reason about them naturally at different levels of abstraction. The agent can query SVS for relationships between the car as a whole with other objects (e.g. does it intersect the pole?), or the relationships between its parts (e.g. are the wheels pointing left or right with respect to the chassis?). The scene graph always contains at least a root node: the *world node*.

Each node other than the world node has a transform with respect to its parent. A transform consists of three components:

position (x, y, z) Specifies the x , y , and z offsets of the node’s origin with respect to its parent’s origin.

rotation (x, y, z) Specifies the rotation of the node relative to its origin in Euler angles.

This means that the node is rotated the specified number of radians along each axis in the order $x - y - z$. For more information, see http://en.wikipedia.org/wiki/Euler_angles.

scaling (x, y, z) Specifies the factors by which the node is scaled along each axis.

The component transforms are applied in the order scaling, then rotation, then position. Each node’s transform is applied with respect to its parent’s coordinate system, so the transforms accumulate down the tree. A node’s transform with respect to the world node, or its world transform, is the aggregate of all its ancestor transforms. For example, if the car has a position transform of $(1, 0, 0)$ and a wheel on the car has a position transform of $(0, 1, 0)$, then the world position transform of the wheel is $(1, 1, 0)$.

SVS represents the scene graph structure in working memory under the `^spatial-scene` link. The working memory representation of the car and pole scene graph is

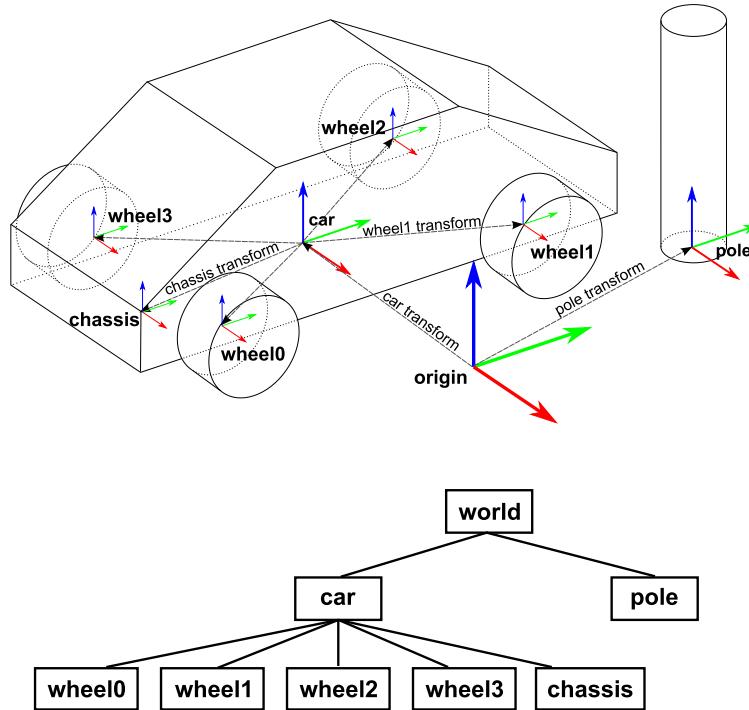


Figure 8.2: (a) A 3D scene. (b) The scene graph representation.

```

(S1 ^svs S3)
  (S3 ^command C3 ^spatial-scene S4)
    (S4 ^child C10 ^child C4 ^id world)
      (C10 ^id pole)
    (C4 ^child C9 ^child C8 ^child C7 ^child C6 ^child C5 ^id car)
      (C9 ^id chassis)
      (C8 ^id wheel3)
      (C7 ^id wheel2)
      (C6 ^id wheel1)
      (C5 ^id wheel0)
  
```

Each state in working memory has its own scene graph. When a new state is created, it will receive an independent copy of its parent's scene graph. This is useful for performing look-ahead search, as it allows the agent to destructively modify the scene graph in a search state using mental imagery operations.

8.1.1 svs_viewer

A viewer has been provided to show the scene graph visually. Run the program `svs_viewer -s PORT` from the soar/out folder to launch the viewer listening on the given port. Once the viewer is running, from within soar use the command `svs connect_viewer PORT` to connect to the viewer and begin drawing the scene graph. Any changes to the scene graph will be reflected in the viewer. The viewer by default draws the topstate scene graph, to draw that on a substate first stop drawing the topstate with `svs S1.scene.draw off` and then `svs S7.scene.draw on`.

8.2 Scene Graph Edit Language

The **Scene Graph Edit Language** (SGEL) is a simple, plain text, line oriented language that is used by SVS to modify the contents of the scene graph. Typically, the scene graph is used to represent the state of the external environment, and the programmer sends SGEL commands reflecting changes in the environment to SVS via the `Agent::SendSVSInput` function in the SML API. These commands are buffered by the agent and processed at the beginning of each input phase. Therefore, it is common to send scene changes through `SendSVSInput` before the input phase. If you send SGEL commands at the end of the input phase, the results won't be processed until the following decision cycle.

Each SGEL command begins with a single word command type and ends with a newline. The four command types are

`add ID PARENT_ID [GEOMETRY] [TRANSFORM]`

Add a node to the scene graph with the given ID, as a child of `PARENT_ID`, and with type `TYPE` (usually object). The `GEOMETRY` and `TRANSFORM` arguments are optional and described below.

`change ID [GEOMETRY] [TRANSFORM]`

Change the transform and/or geometry of the node with the given ID.

`delete ID`

Delete the node with the given ID.

`tag [add|change|delete] ID TAG_NAME TAG_VALUE`

Adds, changes, or deletes a tag from an object. A tag consists of a `TAG_NAME` and `TAG_VALUE` pair and is added to the node with the given ID. Both `TAG_NAME` and `TAG_VALUE` must be strings. Tags can differentiate nodes (e.g. as a type field) and can be used in conjunction with the `tag_select` filter to choose a subset of the nodes.

The `TRANSFORM` argument has the form `[p X Y Z] [r X Y Z] [s X Y Z]`, corresponding to the position, rotation, and scaling components of the transform, respectively. All the components are optional; any combination of them can be excluded, and the included components can appear in any order.

The `GEOMETRY` argument has two forms:

b RADIUS

Make the node a geometry node with sphere shape with radius **RADIUS**.

v X1 Y1 Z1 X2 Y2 Z2 ...

Make the node a geometry node with a convex polyhedron shape with the specified vertices. Any number of vertices can be listed.

8.2.1 Examples

Creating a sphere called ball4 with radius 5 at location (4, 4, 0).

```
add ball4 world b 5 p 4 4 0
```

Creating a triangle in the xy plane, then rotate it vertically, double its size, and move it to (1, 1, 1).

```
add tri9 world v -1 -1 0 1 -1 0 0 0.5 0 p 1 1 1 r 1.507 0 0 s 2 2 2
```

Creating a snowman shape of 3 spheres stacked on each other and located at (2, 2, 0).

```
add snowman world p 2 2 0
```

```
add bottomball snowman b 3 p 0 0 3
```

```
add middleball snowman b 2 p 0 0 8
```

```
add topball snowman b 1 p 0 0 11
```

Set the rotation transform on box11 to 180 degrees around the z axis.

```
change box11 r 0 0 3.14159
```

Changing the color tag on box7 to green.

```
tag change box7 color green
```

8.3 Commands

The Soar agent initiates commands in SVS via the **^command** link, similar to semantic and episodic memory. These commands allow the agent to modify the scene graph and extract filters. Commands are processed during the output phase and the results are added to working memory during the input phase. SVS supports the following commands:

add_node Creates a new node and adds it to the scene graph

copy_node Creates a copy of an existing node

delete_node Removes a node from the scene graph and deletes it

set_transform Changes the position, rotation, and/or scale of a node

set_tag Adds or changes a tag on a node

delete_tag Deletes a tag from a node

extract Compute the truth value of spatial relationships in the current scene graph.

extract_once Same as extract, except it is only computed once and doesn't update when the scene changes.

8.3.1 add_node

This command adds a new node to the scene graph.

^id [string] The id of the node to create

^parent [string] The id of the node to attach the new node to (default is world)

^geometry <> group point ball box >> The geometry the node should have

^position.{^x ^y ^z} Position of the node (optional)

^rotation.{^x ^y ^z} Rotation of the node (optional)

^scale.{^x ^y ^z} Scale of the node (optional)

The following example creates a node called `box5` and adds it to the world. The node has a box shape of side length 0.1 and is placed at position (1, 1, 0).

```
(S1 ^svs S3)
(S3 ^command C3 ^spatial-scene S4)
(C3 ^add_node A1)
  (A1 ^id box5 ^parent world ^geometry box ^position P1 ^scale S6)
    (P1 ^x 1.0 ^y 1.0 ^z 0.0)
    (S6 ^x 0.1 ^y 0.1 ^z 0.1)
```

8.3.2 copy_node

This command creates a copy of an existing node and adds it to the scene graph. This copy is not recursive, it only copies the node itself, not its children. The position, rotation, and scale transforms are also copied from the source node but they can be changed if desired.

^id [string] The id of the node to create

^source [string] The id of the node to copy

^parent [string] The id of the node to attach the new node to (default is world)

^position.{^x ^y ^z} Position of the node (optional)

^rotation.{^x ^y ^z} Rotation of the node (optional)

^scale.{^x ^y ^z} Scale of the node (optional)

The following example copies a node called `box5` as new node `box6` and moves it to position $(2, 0, 2)$.

```
(S1 ^svs S3)
  (S3 ^command C3 ^spatial-scene S4)
    (C3 ^copy_node A1)
      (A1 ^id box6 ^source box5 ^position P1)
        (P1 ^x 2.0 ^y 0.0 ^z 2.0)
```

8.3.3 delete_node

This command deletes a node from the scene graph. Any children will also be deleted.

^id [string] The id of the node to delete

The following example deletes a node called `box5`

```
(S1 ^svs S3)
  (S3 ^command C3 ^spatial-scene S4)
    (C3 ^delete_node D1)
      (D1 ^id box5)
```

8.3.4 set_transform

This command allows you to change the position, rotation, and/or scale of an existing node. You can specify any combination of the three transforms.

^id [string] The id of the node to change

^position.{^x ^y ^z} Position of the node (optional)

^rotation.{^x ^y ^z} Rotation of the node (optional)

^scale.{^x ^y ^z} Scale of the node (optional)

The following example moves and rotates a node called `box5`.

```
(S1 ^svs S3)
  (S3 ^command C3 ^spatial-scene S4)
    (C3 ^set_transform S6)
      (S6 ^id box5 ^position P1 ^rotation R1)
        (P1 ^x 2.0 ^y 2.0 ^z 0.0)
        (R1 ^x 0.0 ^y 0.0 ^z 1.57)
```

8.3.5 set_tag

This command allows you to add or change a tag on a node. If a tag with the same id already exists, the existing value will be replaced with the new value.

^id [string] The id of the node to set the tag on

^tag_name [string] The name of the tag to add

^tag_value [string] The value of the tag to add

The following example adds a shape tag to the node `box5`.

```
(S1 ^svs S3)
  (S3 ^command C3 ^spatial-scene S4)
    (C3 ^set_tag S6)
      (S6 ^id box5 ^tag_name shape ^tag_value cube)
```

8.3.6 delete_tag

This command allows you to delete a tag from a node.

^id [string] The id of the node to delete the tag from

^tag_name [string] The name of the tag to delete

The following example deletes the shape tag from the node `box5`.

```
(S1 ^svs S3)
  (S3 ^command C3 ^spatial-scene S4)
    (C3 ^delete_tag D1)
      (D1 ^name box5 ^tag_name shape)
```

8.3.7 extract and extract_once

This command is commonly used to compute spatial relationships in the scene graph. More generally, it puts the **result** of a filter pipeline (described in section 8.4) in working memory. Its syntax is the same as filter pipeline syntax. During the input phase, SVS will evaluate the filter and put a **^result** attribute on the command's identifier. Under the **^result** attribute is a multi-valued **^record** attribute. Each record corresponds to an output value from the head of the filter pipeline, along with the parameters that produced the value. With the regular **extract** command, these records will be updated as the scene graph changes. With the **extract_once** command, the records will be created once and will not change. Note that you should not change the structure of a filter once it is created (SVS only processes a command once). Instead to extract something different you must create a new command.

The following is an example of an extract command which tests whether the car and pole objects are intersecting. The `^status` and `^result` WMEs are added by SVS when the command is finished.

```
(S1 ^svs S3)
  (S3 ^command C3 ^spatial-scene S4)
    (C3 ^extract E2)
      (E2 ^a A1 ^b B1 ^result R7 ^status success ^type intersect)
        (A1 ^id car ^status success ^type node)
        (B1 ^id pole ^status success ^type node)
        (R7 ^record R17)
          (R17 ^params P1 ^value false)
            (P1 ^a car ^b pole)
```

8.4 Filters

Filters are the basic unit of computation in SVS. They transform the continuous information in the scene graph into symbolic information that can be used by the rest of Soar. Each filter accepts a number of labeled parameters as input, and produces a single output. Filters can be arranged into pipelines in which the outputs of some filters are fed into the inputs of other filters. The Soar agent creates filter pipelines by building an analogous structure in working memory as an argument to an "extract" command. For example, the following structure defines a set of filters that reports whether the car intersects the pole:

```
(S1 ^svs S3)
  (S3 ^command C3 ^spatial-scene S4)
    (C3 ^extract E2)
      (E2 ^a A1 ^b B1 ^type intersect)
        (A1 ^id car ^type node)
        (B1 ^id pole ^type node)
```

The `^type` attribute specifies the type of filter to instantiate, and the other attributes specify parameters. This command will create three filters: an `intersect` filter and two `node` filters. A `node` filter takes an `id` parameter and returns the scene graph node with that ID as its result. Here, the outputs of the `car` and `pole` node filters are fed into the `^a` and `^b` parameters of the `intersect` filter. SVS will update each filter's output once every decision cycle, at the end of the input phase. The output of the `intersect` filter is a boolean value indicating whether the two objects are intersecting. This is placed into working memory as the result of the extract command:

```
(S1 ^svs S3)
  (S3 ^command C3 ^spatial-scene S4)
    (C3 ^extract E2)
      (E2 ^a A1 ^b B1 ^result R7 ^status success ^type intersect)
        (A1 ^id car ^status success ^type node)
        (B1 ^id pole ^status success ^type node)
```

```
(R7 ^record R17)
  (R17 ^params P1 ^value false)
    (P1 ^a car ^b pole)
```

Notice that a `^status success` is placed on each identifier corresponding to a filter. A `^result` WME is placed on the extract command with a single record with value `false`.

8.4.1 Result lists

Spatial queries often involve a large number of objects. For example, the agent may want to compute whether an object intersects any others in the scene graph. It would be inconvenient to build the extract command to process this query if the agent had to specify each object involved explicitly. Too many WMEs would be required, which would slow down the production matcher as well as SVS because it must spend more time interpreting the command structure. To handle these cases, all filter parameters and results can be lists of values. For example, the query for whether one object intersects all others can be expressed as

```
(S1 ^svs S3)
  (S3 ^command C3)
    (C3 ^extract E2)
      (E2 ^a A1 ^b B1 ^result R7 ^status success ^type intersect)
        (A1 ^id car ^status success ^type node)
        (B1 ^status success ^type all_nodes)
        (R7 ^record R9 ^record R8)
          (R9 ^params P2 ^value false)
            (P2 ^a car ^b pole)
            (R8 ^params P1 ^value true)
              (P1 ^a car ^b car)
```

The `all_nodes` filter outputs a list of all nodes in the scene graph, and the `intersect` filter outputs a list of boolean values indicating whether the car intersects each node, represented by the multi-valued attribute `record`. Notice that each `record` contains both the result of the query as well as the parameters that produced that result. Not only is this approach more convenient than creating a separate command for each pair of nodes, but it also allows the `intersect` filter to answer the query more efficiently using special algorithms that can quickly rule out non-intersecting objects.

8.4.2 Filter List

The following is a list of all filters that are included in SVS. You can also get this list by using the cli command `svs filters` and get information about a specific filter using the command `svs filters.FILTER_NAME`. Many filters have a `_select` version. The select version returns a subset of the input nodes which pass a test. For example, the `intersect` filter returns boolean values for each input (a, b) pair, while the `intersect_select` filter returns the

nodes in set b which intersect the input node a. This is useful for passing the results of one filter into another (e.g. take the nodes that intersect node a and find the largest of them).

node

Given an `^id`, outputs the node with that id.

all_nodes

Outputs all the nodes in the scene

combine_nodes

Given multiple node inputs as `^a`, concates them into a single output set.

remove_node

Removes node `^id` from the input set `^a` and outputs the rest.

node_position

Outputs the position of each node in input `^a`.

node_rotation

Outputs the rotation of each node in input `^a`.

node_scale

Outputs the scale of each node in input `^a`.

node_bbox

Outputs the bounding box of each node in input `^a`.

distance and **distance_select**

Outputs the distance between input nodes `^a` and `^b`. Distance can be specified by `^distance_type <> centroid hull >>`, where `centroid` is the euclidean distance between the centers, and the `hull` is the minimum distance between the node surfaces. `distance_select` chooses nodes in set b in which the distance to node a falls within the range `^min` and `^max`.

closest and **farthest**

Outputs the node in set `^b` closest to or farthest from `^a` (also uses `distance_type`).

axis_distance and **axis_distance_select**

Outputs the distance from input node `^a` to `^b` along a particular axis (`^axis <> x y z >>`). This distance is based on bounding boxes. A value of 0 indicates the nodes overlap on the given axis, otherwise the result is a signed value indicating whether node b is greater or less than node a on the given axis. The `axis_distance_select` filter also uses `^min` and `^max` to select nodes in set b.

volume and **volume_select**

Outputs the bounding box volume of each node in set `^a`. For `volume_select`, it outputs a subset of the nodes whose volumes fall within the range `^min` and `^max`.

largest and **smallest**

Outputs the node in set `^a` with the largest or smallest volume.

larger and **larger_select**

Outputs whether input node ^a is larger than each input node ^b , or selects all nodes in b for which a is larger.

smaller and **smaller_select**

Outputs whether input node ^a is smaller than each input node ^b , or selects all nodes in b for which a is smaller.

contain and **contain_select**

Outputs whether the bounding box of each input node ^a contains the bounding box of each input node ^b , or selects those nodes in b which are contained by node a.

intersect and **intersect_select**

Outputs whether each input node ^a intersects each input node ^b , or selects those nodes in b which intersect node a. Intersection is specified by $\text{^intersect_type} \ll \text{hull box} \gg$; either the convex hull of the node or the axis-aligned bounding box.

tag_select

Outputs all the nodes in input set ^a which have the tag specified by ^tag_name and ^tag_value .

8.4.3 Examples

Select all the objects with a volume between 1 and 2.

```
(S1 ^svs S3)
(S3 ^command C3)
(C3 ^extract E1)
(E1 ^type volume_select ^a A1 ^min 1 ^max 2)
(A1 ^type all_nodes)
```

Find the distance between the centroid of ball3 and all other objects.

```
(S1 ^svs S3)
(S3 ^command C3)
(C3 ^extract E1)
(E1 ^type distance ^a A1 ^b B1 ^distance_type centroid)
(A1 ^type node ^id ball3)
(B1 ^type all_nodes)
```

Test where ball2 intersects any red objects.

```
(S1 ^svs S3)
(S3 ^command C3)
(C3 ^extract E1)
(E1 ^type intersect ^a A1 ^b B1 ^intersect_type hull)
(A1 ^type node ^id ball2)
(B1 ^type tag_select ^a A2 ^tag_name color ^tag_value red)
```

```
(A2 ^type all_nodes)
```

Find all the objects on the table. This is done by selecting nodes where the distance between them and the table along the z axis is a small positive number.

```
(S1 ^svs S3)
  (S3 ^command C3)
    (C3 ^extract E1)
      (E1 ^type axis_distance_select ^a A1 ^b B1 ^axis z ^min .0001 ^max .1)
        (A1 ^type node ^id table)
        (B1 ^type all_nodes)
```

Find the smallest object that intersects the table (excluding itself).

```
(S1 ^svs S3)
  (S3 ^command C3)
    (C3 ^extract E1)
      (E1 ^type smallest ^a A1)
        (A1 ^type intersect_select ^a A2 ^b B2 ^intersect_type hull)
          (A2 ^type node ^id table)
          (B1 ^type remove_node ^id table ^a A3)
            (A3 ^type all_nodes)
```

8.5 Writing new filters

SVS contains a small set of generally useful filters, but many users will need additional specialized filters for their application. Writing new filters for SVS is conceptually simple.

1. Write a C++ class that inherits from the appropriate filter subclass.
2. Register the new class in a global table of all filters (`filter_table.cpp`).
3. Recompile the kernel.

8.5.1 Filter subclasses

The fact that filter inputs and outputs are lists rather than single values introduces some complexity to how filters are implemented. Depending on the functionality of the filter, the multiple inputs into multiple parameters must be combined in different ways, and sets of inputs will map in different ways onto the output values. Furthermore, the outputs of filters are cached so that the filter does not repeat computations on sets of inputs that do not change. To shield the user from this complexity, a set of generally useful filter paradigms were implemented as subclasses of the basic `filter` class. When writing custom filters, try to inherit from one of these classes instead of from `filter` directly.

8.5.1.1 Map filter

This is the most straightforward and useful class of filters. A filter of this class takes the Cartesian product of all input values in all parameters, and performs the same computation on each combination, generating one output. In other words, this class implements a one-to-one mapping from input combinations to output values.

To write a new filter of this class, inherit from the `map_filter` class, and define the `compute` function. Below is an example template:

```
class new_map_filter : public map_filter<double> // templated with output type
{
public:
    new_map_filter(Symbol *root, soar_interface *si, filter_input *input, scene *scn)
        : map_filter<double>(root, si, input)    // call superclass constructor
    {}

    /* Compute
       Do the proper computation based on the input filter_params
       and set the out parameter to the result
       Return true if successful, false if an error occurred */
    bool compute(const filter_params* p, double& out){
        sgnode* a;
        if(!get_filter_param(this, p, "a", a)){
            set_status("Need input node a");
            return false;
        }
        out = // Your computation here
    }
};
```

8.5.1.2 Select filter

This is very similar to a map filter, except for each input combination from the Cartesian product the output is optional. This is useful for selecting and returning a subset of the outputs.

To write a new filter of this class, inherit from the `select_filter` class, and define the `compute` function. Below is an example template:

```
class new_select_filter : public select_filter<double> // templated with output type
{
public:
    new_select_filter(Symbol *root, soar_interface *si, filter_input *input, scene *scn)
        : select_filter<double>(root, si, input)    // call superclass constructor
    {}

    /* Compute
       Do the proper computation based on the input filter_params
       and set the out parameter to the result (if desired)
       Also set the select bit to true if you want to the result to be output.
       Return true if successful, false if an error occurred */
```

```

    bool compute(const filter_params* p, double& out, bool& select){
        sgnode* a;
        if(!get_filter_param(this, p, "a", a)){
            set_status("Need input node a");
            return false;
        }
        out = // Your computation here
        select = // test for when to output the result of the computation
    }
};

```

8.5.1.3 Rank filter

A filter where a ranking is computed for each combination from the Cartesian product of the input and only the combination which results in the highest (or lowest) value is output. The default behavior is to select the highest, to select the lowest you can call `set_select_highest(false)` on the filter.

To write a new filter of this class, inherit from the `rank_filter` class, and define the `rank` function. Below is an example template:

```

class new_rank_filter : public rank_filter
{
public:
    new_rank_filter(Symbol *root, soar_interface *si, filter_input *input, scene *scn)
        : rank_filter(root, si, input) // call superclass constructor
    {}

    /* Compute
       Do the proper computation based on the input filter_params
       And set r to the ranking result.
       Return true if successful, false if an error occurred */
    bool compute(const filter_params* p, double& r){
        sgnode* a;
        if(!get_filter_param(this, p, "a", a)){
            set_status("Need input node a");
            return false;
        }
        r = // Ranking computation
    }
};

```

8.5.2 Generic Node Filters

There are also a set of generic filters specialized for computations involving nodes. With these you only need to specify a predicate function involving nodes. (Also see `filters/base_node_filters.h`).

There are three types of these filters:

8.5.2.1 Node Test Filters

These filters involve a binary test between two nodes (e.g. intersection or larger). You must specify a test function of the following form:

```
bool node_test(sgnode* a, sgnode* b, const filter_params* p)
```

For an example of how the following base filters are used, see `filters/intersect.cpp`.

node_test_filter

For each input pair (a, b) this outputs the boolean result of `node_test(a, b)`.

node_test_select_filter

For each input pair (a, b) this outputs node b if `node_test(a, b) == true`.
 (Can choose to select b if the test is false by calling `set_select_true(false)`).

8.5.2.2 Node Comparison Filters

These filters involve a numerical comparison between two nodes (e.g. distance). You must specify a comparison function of the following form:

```
double node_comparison(sgnode* a, sgnode* b, const filter_params* p)
```

For an example of how the following base filters are used, see `filters/distance.cpp`.

node_comparison_filter

For each input pair (a, b), outputs the numerical result of `node_comparison(a, b)`.

node_comparison_select_filter

For each input pair (a, b), outputs node b if `min <= node_comparison(a, b) <= max`.
 Min and max can be set through calling `set_min(double)` and `set_max(double)`, or as specified by the user through the `filter_params`.

node_comparison_rank_filter

This outputs the input pair (a, b) for which `node_comparison(a, b)` produces the highest value. To instead have the lowest value output call `set_select_highest(true)`.

8.5.2.3 Node Evaluation Filters

These filters involve a numerical evaluation of a single node (e.g. volume). You must specify an evaluation function of the following form:

```
double node_evaluation(sgnode* a, const filter_params* p)
```

For an example of how the following base filters are used, see `filters/volume.cpp`.

node_evaluation_filter

For each input node a, this outputs the numerical result of `node_evaluation(a)`.

node_evaluation_select_filter

For each input node a, this outputs the node if `min <= node_evaluation(a) <= max`. Min and max can be set through calling `set_min(double)` and `set_max(double)`, or as specified by the user through the `filter_params`.

node_evaluation_rank_filter

This outputs the input node a for which `node_evaluation(a)` produces the highest value. To instead have the lowest value output call `set_select_highest(true)`.

8.6 Command line interface

The user can query and modify the runtime behavior of SVS using the **svs** command. The syntax of this command differs from other Soar commands due to the complexity and object-oriented nature of the SVS implementation. The basic idea is to allow the user to access each object in the SVS implementation (not to be confused with objects in the scene graph) at runtime. Therefore, the command has the form **svs PATH [ARGUMENTS]**, where PATH uniquely identifies an object or the method of an object. ARGUMENTS is a space separated list of strings that each object or function interprets in its own way. For example, **svs S1.scene.world.car** identifies the car object in the scene graph of the top state. As another example, **svs connect_viewer 5999** calls the method to connect to the SVS visualizer with 5999 being the TCP port to connect on. Every path has two special arguments.

- **svs PATH dir** prints all the children of the object at PATH.
- **svs PATH help** prints text about how to use the object, if available.

See Section 9.3.4 on page 229 for more details.

Chapter 9

The Soar User Interface

This chapter describes the set of user interface commands for Soar. All commands and examples are presented as if they are being entered at the Soar command prompt.

This chapter is organized into 7 sections:

1. Basic Commands for Running Soar
2. Examining Memory
3. Configuring Trace Information and Debugging
4. Configuring Soar's Run-Time Parameters
5. File System I/O Commands
6. Soar I/O commands
7. Miscellaneous Commands

Each section begins with a summary description of the commands covered in that section, including the role of the command and its importance to the user. Command syntax and usage are then described fully, in alphabetical order.

The following pages were automatically generated from the git repository at

<https://github.com/SoarGroup/Soar/wiki>

on the date listed on the title page of this manual. Please consult the repository directly for the most accurate and up-to-date information.

For a concise overview of the Soar interface functions, see the Function Summary and Index on page 299. 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.

Note that the command list was revamped and simplified in Soar 9.6.0. While we encourage people to learn the new syntax, aliases and some special mechanism have been added to maintain backwards compatibility with old Soar commands. As a result, many of the sub-commands of the newer commands may use different styles of arguments.

9.1 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. It also describes how to configure some general settings for Soar.

The specific commands described in this section are:

- soar** - Commands and settings related to running Soar. Use **soar ?** for a summary of sub-commands listed below.
- soar init** - Reinitialize Soar so a program can be rerun from scratch.
- soar stop** - Interrupt a running Soar program.
- soar max-chunks** - Limit the number of chunks created during a decision cycle.
- soar max-dc-time** - Set a wall-clock time limit such that the agent will be interrupted when a single decision cycle exceeds this limit.
- soar max-elaborations** - Limit the maximum number of elaboration cycles in a given phase.
- soar max-goal-depth** - Limit the sub-state stack depth.
- soar max-memory-usage** - Set the number of bytes that when exceeded by an agent, will trigger the memory usage exceeded event.

soar max-nil-output-cycles - Limit the maximum number of decision cycles executed without producing output.

soar max-gp - Set the upper-limit to the number of productions generated by the gp command.

soar stop-phase - Controls the phase where agents stop when running by decision.

soar tcl - Controls whether Soar Tcl mode is enabled.

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

soar version - Returns version number of Soar kernel.

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

run - Begin Soar's execution cycle.

exit - Shut down the Soar environment.

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

decide - Commands and settings related to the selection of operators during the Soar decision process

decide indifferent-selection - Controls indifferent preference arbitration.

decide numeric-indifferent-mode - Select method for combining numeric preferences.

decide predict - Predict the next selected operator

decide select - Force the next selected operator

decide set-random-seed - Seed the random number generator.

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

These commands are all frequently used anytime Soar is run.

9.1.1 soar

Commands and settings related to running Soar

Synopsis

===== Soar General Commands and Settings =====	
soar ?	Print this help listing
soar init	Re-initializes Soar
soar stop [--self]	Stop Soar execution
soar version	Print version number
----- Settings -----	
keep-all-top-oprefs	off
max-elaborations	100
	Top state o-supported WMEs
	Maximum elabs in a cycle

max-goal-depth	23	Halt at this depth
max-nil-output-cycles	15	Impasse after nil outputs
max-dc-time	0	Interrupt after time
max-memory-usage	100000000	Threshold for memory warning
max-gp	20000	Max rules gp can generate
stop-phase	apply	Phase before which Soar stop
tcl	off	Allow Tcl code in commands
timers	on	Profile Soar
wait-snc	off	Wait instead of impasse

To change a setting:

`soar <setting> [<value>]`

For a detailed explanation of these settings:

`help soar`

9.1.1.1 Summary View

Using the `soar` command without any arguments will display a summary of Soar's current state of execution and which capabilities of Soar are enabled:

```
=====
Soar 9.6.0 Summary
=====
Enabled: Core, EBC, SMem, EpMem
Disabled: SVS, RL, WMA, SSA
-----
Number of rules: 52
Decisions 20
Elaborations 61
-----
State stack S1, S21 ... S29, S33
Current number of states 5
Next phase apply
-----
```

For a full list of sub-commands and settings: `soar ?`

9.1.1.2 soar init

The `init` command re-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 `soar init` command

allows a Soar program that has been halted to be reset and start its execution from the beginning.

`soar init` 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.

9.1.1.3 soar stop

soar stop [--self]

The **soar stop** 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. The **--self** option will stop only the soar agent where the command is issued. All other agents continue running as previously specified.

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).

Note that if a graphical interface doesn’t periodically do an “update” /flush the pending I/O, then it may not be possible to interrupt a Soar agent from the command line.

9.1.1.4 soar version

This command prints the version of Soar to the screen.

9.1.1.5 Settings

Invoke a sub-command with no arguments to query the current setting. Partial commands are accepted.

Option	Valid Values	Default
keep-all-top-oprefs	on or off	off
max-dc-time	≥ 0	0
max-elaborations	> 0	100
max-goal-depth	> 0	23
max-gp	> 0	20000
max-memory-usage	> 0	1000000000
max-nil-output-cycles	> 0	15
stop-phase		apply
tcl	on or off	off
timers	on or off	on
wait-snc	≥ 1	1

soar keep-all-top-oprefs Enabling `keep-all-top-oprefs` turns off an optimization that reduces memory usage by discarding any internal preferences for WMEs that already have top-level o-support. Turning this setting off allows those preferences to be examined during debugging.

soar max-dc-time `max-dc-time` sets a maximum amount of time a decision cycle is permitted. After output phase, the elapsed decision cycle time is checked to see if it is greater than the old maximum, and the maximum dc time stat is updated (see [stats](#)). At this time, this threshold is also checked. If met or exceeded, Soar stops at the end of the current output phase with an interrupted state.

soar max-elaborations `max-elaborations` sets and prints the maximum number of elaboration cycles allowed in a single decision cycle.

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.

The elaboration phase will end after `max-elaboration` cycles have completed, even if there are more productions eligible to fire or retract; and Soar will proceed to the next phase after a warning message is printed to notify the user. This limits the total number of cycles of parallel production firing but does not limit the total number of productions that can fire during elaboration.

This limit is included in Soar to prevent getting stuck in infinite loops (such as a production that repeatedly fires in one elaboration cycle and retracts in the next); if you see the warning message, it may be a signal that you have a bug your code. However some Soar programs are designed to require a large number of elaboration cycles, so rather than a bug, you may need to increase the value of `max-elaborations`.

`max-elaborations` is checked during both the Propose Phase and the Apply Phase. If Soar runs more than the `max-elaborations` limit in either of these phases, Soar proceeds to the next phase (either Decision or Output) even if quiescence has not been reached.

soar max-goal-depth The `max-goal-depth` command is used to limit the maximum depth of sub-states that an agent can subgoal to. The initial value of this variable is 100; allowable settings are any integer greater than 0. This limit is also included in Soar to prevent getting stuck in an infinite recursive loop, which may come about due to deliberate actions or via an agent bug, such as dropping inadvertently to state-no-change impasses.

soar max-gp `max-gp` is used to limit the number of productions produced by a `gp` command. It is easy to write a `gp` rule that has a combinatorial explosion and hangs for a long time while those productions are added to memory. The `max-gp` setting bounds this.

soar max-memory-usage The `max-memory-usage` setting is used to trigger the memory usage exceeded event. The initial value of this is 100MB (100,000,000); allowable settings are any integer greater than 0.

NOTE: The code supporting this event is not enabled by default because the test can be computationally expensive and is needed only for specific embedded applications. Users may enable the test and event generation by uncommenting code in `mem.cpp`.

soar max-nil-output-cycles `max-nil-output-cycles` 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.

soar stop-phase `stop-phase` allows the user to control which phase Soar stops in. When running by decision cycle it can be helpful to have agents stop at a particular point in its execution cycle. The precise definition is that “running for n decisions and stopping before phase ph means to run until the decision cycle counter has increased by n and then stop when the next phase is ph ”. The phase sequence (as of this writing) is: input, proposal, decision, apply, output. Stopping after one phase is exactly equivalent to stopping before the next phase.

soar tcl The `tcl` setting enables the TCL command line interface. It provides the ability to run Tcl code from any Soar command line. When the `on` parameter is given, all future Soar commands will be passed to a Tcl interpreter for processing.

Note that you cannot turn Tcl on and use Tcl code within the same file. If you’d like to have Tcl turned on automatically when Soar launches, add the `tcl on` command to your `settings.soar` file in the main Soar directory. This activates the mode on initial launch, allowing you to immediately source files that use Tcl code.

Note that `tcl off` is currently not supported due to memory issues.

soar timers This setting 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.

soar wait-snc wait-snc controls an architectural wait state. On some systems, especially those that model expert 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.

9.1.1.6 Examples

```
soar init
soar stop -s
soar timers off
soar stop-phase output           // stop before output phase
soar max-goal-depth 100
soar max-elaborations
```

9.1.1.7 Default Aliases

init	soar init
is	soar init
init-soar	soar init
interrupt	soar stop
ss	soar stop
stop	soar stop
stop-soar	soar stop
gp-max	soar max-gp
max-dc-time	soar max-dc-time
max-elaborations	soar max-elaborations
max-goal-depth	soar max-goal-depth
max-memory-usage	soar max-memory-usage
max-nil-output-cycles	soar max-nil-output-cycles
set-stop-phase	soar stop-phase
timers	soar timers
version	soar version
waitsnc	soar wait-snc

9.1.1.8 See Also

[run](#)
[stats](#)

9.1.2 run

Begin Soar's execution cycle.

Synopsis

```
run -[d|e|o|p] [g] [u|n] [s] [count] [-i e|p|d|o]
```

9.1.2.1 Options

Option	Description
-d, --decision	Run Soar for count decision cycles.
-e, --elaboration	Run Soar for count elaboration cycles.
-o, --output	Run Soar until the nth time output is generated by the agent. Limited by the value of max-nil-output-cycles .
-p, --phase	Run Soar by phases. A phase is either an input phase, proposal phase, decision phase, apply phase, or output phase.
-s, --self	If other agents exist within the kernel, do not run them at this time.
-u, --update	Sets a flag in the update event callback requesting that an environment updates. This is the default if --self is not specified.
-n, --noupdate	Sets a flag in the update event callback requesting that an environment does not update. This is the default if --self is specified.
count	A single integer which specifies the number of cycles to run Soar.
-i, --interleave	Support round robin execution across agents at a finer grain than the run-size parameter. e = elaboration, p = phase, d = decision, o = output
-g, --goal	Run agent until a goal retracts

Deprecated Run Options :

These may be reimplemented in the future.

Option	Description
--operator	Run Soar until the nth time an operator is selected.
--state	Run Soar until the nth time a state is selected.

9.1.2.2 Description

The `run` command starts the Soar execution cycle or continues any execution that was temporarily stopped. The default behavior of `run`, with no arguments, is to cause Soar to execute until it is halted or interrupted by an action of a production, or until an external interrupt is issued by the user. The `run` command can also specify that Soar should run only for a specific number of Soar cycles or phases (which may also be prematurely stopped by a production action or the `stop-soar` command). This is helpful for debugging sessions, where users may want to pay careful attention to the specific productions that are firing and retracting.

The `run` command takes optional arguments: an integer, `count`, which specifies how many units to run; and a `units` flag indicating what steps or increments to use. If `count` is specified, but no `units` are specified, then Soar is run by decision cycles. If `units` are specified, but `count` is unspecified, then `count` defaults to ‘1’. If both are unspecified, Soar will run until either a `halt` is executed, an interrupt is received, or max stack depth is reached.

If there are multiple Soar agents that exist in the same Soar process, then issuing a `run` command in any agent will cause all agents to run with the same set of parameters, unless the flag `--self` is specified, in which case only that agent will execute.

If an environment is registered for the kernel’s update event, then when the event it triggered, the environment will get information about how the `run` was executed. If a `run` was executed with the `--update` option, then the event sends a flag requesting that the environment actually update itself. If a `run` was executed with the `-noupdate` option, then the event sends a flag requesting that the environment not update itself. The `--update` option is the default when `run` is specified without the `--self` option is not specified. If the `--self` option is specified, then the `--noupdate` option is on by default. It is up to the environment to check for these flags and honor them.

Some use cases include:

Option	Description
<code>run --self</code>	runs one agent but not the environment
<code>run --self --update</code>	runs one agent and the environment
<code>run</code>	runs all agents and the environment
<code>run --noupdate</code>	runs all agents but not the environment

9.1.2.3 Setting an interleave size

When there are multiple agents running within the same process, it may be useful to keep agents more closely aligned in their execution cycle than the `run` increment (`--elaboration`, `--phases`, `--decisions`, `--output`) specifies. For instance, it may be necessary to keep agents in “lock step” at the phase level, even though the `run` command issued is for 5 decisions. Some use cases include:

Option	Description
<code>run -d 5 -i p</code>	run the agent one phase and then move to the next agent, looping over agents until they have run for 5 decision cycles
<code>run -o 3 -i d</code>	run the agent one decision cycle and then move to the next agent. When an agent generates output for the 3rd time, it no longer runs even if other agents continue.

The `interleave` parameter must always be equal to or smaller than the specified run parameter.

Note If Soar has been stopped due to a `halt` action, an `init-soar` command must be issued before Soar can be restarted with the `run` command.

9.1.2.4 Default Aliases

<code>d</code>	<code>run -d 1</code>
<code>e</code>	<code>run -e 1</code>
<code>step</code>	<code>run -d 1</code>

9.1.3 exit

Terminates Soar and exits the kernel.

9.1.3.1 Default Aliases

<code>stop</code>	<code>exit</code>
-------------------	-------------------

9.1.4 help

Provide formatted usage information about Soar commands.

Synopsis

`help [command_name]`

9.1.4.1 Default Aliases

- ?
- man

9.1.4.2 Description

This command prints formatted help for the given command name. Issue alone to see what topics have help available.

9.1.5 decide

Commands and settings related to the selection of operators during the Soar decision process

Synopsis

```
=====
-          Decide Sub-Commands and Options      -
=====

decide           [? | help]
-----
decide numeric-indifferent-mode [--avg --sum]
-----
decide indifferent-selection
decide indifferent-selection <policy>
<policy> = [--boltzmann | --epsilon-greedy |
            --first | --last | --softmax ]
decide indifferent-selection <param> [value]
<param> = [--epsilon --temperature]
decide indifferent-selection [--reduction-policy| -p] <param> [<policy>]
decide indifferent-selection [--reduction-rate| -r] <param> <policy> [<rate>]
decide indifferent-selection [--auto-reduce] [setting]
decide indifferent-selection [--stats]
-----
decide predict
decide select           <operator ID>
-----
decide set-random-seed      [<seed>]
-----
For a detailed explanation of sub-commands:    help decide
```

9.1.5.1 Summary Screen

Using the `decide` command without any arguments will display key elements of Soar's current decision settings:

```
=====
Decide Summary
=====
Numeric indifference mode: sum
-----
Exploration Policy: softmax
Automatic Policy Parameter Reduction: off
Epsilon: 0.100000
Epsilon Reduction Policy: exponential
Temperature: 25.000000
Temperature Reduction Policy: exponential
-----
```

Use '`decide ?`' for a command overview or '`help decide`' for the manual page.

9.1.5.2 decide numeric-indifferent-mode

The `numeric-indifferent-mode` command sets how multiple numeric indifferent preference values given to an operator are combined into a single value for use in random selection.

The default procedure is `--sum` which sums all numeric indifferent preference values given to the operator, defaulting to 0 if none exist. The alternative `--avg` mode will average the values, also defaulting to 0 if none exist.

9.1.5.3 decide indifferent-selection

The `indifferent-selection` command allows the user to set options relating to selection between operator proposals that are mutually indifferent in preference memory.

The primary option is the exploration policy (each is covered below). When Soar starts, *softmax* is the default policy.

Note: As of version 9.3.2, the architecture no longer automatically changes the policy to *epsilon-greedy* the first time Soar-RL is enabled.

Some policies have parameters to temper behavior. The indifferent-selection command provides basic facilities to automatically reduce these parameters exponentially and linearly each decision cycle by a fixed rate. In addition to setting these policies/rates, the *auto-reduce* option enables the automatic reduction system (disabled by default), for which the Soar decision cycle incurs a small performance cost.

indifferent-selection options :

Option	Description
-s, --stats	Summary of settings
policy	Set exploration policy
parameter [exploration policy parameters]	Get/Set exploration policy parameters (if value not given, returns the current value)
parameter [reduction_policy] (value)	Get/Set exploration policy parameter reduction policy (if policy not given, returns the current)
parameter reduction_policy [exploration policy parameter]	Get/Set exploration policy parameter reduction rate for a policy (if rate not give, returns the current)
-a, --auto-reduce [on,off] (reduction-rate)	Get/Set auto-reduction setting (if setting not provided, returns the current)

indifferent-selection exploration policies :

Option	Description
-b, --boltzmann	Tempered softmax (uses temperature)
-g, --epsilon-greedy	Tempered greedy (uses epsilon)
-x, --softmax	Random, biased by numeric indifferent values (if a non-positive value is encountered, resorts to a uniform random selection)
-f, --first	Deterministic, first indifferent preference is selected
-l, --last	Deterministic, last indifferent preference is selected

indifferent-selection exploration policy parameters :

Parameter Name	Acceptable Values	Default Value
-e, --epsilon	[0, 1]	0.1
-t, --temperature	(0, inf)	25

indifferent-selection auto-reduction policies :

Parameter Name	Acceptable Values	Default Value
exponential default	[0, 1]	1
linear	[0, inf]	0

9.1.5.4 decide predict

The predict command determines, based upon current operator proposals, which operator will be chosen during the next decision phase. If predict determines an operator tie will be encountered, “tie” is returned. If predict determines no operator will be selected (state no-change), “none” is returned. If predict determines a conflict will arise during the decision phase, “conflict” is returned. If predict determines a constraint failure will occur, “constraint” is returned. Otherwise, predict will return the id of the operator to be chosen. If operator selection will require probabilistic selection, and no alterations to the probabilities are made between the call to predict and decision phase, predict will manipulate the random number generator to enforce its prediction.

9.1.5.5 decide select

The select command will force the selection of an operator, whose id is supplied as an argument, during the next decision phase. If the argument is not a proposed operator in the next decision phase, an error is raised and operator selection proceeds as if the select command had not been called. Otherwise, the supplied operator will be selected as the next operator, regardless of preferences. If select is called with no id argument, the command returns the operator id currently forced for selection (by a previous call to select), if one exists.

Example Assuming operator “O2” is a valid operator, this would select it as the next operator to be selected:

```
decide select 02
```

9.1.5.6 decide set-random-seed

Seeds the random number generator with the passed seed. Calling `decide set-random-seed` (or equivalently, `decide srand`) without providing a seed will seed the generator based on the contents of /dev/urandom (if available) or else based on time() and clock() values.

Example

```
decide set-random-seed 23
```

9.1.5.7 Default Aliases

inds	indifferent-selection
srand	set-random-seed

9.1.5.8 See Also

[rl](#)

9.1.6 alias

Define a new alias of existing commands and arguments.

Synopsis

```
alias
alias <name> [args]
alias -r <name>
```

9.1.6.1 Adding a new alias

This command defines new aliases by creating Soar procedures with the given name. The new procedure can then take an arbitrary number of arguments which are post-pended to the given definition and then that entire string is executed as a command. The definition must be a single command, multiple commands are not allowed. The alias procedure checks to see if the name already exists, and does not destroy existing procedures or aliases by the same name. Existing aliases can be removed by using the [unalias](#) command.

9.1.6.2 Removing an existing alias

To undefine a previously created alias, use the **-r** argument along with the name of the alias to remove.

```
alias -r existing-alias
```

Note: If you are trying to create an alias for a command that also has a **-r** option, make sure to enclose it in quotes. For example:

```
alias unalias "alias -r"
```

9.1.6.3 Printing Existing Aliases

With no arguments, alias returns the list of defined aliases. With only the name given, alias returns the current definition.

9.1.6.4 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
```

9.1.6.5 Default Alias Aliases

```
a           alias
unalias, un   alias -r
```

9.2 Procedural Memory Commands

This section describes the commands used to create and delete Soar productions, to see what productions will match and fire in the next Propose or Apply phase, to watch when specific productions fire and retract, and to configure options for selecting between mutually indifferent operators, along with various other methods for examining the contents and statistics of procedural memory.

The specific commands described in this section are:

sp - Create a production and add it to production memory.

gp - Define a pattern used to generate and source a set of Soar productions.

production - Commands to manipulate Soar rules and analyze their usage

production break - Set interrupt flag on specific productions.

production excise - This command removes productions from Soar's memory.

production find - Find productions that contain a given pattern.

production firing-counts - Print the number of times productions have fired.

production matches - Print information about the match set and partial matches.

production memory-usage - Print memory usage for production matches.

production optimize-attribute - Declare an attribute as multi-attributes so as to increase Rete production matching efficiency.

production watch - Trace firings and retractions of specific productions.

`sp` is of course used in virtually all Soar programming. Of the remaining commands, **production matches** and **production memory-usage** are most often used. **production find** is especially useful when the number of productions loaded is high. **production firing-counts** is used to see if how many times certain rules fire. **production watch** is related to `wm watch`, but applies only to specific, named productions.

9.2.1 sp

Define a Soar production.

Synopsis

```
sp {production_body}
```

9.2.1.1 Options

Option	Description
<code>production_body</code>	A Soar production.

9.2.1.2 Description

The `sp` command creates a new production and loads it into production memory. `production_body` is a single argument parsed by the Soar kernel, so it should be enclosed in curly braces to avoid being parsed by other scripting languages that might be in the same process. The overall syntax of a rule is as follows:

```
name
  ["documentation-string"]
  [FLAG*]
  LHS
  -->
  RHS
```

The first element of a rule is its name. 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 \rightarrow 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. (See the *Syntax of Soar Programs* chapter of the manual for naming conventions and 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).

Rules matching the following requirement are flagged upon being created/sourced: a rule is a Soar-RL rule if and only if its right hand side (RHS) consists of a single numeric preference and it is not a template rule (see FLAGs below). This format exists to ease technical requirements of identifying/updating Soar-RL rules, as well as to make it easy for the agent programmer to add/maintain RL capabilities within an agent. (See the *Reinforcement Learning* chapter of the manual for further details.)

9.2.1.3 Rule Flags

The optional flags are given below. Note that these switches are preceded 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
:i-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 trace task)
:chunk	specifies that this production is a chunk (this matters for learn trace)
:interrupt	specifies that Soar should stop running when this production matches but before it fires (this is a useful debugging tool)
:template	specifies that this production should be used to generate new reinforcement learning rules by filling in those variables that match constants in working memory

Multiple flags may be used, but not both of o-support and no-support.

Although you could force your productions to provide o-support or i-support by using these commands — regardless of the structure of the conditions and actions of the production —

this is not proper coding style. The `o-support` and `i-support` flags are included to help with debugging, but should not be used in a standard Soar program.

9.2.1.4 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)
}
```

9.2.1.5 See Also

[production](#)
[chunk](#)
[trace](#)

9.2.2 gp

Generate productions according to a specified pattern.

Synopsis

```
gp { production_body }
```

9.2.2.1 Description

The `gp` command defines a pattern used to generate and source a set of Soar productions. `production_body` is a single argument that looks almost identical to a standard Soar rule that would be used with the `sp` command. Indeed, any syntax that is allowed in `sp` is also allowed in `gp`.

Patterns in `gp` are specified with sets of whitespace-separated values in square brackets. Every combination of values across all square-bracketed value lists will be generated. Values with whitespaces can be used if wrapped in pipes. Characters can also be escaped with a backslash (so string literals with embedded pipes and spaces outside of string literals are both possible).

`gp` is primarily intended as an alternative to `:template` rules for reinforcement learning. `:template` rules generate new rules as patterns occur at run time. Unfortunately, this

incurs a high run time cost. If all possible values are known in advance, then the rules can be generated using gp at source time, thus allowing code to run faster. gp is not appropriate when all possible values are not known or if the total number of possible rules is very large (and the system is likely to encounter only a small subset at run time). It is also possible to combine gp and :template (e.g., if some of the values are known and not others). This should reduce the run time cost of :template.

There is nothing that actually restricts gp to being used for RL, although for non-RL rules, a disjunction list (using << and >>) is better where it can be used. More esoteric uses may include multiple bracketed value lists inside a disjunction list, or even variables in bracketed value lists.

Each rule generated by gp has *integer appended to its name (where integer is some incrementing number).

9.2.2.2 Examples

Template version of rule:

```
sp {water-jug*fill
    :template
    (state <s1> ^name water-jug ^operator <op> +
        ^jug <j1> <j2>)
    (<op> ^name fill ^fill-jug.volume <fv>)
    (<j1> ^volume 3 ^contents <c1>)
    (<j2> ^volume 5 ^contents <c2>)
-->
    (<s1> ^operator <op> = 0)
}
```

gp version of rule (generates 144 rules):

```
gp {water-jug*fill
    (state <s1> ^name water-jug ^operator <op> +
        ^jug <j1> <j2>)
    (<op> ^name fill ^fill-jug.volume [3 5])
    (<j1> ^volume 3 ^contents [0 1 2 3])
    (<j2> ^volume 5 ^contents [0 1 2 3 4 5])
-->
    (<s1> ^operator <op> = 0)
}
```

Esoteric example (generates 24 rules):

```
gp {strange-example
    (state <s1> ^<< [att1 att2] [att3 att4] >> [ val |another val| |strange val|| ])
-->
    (<s1> ^foo [bar <bar>])
```

```
}
```

`testgp.soar` contains many more examples.

9.2.2.3 See Also

`sp`

9.2.3 production

Commands to manipulate Soar rules and analyze their usage.

Synopsis

```
=====
-           Production Sub-Commands and Options
=====
production      [? | help]
-----
production break      [--clear --print]
production break      --set <prod-name>
-----
production excise      <production-name>
production excise      [--all --chunks --default ]
production excise      [--never-fired --rl      ]
production excise      [--task --templates --user]
-----
production find      [--lhs --rhs      ] <pattern>
production find      [--show-bindings   ]
production find      [--chunks --nochunks ]
-----
production firing-counts      [--all --chunks --default --rl]  [n]
production firing-counts      [--task --templates --user --fired]
production firing-counts      <prod-name>
-----
production matches      [--names --count  ] <prod-name>
production matches      [--timetags --wmes]
production matches      [--names --count  ] [--assertions ]
production matches      [--timetags --wmes] [--retractions]
-----
production memory-usage      [options] [max]
production memory-usage      <production_name>
```

```
-----
production optimize-attribute [symbol [n]]
-----
production watch           [--disable --enable] <prod-name>
-----
```

For a detailed explanation of sub-commands: help production

9.2.3.1 Summary Screen

Using the `production` command without any arguments will display a summary of how many rules are loaded into memory:

```
=====
-          Productions          -
=====
User rules                      0
Default rules                   0
Chunks                          0
Justifications                  0
-----
Total                           0
-----
Use 'production ?' to learn more about the command
```

9.2.3.2 production break

Toggles the `:interrupt` flag on a rule at run-time, which stops the Soar decision cycle when the rule fires. The `break` command can be used to toggle the `:interrupt` flag on production rules which did not have it set in the original source file, which stops the Soar decision cycle when the rule fires. This is intended to be used for debugging purposes.

Synopsis

```
production break -c|--clear <production-name>
production break -p|--print
production break -s|--set <production-name>
production break <production-name>
```

Options :

Parameter	Argument	Description
-c, --clear	<production-name>	Clear <code>:interrupt</code> flag from a production.

Parameter	Argument	Description
-p, --print	(none)	Print which production rules have had their :interrupt flags set.
(none)	(none)	Print which production rules have had their :interrupt flags set.
-s, --set	<production-name>	Set :interrupt flag on a production rule.
(none)	<production-name>	Set flag :interrupt on a production rule.

9.2.3.3 production excise

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.

Note: As of Soar 9.6, using the flag -a or --all no longer causes a `soar init`.

Synopsis

```
production excise production_name
production excise options
```

Options :

Option	Description
-a, --all	Remove all productions from memory and perform an <code>init-soar</code> command
-c, --chunks	Remove all chunks (learned productions) and justifications from memory
-d, --default	Remove all default productions (<code>:default</code>) from memory
-n, --never-fired	Excise rules that have a firing count of 0
-r, --rl	Excise Soar-RL rules
-t, --task	Remove chunks, justifications, and user productions from memory
-T, --templates	Excise Soar-RL templates
-u, --user	Remove all user productions (but not chunks or default rules) from memory
production_name	Remove the specific production with this name.

Examples :

This command removes the production `my*first*production` and all chunks:

```
production excise my*first*production --chunks
```

This removes all productions:

```
production excise --all
```

9.2.3.4 production find

Find productions by condition or action patterns.

Synopsis

```
production find [-lrs[n|c]] pattern
```

Options :

Option	Description
<code>-c, --chunks</code>	Look <i>only</i> for chunks that match the pattern.
<code>-l, --lhs</code>	Match pattern only against the conditions (left-hand side) of productions (default).
<code>-n, --nochunks</code>	<i>Disregard</i> chunks when looking for the pattern.
<code>-r, --rhs</code>	Match pattern against the actions (right-hand side) of productions.
<code>-s, --show-bindings</code>	Show the bindings associated with a wildcard pattern.
<code>pattern</code>	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.

Production Find Examples :

Find productions that test that some object `gumby` has an attribute `alive` with value `t`. In addition, limit the rules to only those that test an operator named `foo`:

```
production find (<state> ^gumby <gv> ^operator.name foo)(<gv> ^alive t)
```

Note that in the above command, `<state>` does not have to match the exact variable name used in the production.

Find productions that propose the operator `foo`:

```
production find --rhs (<x> ^operator <op> +)(<op> ^name foo)
```

Find chunks that test the attribute `^pokey`:

```
production find --chunks (<x> ^pokey *)
```

Examples using the water-jugs demo:

```
source demos/water-jug/water-jug.soar
production-find (<s> ^name *)(<j> ^volume *)
production-find (<s> ^name *)(<j> ^volume 3)
production-find --rhs (<j> ^* <volume>)
```

9.2.3.5 production firing-counts

Print the number of times productions have fired.

Synopsis

```
production firing-counts [type] [n]
production firing-counts production_name
```

Options :

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

Option	Description
<code>n</code>	List the top <code>n</code> productions. If <code>n</code> is 0, only the productions which haven't fired are listed
<code>production_name</code>	Print how many times a specific production has fired
<code>-f, --fired</code>	Prints only rules that have fired
<code>-c, --chunks</code>	Print how many times chunks (learned rules) fired

Option	Description
-j, --justifications	Print how many times justifications fired
-d, --default	Print how many times default productions (:default) fired
-r, --rl	Print how many times Soar-RL rules fired
-T, --templates	Print how many times Soar-RL templates fired
-u, --user	Print how many times user productions (but not chunks or default rules) fired

Description The `production firing-counts` command prints the number of times each production has fired; production names are given from most frequently 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 `-fired` is used, the opposite happens. Only rules that have fired are listed. If a production name is given as an argument, the firing count for that production is printed.

Note that firing counts are reset by a call to `[soar init]` (`cmd_soar`).

Examples :

This example prints the 10 productions which have fired the most times along with their firing counts:

```
production firing-counts 10
```

This example prints the firing counts of production `my*first*production`:

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

This example prints all rules that have fired at least once:

```
production firing-counts -f
```

9.2.3.6 production matches

The `production 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.

Synopsis

```
production matches [options] production_name
production matches [options] -[a|r]
```

Options :

Option	Description
<code>production_name</code>	Print partial match information for the named production.
<code>-n, --names, -c, --count</code>	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.
<code>-t, --timetags</code>	Also print the timetags of the wmes at the first failing condition
<code>-w, --wmes</code>	Also print the full wmes, not just the timetags, at the first failing condition.
<code>-a, --assertions</code>	List only productions about to fire.
<code>-r, --retractions</code>	List only productions about to retract.

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 can be specified to restrict the output to print only the assertions or retractions.

Printing partial matches for productions In addition to printing the current match set, the `matches` command can be used to print information about partial matches for a named production. In this case, the conditions of the production are listed, each preceded by the number of currently active matches for that condition. If a condition is negated, it is preceded by a minus sign `-`. The pointer `>>>` before a condition indicates that this is the first condition that failed to match.

When printing partial matches, the default action is to print only the counts of the number of WME's that match, and is a handy tool for determining which condition failed to match for a production that you thought should have fired. At levels `--timetags` and `--wmes` the `matches` command displays the WME's immediately after the first condition that failed to match – temporarily interrupting the printing of the production conditions themselves.

Notes :

When printing partial match information, some of the matches displayed by this command may have already fired, depending on when in the execution cycle this command is called. To check for the matches that are about to fire, use the `matches` command without a named production.

In Soar 8, the execution cycle (decision cycle) is input, propose, decide, apply output; it no longer stops for user input after the decision phase when [running](#) by decision cycles (`run -d 1`). If a user wishes to print the match set immediately after the decision phase and before the apply phase, then the user must run Soar by *phases* (`run -p 1`).

Examples :

This example prints the productions which are about to fire and the WMEs that match the productions on their left-hand sides:

```
production matches --assertions --wmes
```

This example prints the WME timetags for a single production.

```
production matches -t my*first*production
```

9.2.3.7 production memory-usage

Print memory usage for partial matches.

Synopsis

```
production memory-usage [options] [number]
production memory-usage production_name
```

Options :

Option	Description
<code>-c, --chunks</code>	Print memory usage of chunks.
<code>-d, --default</code>	Print memory usage of default productions.
<code>-j, --justifications</code>	Print memory usage of justifications.
<code>-u, --user</code>	Print memory usage of user-defined productions.
<code>production_name</code>	Print memory usage for a specific production.
<code>number</code>	Number of productions to print, sorted by those that use the most memory.
<code>-T, --template</code>	Print memory usage of Soar-RL templates.

Description The `memory-usage` 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 `memory-usage` command prints memory usage for all productions. If a `production_name` is specified, memory usage will be printed only for that production. If a positive integer `number` is given, only `number` productions will be printed: the `number` productions that use the most memory. Output may be restricted to

print memory usage for particular types of productions using the command options.

Memory usage is recorded according to the tokens that are allocated in the Rete network for the given production(s). This number is a function of the number of elements in working memory that match each production. Therefore, this command will not provide useful information at the beginning of a Soar run (when working memory is empty) and should be called in the middle (or at the end) of a Soar run.

The `memory-usage` 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.

9.2.3.8 `production optimize-attribute`

Declare a symbol to be multi-attributed so that conditions in productions that test that attribute are re-ordered so that the rule can be matched more efficiently.

Synopsis

```
production optimize-attribute [symbol [n]]
```

Options :

Option	Description
<code>symbol</code>	Any Soar attribute.
<code>n</code>	Integer greater than 1, estimate of degree of simultaneous values for attribute.

Description :

This command is used to improve efficiency of matching against attributes that can have multiple values at once.

```
(S1 ^foo bar1)
(S1 ^foo bar2)
(S1 ^foo bar3)
```

If you know that a certain attribute will take on multiple values, `optimize-attribute` can be used to provide hints to the production condition reorderer so that it can produce better orderings that allow the Rete network to match faster. This command has no effect on the actual contents of working memory and is only used to improve efficiency in problematic situations.

`optimize-attribute` declares a symbol to be an attribute which can take on multiple values. The optional `n` is an integer (greater than 1) indicating an upper limit on the number of expected values that will appear for an attribute. If `n` is not specified, the value 10 is used for each declared multi-attribute. More informed values will tend to result in greater efficiency.

Note that `optimize-attribute` declarations must be made before productions are loaded into soar or this command will have no effect.

Example :

Declare the symbol “thing” to be an attribute likely to take more than 1 but no more than 4 values:

```
production optimize-attribute thing 4
```

9.2.3.9 production watch

Trace firings and retractions of specific productions.

Synopsis

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

Options :

Option	Description
<code>-d, --disable, --off</code>	Turn production watching off for the specified production. If no production is specified, turn production watching off for all productions.
<code>-e, --enable, --on</code>	Turn production watching on for the specified production. The use of this flag is optional, so this is watch’s default behavior. If no production is specified, all productions currently being watched are listed.
<code>production name</code>	The name of the production to watch.

Description The production `watch` 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, production `watch` lists the productions currently being traced. With one production-name argument, production `watch` enables tracing the production; `--enable` can be explicitly stated, but it is the default action.

If `--disable` is specified followed by a production-name, tracing is turned off for the production. When no production-name is specified, `--enable` lists all productions currently being traced, and `--disable` disables tracing of all productions.

Note that production `watch` now only takes one production per command. Use multiple times to watch multiple functions.

9.2.3.10 Default Aliases

<code>ex</code>	<code>production excise</code>
<code>excise</code>	<code>production excise</code>
<code>fc</code>	<code>production firing-counts</code>
<code>firing-counts</code>	<code>production firing-counts</code>
<code>matches</code>	<code>production matches</code>
<code>memories</code>	<code>production memory-usage</code>
<code>multi-attributes</code>	<code>production optimize-attribute</code>
<code>pbreak</code>	<code>production break</code>
<code>production-find</code>	<code>production find</code>
<code>pw</code>	<code>production watch</code>
<code>pwatch</code>	<code>production watch</code>

9.2.3.11 See Also

[soar init](#)
[sp](#)
[trace](#)

9.3 Short-term Memory Commands

This section describes the commands for interacting with working memory and preference memory, seeing what productions will match and fire in the next Propose or Apply phase, and examining the goal dependency set. These commands are particularly useful when running or debugging Soar, as they let users see what Soar is “thinking.” Also included

in this section is information about using Soar's Spatial Visual System (SVS), which filters perceptual input into a form usable in symbolic working memory.

The specific commands described in this section are:

print - Print items in working, semantic and production memory. Can also print the print the WMEs in the goal dependency set for each goal.

wm Commands and settings related to working memory and working memory activation.

wm activation - Get/Set working memory activation parameters.

wm add - Manually add an element to working memory.

wm remove - Manually remove an element from working memory.

wm watch - Print information about wmes that match a certain pattern as they are added and removed.

preferences - Examine items in preference memory.

svs - Perform spatial visual system commands.

Of these commands, **print** is the most often used (and the most complex). **print --gds** is useful for examining the goal dependency set when subgoals seem to be disappearing unexpectedly. **preferences** is used to examine which candidate operators have been proposed.

9.3.1 print

Print items in working memory or production memory.

Synopsis

```
print [options] [production_name]
print [options] identifier|timetag|pattern
print [--gds --stack]
```

9.3.1.1 Options

Production printing options :

Option	Description
-a, --all	print the names of all productions currently loaded
-c, --chunks	print the names of all chunks currently loaded

Option	Description
<code>-D, --defaults</code>	print the names of all default productions currently loaded
<code>-j, --justifications</code>	print the names of all justifications currently loaded.
<code>-r, --rl</code>	Print Soar-RL rules
<code>-T, --template</code>	Print Soar-RL templates
<code>-u, --user</code>	print the names of all user productions currently loaded
<code>production_name</code>	print the production named <code>production-name</code>

Production print formatting :

Option	Description
<code>-f, --full</code>	When printing productions, print the whole production. This is the default when printing a named production.
<code>-F, --filename</code>	also prints the name of the file that contains the production.
<code>-i, --internal</code>	items should be printed in their internal form. For productions, this means leaving conditions in their reordered (rete net) form.
<code>-n, --name</code>	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.

Working memory printing options :

Option	Description
<code>-d, --depth n</code>	This option overrides the default printing depth (see the default-wme-depth command for more detail).
<code>-e, --exact</code>	Print only the wmes that match the pattern
<code>-i, --internal</code>	items should be printed in their internal form. For working memory, this means printing the individual elements with their timetags and activation, rather than the objects.
<code>-t, --tree</code>	wmes should be printed in a tree form (one wme per line).
<code>-v, --varprint</code>	Print identifiers enclosed in angle brackets.

Option	Description
<code>identifier</code>	print the object <code>identifier</code> . <code>identifier</code> must be a valid Soar symbol such as <code>S1</code>
<code>pattern</code>	print the object whose working memory elements matching the given <code>pattern</code> . See Description for more information on printing objects matching a specific <code>pattern</code> .
<code>timetag</code>	print the object in working memory with the given <code>timetag</code>

Subgoal stack printing options :

Option	Description
<code>-s, --stack</code>	Specifies that the Soar goal stack should be printed. By default this includes both states and operators.
<code>-o, --operators</code>	When printing the stack, print only <i>operators</i> .
<code>-S, --states</code>	When printing the stack, print only <i>states</i> .

9.3.1.2 Printing the Goal Dependency Set

```
:  
print --gds
```

The Goal Dependency Set (GDS) is described in a subsection of the [The Soar Architecture](#) chapter of the manual. This command is a debugging command for examining the GDS for each goal in the stack. First it steps through all the working memory elements in the rete, looking for any that are included in *any* goal dependency set, and prints each one. Then it also lists each goal in the stack and prints the wmes in the goal dependency set for that particular goal. This command is useful when trying to determine why subgoals are disappearing unexpectedly: often something has changed in the goal dependency set, causing a subgoal to be regenerated prior to producing a result.

`print --gds` is horribly inefficient and should not generally be used except when something is going wrong and you need to examine the Goal Dependency Set.

9.3.1.3 Description

The `print` command is used to print items from production memory or working memory. It can take several kinds of arguments. When printing items from working memory, the Soar objects are printed unless the `--internal` flag is used, in which case the wmes themselves are printed.

```
(identifier ^attribute value [activation] [+])
```

The activation value is only printed if activation is turned on. See [wma](#).

The pattern is surrounded by parentheses. The `identifier`, `attribute`, and `value` must be valid Soar symbols or the wildcard symbol * which matches all occurrences. The optional + symbol restricts pattern matches to acceptable preferences. If wildcards are included, an object will be printed for each pattern match, even if this results in the same object being printed multiple times.

9.3.1.4 Examples

Print the objects in working memory (and their timetags) which have wmes with identifier `s1` and value `v2` (note: this will print the entire `s1` object for each match found):

```
print --internal (s1 ^* v2)
```

Print the Soar stack which includes states and operators:

```
print --stack
```

Print the named production in its RETE form:

```
print -if named*production
```

Print the names of all user productions currently loaded:

```
print -u
```

Default print vs tree print:

```
print s1 --depth 2
(S1 ^io I1 ^reward-link R1 ^superstate nil ^type state)
(I1 ^input-link I2 ^output-link I3)
```

```
print s1 --depth 2 --tree
(S1 ^io I1)
(I1 ^input-link I2)
(I1 ^output-link I3)
(S1 ^reward-link R1)
(S1 ^superstate nil)
(S1 ^type state)
```

9.3.1.5 Default Aliases

p print

pc print -chunks

ps print -stack

wmes print -depth 0 -internal

```
varprint print -varprint -depth 100
gds_print print -gds
```

9.3.1.6 See Also

[output](#)

[trace](#)

[wm](#)

9.3.2 **wm**

Commands and settings related to working memory and working memory activation. There are four sub-commands: `add`, `remove`, `activation`, and `watch`.

Synopsis

```
=====
-          WM Sub-Commands and Options      -
=====

wm          [? | help]

-----
wm add      <id> [^]<attribute> <value> [+]
wm remove   <timetag>

-----
wm activation --get <parameter>
                --set <parameter>           <value>
                    activation          [ on | OFF ]
                    petrov-approx        [ on | OFF ]
                    forgetting          [ on | OFF ]
                    fake-forgetting       [ on | OFF ]
                    forget-wme            all [all, lti]
                    decay-rate            -0.5 [0 to 1]
                    decay-thresh          -2 [0 to infinity]
                    max-pow-cache         10 MB
                    timers                off [off, one]
                --history <timetag>
                --stats               Print forget stats
                --timers [<timer>]        Print timing results
                    <timer> = wma_forgetting or wma_history

-----
wm watch    --add-filter   --type <t>  pattern
                --remove-filter --type <t>  pattern
                --list-filter  [--type <t>]
                --reset-filter [--type <t>]
```

```
<t> = adds, removes or both
```

```
For a detailed explanation of sub-commands:      help wm
```

9.3.2.1 wm activation

The `wm activation` command changes the behavior of and displays information about working memory activation.

To get the activation of individual WMEs, use `print -i`.

To get the reference history of an individual WME, use `wm activation -h|--history<timetag>`. For example:

```
print --internal s1
(4000016: S1 ^ct 1000000 [3.6])
(4: S1 ^epmem E1 [1])
(11: S1 ^io I1 [1])
(20: S1 ^max 1000000 [3.4])
(18: S1 ^name ct [3.4])
(4000018: S1 ^operator 01000001 [1] +)
(4000019: S1 ^operator 01000001 [1])
(3: S1 ^reward-link R1 [1])
(8: S1 ^smem S2 [1])
(2: S1 ^superstate nil [1])
(14: S1 ^top-state S1 [1])
(1: S1 ^type state [1])
```

The bracketed values are activation. To get the history of an individual element:

```
wm activation --history 18
history (60/5999999, first @ d1):
 6 @ d1000000 (-1)
 6 @ d999999 (-2)
 6 @ d999998 (-3)
 6 @ d999997 (-4)
 6 @ d999996 (-5)
 6 @ d999995 (-6)
 6 @ d999994 (-7)
 6 @ d999993 (-8)
 6 @ d999992 (-9)
 6 @ d999991 (-10)
```

```
considering WME for decay @ d1019615
```

This shows the last 60 references (of 5999999 in total, where the first occurred at decision cycle 1). For each reference, it says how many references occurred in the cycle (such as 6 at

decision 1000000, which was one cycle ago at the time of executing this command). Note that references during the current cycle will not be reflected in this command (or computed activation value) until the end of output phase. If `forgetting` is `on`, this command will also display the cycle during which the WME will be considered for decay. Even if the WME is not referenced until then, this is not necessarily the cycle at which the WME will be forgotten. However, it is guaranteed that the WME will not be forgotten before this cycle.

Options :

Option	Description
<code>-g</code> , <code>--get</code>	Print current parameter setting
<code>-s</code> , <code>--set</code>	Set parameter value
<code>-S</code> , <code>--stats</code>	Print statistic summary or specific statistic
<code>-t</code> , <code>--timers</code>	Print timer summary or specific timer
<code>-h</code> , <code>--history</code>	Print reference history of a WME

Parameters :

The `activation` command uses the `--get|--set <parameter> <value>` convention rather than individual switches for each parameter. Running `wm activation` without any switches displays a summary of the parameter settings.

Parameter	Description	Possible values	Default
<code>activation</code>	Enable working memory activation	<code>on</code> , <code>off</code>	<code>off</code>
<code>decay-rate</code>	WME decay factor	<code>[0, 1]</code>	<code>0.5</code>
<code>decay-thresh</code>	Forgetting threshold	<code>(0, inf)</code>	<code>2.0</code>
<code>forgetting</code>	Enable removal of WMEs with low activation values	<code>on</code> , <code>off</code>	<code>off</code>
<code>forget-wme</code>	If <code>lti</code> only remove WMEs with a long-term id	<code>all</code> , <code>lti</code>	<code>all</code>
<code>max-pow-cache</code>	Maximum size, in MB, for the internal pow cache	<code>1, 2, ...</code>	<code>10</code>
<code>petrov-approx</code>	Enables the (Petrov 2006) long-tail approximation	<code>on</code> , <code>off</code>	<code>off</code>
<code>timers</code>	Timer granularity	<code>off</code> , <code>one</code>	<code>off</code>

The `decay-rate` and `decay-thresh` parameters are entered as positive decimals, but are internally converted to, and printed out as, negative.

The `petrov-approx` may provide additional validity to the activation value, but comes at a significant computational cost, as the model includes unbounded positive exponential computations, which cannot be reasonably cached.

When `activation` is enabled, the system produces a cache of results of calls to the `pow` function, as these can be expensive during runtime. The size of the cache is based upon three run-time parameters (`decay-rate`, `decay-thresh`, and `max-pow-cache`), and one compile time parameter, `WMA_REFERENCES_PER_DECISION` (default value of 50), which estimates the maximum number of times a WME will be referenced during a decision. The cache is composed of `double` variables (i.e. 64-bits, currently) and the number of cache items is computed as follows:

$$e^{((\text{decay_thresh} - \ln(\text{max_refs})) / \text{decay_rate})}$$

With the current default parameter values, this will incur about 1.04MB of memory. Holding the `decay-rate` constant, reasonable changes to `decay-thresh` (i.e. +/- 5) does not greatly change this value. However, small changes to `decay-rate` will dramatically change this profile. For instance, keeping everything else constant, a `decay-thresh` of 0.3 requires ~2.7GB and 0.2 requires ~50TB. Thus, the `max-pow-cache` parameter serves to allow you to control the space vs. time tradeoff by capping the maximum amount of memory used by this cache. If `max-pow-cache` is much smaller than the result of the equation above, you may experience somewhat degraded performance due to relatively frequent system calls to `pow`.

If `forget-wme` is `lti` and `forgetting` is `on`, only those WMEs whose id is a long-term identifier at the decision of forgetting will be removed from working memory. If, for instance, the id is stored to semantic memory after the decision of forgetting, the WME will not be removed till some time after the next WME reference (such as testing/creation by a rule).

Statistics Working memory activation tracks statistics over the lifetime of the agent. These can be accessed using `wm activation --stats <statistic>`.

Running `wm activation --stats` without a statistic will list the values of all statistics. Unlike timers, statistics will always be updated.

Available statistics are:

Name	Label	Description
<code>forgotten-wmes</code>	Forgotten WMEs	Number of WMEs removed from working memory due to forgetting

Timers Working memory activation also has a set of internal timers that record the durations of certain operations. Because fine-grained timing can incur runtime costs, working memory activation timers are off by default. Timers of different levels of detail can be turned on by issuing `wm activation --set timers <level>`, where the levels can be `off` or `one`, `one` being most detailed and resulting in all timers being turned on. Note that none of the working memory activation statistics nor timing information is reported by the `stats` command.

All timer values are reported in seconds.

Timer Levels:

Option	Description
wma_forgetting	Time to process forgetting operations each cycle
wma_history	Time to consolidate reference histories each cycle

9.3.2.2 `wm add`

Manually add an element to working memory.

```
wm add id [^]attribute value [+]
```

Options :

Option	Description
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. `wm add` is often used by an input function to update Soar's information about the state of the external world.

`wm add` 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:

```
wm add 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.

```
wm add Z2 message * +
```

Warnings Be careful how you use this command. It may have weird side effects (possibly even including system crashes). For example, the chunking mechanism can't backtrace through WMEs created via `wm add` nor will such WMEs ever be removed through Soar's garbage collection. Manually removing context/impasse WMEs may have unexpected side effects.

9.3.2.3 `wm remove`

Manually remove an element from working memory.

```
wm remove timetag
```

Options :

Option	Description
<code>timetag</code>	A positive integer matching the timetag of an existing working memory element.

Description The `wm remove` 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, `wm remove` should only be called from registered input functions to delete working memory elements on Soar's input link.

Beware of weird side effects, including system crashes.

Warnings `wm remove` should never be called from the RHS of a production: 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.

9.3.2.4 **wm watch**

Print information about WMEs matching a certain pattern as they are added and removed.

```
wm watch -[a|r] -t <type> >pattern>
wm watch -[l|R] [-t <type>]
```

Options :

Option	Description
-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.

Watch Patterns :

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 matches any string. Note that braces are not used anymore.

Watch Types When using the -t flag, it must be followed by one of the following:

Option	Description
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 command 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.

Examples Users can watch an `attribute` of a particular object (as long as that object already exists):

```
soar> wm watch --add-filter -t both D1 speed *
```

or print WMEs that retract in a specific state (provided the `state` already exists):

```
soar> wm watch --add-filter -t removes S3 * *
```

or watch any relationship between objects:

```
soar> wm watch --add-filter -t both * ontop *
```

9.3.2.5 Default Aliases

add-wme	wm add
aw	wm add
remove-wme	wm remove
rw	wm remove
watch-wmes	wm watch
wma	wm activation

9.3.2.6 See Also

[print](#)
[trace](#)

9.3.3 preferences

Examine details about the preferences that support the specified *identifier* and *attribute*.

Synopsis

```
preferences [options] [identifier [attribute]]
```

9.3.3.1 Options

Option	Description
-0, -n, --none	Print just the preferences themselves

Option	Description
-1, -N, --names	Print the preferences and the names of the productions that generated them
-2, -t, --timetags	Print the information for the --names option above plus the timetags of the wmes matched by the LHS of the indicated productions
-3, -w, --wmes	Print the information for the --timetags option above plus the entire WME matched on the LHS.
-o, --object identifier attribute	Print the support for all the WMEs that comprise the object (the specified <i>identifier</i>). Must be an existing Soar object <i>identifier</i> . Must be an existing <i>attribute</i> of the specified <i>identifier</i> .

9.3.3.2 Description

The **preferences** command prints all the preferences for the given object identifier and attribute. If identifier and attribute are not specified, they default to the current state and the current operator. The Soar syntax attribute carat (^) is optional when specifying the attribute. The optional arguments indicates the level of detail to print about each preference.

This command is useful for examining which candidate operators have been proposed and what relationships, if any, exist among them. If a preference has o-support, the string, :0 will also be printed.

When only the identifier is specified on the command line, if the identifier is a state, Soar uses the default attribute ^operator. If the identifier is not a state, Soar prints the support information for all WMEs whose value is the identifier.

When an identifier and the --object flag are specified, Soar prints the preferences / WME support for all WMEs comprising the specified identifier.

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

By default, using the --wmes option with a WME on the top state will only print the timetags. To change this, the kernel can be recompiled with DO_TOP_LEVEL_REF_CTS, but this has other consequences (see comments in **kernel.h**).

9.3.3.3 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:
  02 (fill) + :I
    From water-jug*propose*fill
```

```
03 (fill) + :I
  From water-jug*propose*fill
```

unary indifferents:

```
02 (fill) = :I
  From water-jug*propose*fill
```

```
03 (fill) = :I
  From water-jug*propose*fill
```

selection probabilities:

```
03 (fill) + =0. :I (50.0%)
  From water-jug*propose*fill
```

```
02 (fill) + =0. :I (50.0%)
  From water-jug*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:
  (S1 ^jug I4)  :0
  (S1 ^jug J1)  :0
```

This example shows the support for the WMEs with value J1, and the productions that generated them:

```
soar> pref J1 -1
```

```
Support for (33: 03 ^fill-jug J1)
(03 ^fill-jug J1)  =0. :I (100.0%)
  From water-jug*propose*fill
```

```
Support for (22: S1 ^jug J1)
(S1 ^jug J1)  =0. :0 (100.0%)
  From water-jug*apply*initialize-water-jug
```

This example shows the support for all WMEs that make up the object S1:

```
soar> pref -o s1
```

```
Support for S1 ^name:
(S1 ^name water-jug)  :0
Support for S1 ^jug:
(S1 ^jug I4)  :0
(S1 ^jug J1)  :0
Support for S1 ^svs:
Preferences for S1 ^operator:
acceptables:
  02 (fill) + :I
  03 (fill) + :I

unary indifferents:
  02 (fill) = :I
  03 (fill) = :I
Support for S1 ^smem:
Support for S1 ^epmem:
Support for S1 ^reward-link:
Arch-created wmes for S1 :
(2: S1 ^superstate nil)
(1: S1 ^type state)
Input (IO) wmes for S1 :
(15: S1 ^io I1)
```

Default Aliases

- pref

9.3.3.4 See Also

[decide](#)

9.3.4 svs

Control the behavior of the Spatial Visual System

Synopsis

```
svs <path> dir
svs <path> help
svs connect_viewer <port>
svs disconnect_viewer
svs filters
svs filters.<filter_name>
svs commands
svs commands.<command_name>
svs <state>.scene.world
svs <state>.scene.world.<path-to-node>
svs <state>.scene.properties
svs <state>.scene.sgel <sgel-command>
svs <state>.scene.draw on|off
svs <state>.scene.clear
```

9.3.4.1 Paths

SVS can be navigated by specifying a path after the svs command. This path mimicks a directory structure and is specified by dot notation.

Path	Argument	Description
connect_viewer	<port>	Connects to a svs_viewer listening on the given port
disconnect_viewer		Disconnects from an active svs_viewer
filters		Prints out a list of all the filters
filters.<filter_name>		Prints information about a specific filter
commands		Prints out a list of all the soar commands
commands.<command_name>		Prints information about a specific command
<state>.scene.world		Prints information about the world
<state>.scene.<node-path>		Prints information about a specific node
<state>.scene.properties		Prints pos/rot/scale/tag info about all nodes
<state>.scene.sgel	<sgel>	Sends an sgel command to the scene
<state>.scene.draw	on	Causes this scene to be the one drawn on the viewer
<state>.scene.draw	off	Stops this scene from being drawn in the viewer
<state>.scene.clear		Removes all objects from the given scene

9.3.4.2 Description

Each path can be followed by `help` to print some help info, or followed by `dir` to see the children of that path.

The `<state>` variable is the identifier for the substate you want to examine. For example, to do things to the topstate scene you would use `svs S1.scene`.

9.3.4.3 Examples

Print the full SVS directory structure

```
svs . dir
```

Print help information about connect_viewer

```
svs connect_viewer help
```

Print information about a distance filter

```
svs filters.distance
```

Print all the nodes in the scene for substate S17

```
svs S17.scene.world dir
```

Print information about the node wheel2 on car5

```
svs S1.scene.world.car5.wheel2
```

Add a new node to the scene using SGEL

```
svs S1.scene.sgel add ball3 world ball .5 position 1 1 1
```

9.4 Learning

This section describes the commands for enabling and configuring Soar's mechanisms of chunking and reinforcement learning. The specific commands described in this section are:

chunk - Set the parameters for explanation-based chunking, Soar's learning mechanism.

rl - Get/Set RL parameters and statistics.

9.4.1 chunk

Sets the parameters for explanation-based chunking.

Synopsis

Chunk Commands and Settings	
? help	Print this help listing
stats	Print statistics on learning
----- Settings -----	
always NEVER only except	When Soar will learn new rules
bottom-only [on OFF]	Learn only from bottom sub-state
naming-style [numbered RULE]	Numeric names or rule-based names
max-chunks 50	Maximum chunks that can be learned
max-dupes 3	Maximum duplicate chunks (per rule)
----- Debugging -----	
interrupt [on OFF]	Stop after learning from any rule
explain-interrupt [on OFF]	Stop after learning rule watched
warning-interrupt [on OFF]	Stop after detecting learning issue
----- Fine Tune -----	
singleton	Print all WME singletons
singleton <type> <attribute> <type>	Add a WME singleton pattern
singleton -r <type> <attribute> <type>	Remove a WME singleton pattern
----- EBC Mechanisms -----	
add-ltm-links [ON off]	Recreate LTM links in results
add-osk [on OFF]	Incorporate operator selection rules
merge [ON off]	Merge redundant conditions
lhs-repair [ON off]	Addconds for unconnected LHS IDs
rhs-repair [ON off]	Addconds for unconnected RHS IDs
user-singletons [ON off]	Unify with domain singletons
----- Correctness Guarantee Filters -----	
allow-local-negations [ON off]	Allow local negative reasoning
allow-opaque* [ON off]	Allow knowledge from a LTM recall
allow-missing-osk* [ON off]	Allow problem-solving that used OSK
allow-uncertain-operators* [ON off]	Allow operators decided probabilist
allow-conflated-reasoning* [ON off]	Allow problem-solving with multiple
* disabled	
To change a setting:	chunk <setting> [<value>]
For a detailed explanation of these settings:	help chunk

9.4.1.1 Description

The `chunk` command controls the parameters for explanation-based chunking. With no arguments, this command prints out a basic summary of the current learning parameters, how many rules have been learned and which states have learning active. With an `?` argument, it will list all sub-commands, options and their current values.

Turning on Explanation-Based Chunking Chunking is *disabled* by default. Learning can be turned on or off at any point during a run. Also note that Soar uses most aspects of EBC to create justifications as well, so many aspects of the chunking algorithm still occur even when learning is off.

`chunk always:` Soar will always attempt to learn rules from sub-state problem-solving.

```

chunk never:      Soar will never attempt to learn rules.
chunk unflagged: Chunking is on in all states _except_ those that have had RHS
                  'dont-learn' actions executed in them.
chunk flagged:   Chunking is off for all states except those that are flagged
                  via a RHS 'force-learn' actions.

```

The **flagged** argument 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 **unflagged** 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 **bottom-only** setting control when chunks are formed when there are multiple levels of subgoals. With bottom-up learning, chunks are learned only in states in which no subgoal has yet generated a chunk. In this mode, chunks are learned only for the “bottom” of the subgoal hierarchy and not the intermediate levels. With experience, the subgoals at the bottom will be replaced by the chunks, allowing higher level subgoals to be chunked.

9.4.1.2 Debugging Explanation-Based Chunking

The best way to understand why and how rules formed is to use the **explain** command. It will create detailed snapshots of everything that existed when a rule or justification formed that you can interactively explore. See [explain](#) for more information. You can even use it in conjunction with the visualizer to create graphs depicting the dependency between rules in a sub-state.

The **stats** command will print a detailed table containing statistics about all chunking activity during that run.

The **interrupt** setting forces Soar to stop after forming any rule.

The **explain-interrupt** setting forces Soar to stop when it attempts to form a rule from a production that is being watched by the explainer. See [explain](#) for more information.

The **warning interrupts** setting forces Soar to stop when it attempts to form a rule but detects an issue that may be problematic.

The **record-utility** command is a tool to determine how much processing may be saved by a particular learned rule. When enabled, Soar will detect that a chunk matched, but will not fire it. Assuming that the rule is correct, this should lead to an impasse that causes a duplicate chunk to form. The amount of time and decision cycles spent in that impasse are recorded and stored for the rule. Rules are also flagged if a duplicate is not detected or if an impasse is not generated.

This feature is not yet implemented.

9.4.1.3 Preventing Possible Correctness Issues

chunk allow-local-negations The option `allow-local-negations` control whether or not chunks can be created that are derived from rules that check local WMEs in the substate don't exist. Chunking through local negations can result in overgeneral chunks, but disabling this ability will reduce the number of chunks formed. The default is to enable chunking through local negations.

If chunking through local negations is disabled, to see when chunks are discarded (and why), set `watch --learning print` (see `watch` command).

The following commands are not yet enabled. Soar will currently allow all of these situations.

allow-missing-osk Used operator selection rules to choose operator

allow-opaque Used knowledge from opaque knowledge retrieval

allow-uncertain-operators Used operators selected probabilistically

allow-conflated-reasoning Tests a WME that has multiple reasons it exists

9.4.1.4 Other Settings that Control WHEN Rules are Learned

chunk max-chunks 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.

Important note:

If you see this warning, something is seriously wrong; Soar will be 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).

chunk max-dupes The `max-dupes` command is used to limit the maximum number of duplicate chunks that can form from a particular rule in a single decision cycle. The initial value of this variable is 3; allowable settings are any integer greater than 0. Note that this limit is per-rule, per-state. With the default value, each rule can match three times in a sub-state and create two duplicate, reject rules before Soar will stop attempting to create new rules based on that rule. The limit is reset the next decision cycle.

This limit is included in Soar to prevent slowing down when multiple matches of a rule in a substate produce the same general rule. Explanation-based chunking can now produce very general chunks, so this can happen in problem states in which the logic leads to multiple matches, which leads to results being created multiple times in the same decision cycle.

9.4.1.5 Settings that Alter the Mechanisms that EBC Uses

chunk add-osk The option `add-osk` control whether or not operator selection knowledge is backtraced through when creating justifications and chunks. When this option is disabled, only requirement preferences (requires and prohibits) will be added backtraced through. When this option is enabled, relevant desirability prefs (better, best, worse, worst, indifferent) will also be added, producing more specific and possibly correct chunks. This feature is still experimental, so the default is to not include operator selection knowledge.

The following commands are not yet enabled. Soar will always use the EBC mechanisms listed below.

variablize-identity Variablize symbols based on identity analysis

variablize-rhs-funcs Variablize and compose RHS functions

enforce-constraints Track and enforce transitive constraints

repair Repair rules that aren't fully connected

merge Merge redundant conditions

user-singletons Unify identities using domain-specific singletons

9.4.1.6 Chunk Naming Style

The numbered style for naming newly-created chunks is:

`<prefix><chunknum>`

The rule-based (default) style for naming chunks is:

```
<prefix>*<original-rule-name>*<impassetyp>*<dc>-<dcChunknum>
```

where:

- prefix is either chunk or justification, depending on whether learning was on for that state,
- chunknum is a counter starting at 1 for the first chunk created,
- original-rule-name is the name of the production that produced the result that resulted in this chunk,
- dc is the number of the decision cycle in which the chunk was formed,
- impassetyp is one of Tie, Conflict, Failure, StateNoChange, OpNoChange,
- dcChunknum is the number of the chunk within that specific decision cycle.

Note that when using the rule-based naming format, a chunk based on another chunk will have a name that begins with prefix followed by -xN, for example
chunk-x3*apply-rule*42-2.

9.4.1.7 Default Aliases

```
learn      chunk
cs         chunk --stats
```

9.4.1.8 See Also

[explain](#)
[trace](#)
[visualize](#)

9.4.2 rl

Control how numeric indifferent preference values in RL rules are updated via reinforcement learning.

Synopsis

```
rl -g|--get <parameter>
rl -s|--set <parameter> <value>
rl -t|--trace <parameter> <value>
rl -S|--stats <statistic>
```

Options :

Option	Description
-g, --get	Print current parameter setting
-s, --set	Set parameter value
-t, --trace	Print, clear, or init traces
-S, --stats	Print statistic summary or specific statistic

Description The `r1` command sets parameters and displays information related to reinforcement learning. The `print` and `trace` commands display additional RL related information not covered by this command.

9.4.2.1 Parameters

Due to the large number of parameters, the `r1` command uses the `--get|--set <parameter> <value>` convention rather than individual switches for each parameter. Running `r1` without any switches displays a summary of the parameter settings.

Parameter	Description	Possible values	Default
chunk-stop	If enabled, chunking does not create duplicate RL rules that differ only in numeric-indifferent preference value	on, off	on
decay-mode	How the learning rate changes over time	normal, exponential, logarithmic, delta-bar-delta	normal
discount-rate	Temporal discount (gamma)	[0, 1]	0.9
eligibility-trace-decay-rate	Eligibility trace decay factor (lambda)	[0, 1]	0
eligibility-trace-tolerance	Smallest eligibility trace value not considered 0	(0, inf)	0.001
hrl-discount	Discounting of RL updates over time in impassed states	on, off	off
learning	Reinforcement learning enabled	on, off	off

Parameter	Description	Possible values	Default
learning-rate	Learning rate (alpha)	[0, 1]	0.3
step-size-parameter	Secondary learning rate	[0,1]	1
learning-policy	Value update policy	sarsa, q-learning, off-policy-gq-lambda, on-policy-gq-lambda	sarsa
meta	Store rule metadata in header string	on, off	off
temporal-discount	Discount RL updates over gaps	on, off	on
temporal-extension	Propagation of RL updates over gaps	on, off	on
trace	Update the trace	on, off	off
update-log-path	File to log information about RL rule updates	"", <filename>	""

Apoptosis Parameters :

Parameter	Description	Possible values	Default
apoptosis	Automatic excising of productions via base-level decay	none, chunks, rl-chunks	none
apoptosis-decay	Base-level decay parameter	[0, 1]	0.5
apoptosis-thresh	Base-level threshold parameter (negates supplied value)	(0, inf)	2

Apoptosis is a process to automatically excise chunks via the base-level decay model (where rule firings are the activation events). A value of `chunks` has this apply to any chunk, whereas `rl-chunks` means only chunks that are also RL rules can be forgotten.

9.4.2.2 RL Statistics

Soar tracks some RL statistics over the lifetime of the agent. These can be accessed using `rl --stats <statistic>`. Running `rl --stats` without a statistic will list the values of all statistics.

Option	Description
<code>update-error</code>	Difference between target and current values in last RL update
<code>total-reward</code>	Total accumulated reward in the last update
<code>global-reward</code>	Total accumulated reward since agent initialization

Delta-Bar-Delta This is an experimental feature of Soar RL. It based on the work in Richard S. Sutton’s paper “Adapting Bias by Gradient Descent: An Incremental Version of Delta-Bar-Delta”, available online at <http://webdocs.cs.ualberta.ca/~sutton/papers/sutton-92a.pdf>.

Delta Bar Delta (DBD) is implemented in Soar RL as a decay mode. It changes the way all the rules in the eligibility trace get their values updated. In order to implement this, the agent gets an additional learning parameter “meta-learning-rate” and each rule gets two additional decay parameters: beta and h. The meta learning rate is set manually; the per-rule features are handles automatically by the DBD algorithm. The key idea is that the meta parameters keep track of how much a rule’s RL value has been updated recently, and if a rule gets updates in the same direction multiple times in a row then subsequent updates in the same direction will have more effect. So DBD acts sort of like momentum for the learning rate.

To enable DBD, use “`rl –set decay-mode delta-bar-delta`”. To change the meta learning rate, use e.g. “`rl –set meta-learning-rate 0.1`”. When you execute “`rl`”, under the Experimental section you’ll see the current settings for decay-mode and meta-learning-rate. Also, if a rule gets printed concisely (e.g. by executing “`p`”), and the rule is an RL rule, and the decay mode is set to delta-bar-delta, then instead of printing the rule name followed by the update count and the RL value, it will print the rule name, beta, h, update count, and RL value.

Note that DBD is a different feature than “meta”. Meta determines whether metadata about a production is stored in its header string. If meta is on and DBD is on, then each rule’s beta and h values will be stored in the header string in addition to the update count, so you can print out the rule, source it later and that metadata about the rule will still be in place.

GQ(λ) Linear GQ(λ) is a gradient-based off-policy temporal-difference learning algorithm, as developed by Hamid Maei and described by Adam White and Rich Sutton (<https://arxiv.org/pdf/1705.03967.pdf>). This reinforcement learning option provides off-policy learning quite effectively. This is a good approach in cases when agent training performance is less important than agent execution performance. GQ(λ) converges despite irreversible ac-

tions and other difficulties approaching the training goal. Convergence should be guaranteed for stable environments.

“rl –set learning-policy off-policy-gq-lambda” will set Soar to use linear GQ(λ). It is preferable to use GQ(λ) over sarsa or q-learning when multiple weights are active in parallel and sequences of actions required for agents to be successful are sufficiently complex that divergence is possible. To take full advantage of GQ(λ), it is important to set step-size-parameter to a reasonable value for a secondary learning rate, such as 0.01.

“rl –set learning-policy on-policy-gq-lambda” will set Soar to use a simplification of GQ(λ) to make it on-policy while otherwise functioning identically. It is still important to set step-size-parameter to a reasonable value for a secondary learning rate, such as 0.01.

To change the secondary learning rate that only applies when learning with GQ(λ), use “rl –set step-size-parameter [0,1]”. It controls how fast the secondary set of weights changes to allow GQ(λ) to improve the rate of convergence to a stable policy. Small learning rates such as 0.01 or even lower seems to be good practice.

For more information, please see the relevant slides on <http://www-personal.umich.edu/~bazald/b/publications/009-sw35-gql.pdf>

RL Update Logging Sets a path to a file that Soar RL will write to whenever a production’s RL value gets updated. This can be useful for logging these updates without having to capture all of Soar’s output and parse it for these updates. Enable with e.g. “rl –set update-log-path rl_log.txt”. Disable with <rl –set update-log-path “”> - that is, use the empty string “” as the log path. The current log path appears under the experimental section when you execute “rl”.

RL Trace If “rl –set trace on” has been called, then proposed operators will be recorded in the trace for all goal levels. Along with operator names and other attribute-value pairs, transition probabilities derived from their numeric preferences are recorded.

Legal arguments following “rl -t” or “rl –trace” are as follows:

Option	Description
print	Print the trace for the top state.
clear	Erase the traces for all goal levels.
init	Restart recording from the beginning of the traces for all goal levels.

These may be followed by an optional numeric argument specifying a specific goal level to print, clear, or init. “rl -t init” is called automatically whenever Soar is reinitialized. However, “rl -t clear” is never called automatically.

The format in which the trace is printed is designed to be used by the program dot, as part of the Graphviz suite. The command “ctf rl.dot rl -t” will print the trace for the top state

to the file “rl.dot”. (The default behavior for “rl -t” is to print the trace for the top state.)

Here are some sample dot invocations for the top state:

Option	Description
dot -Tps rl.dot -o rl.ps	ps2pdf rl.ps
dot -Tsvg rl.dot -o rl.svg	inkscape -f rl.svg -A rl.pdf

The .svg format works better for large traces.

9.4.2.3 See Also

[excise](#)
[print](#)
[trace](#)

9.5 Long-term Declarative Memory

This section describes the commands for enabling and configuring Soar’s long-term semantic memory and episodic memory systems. The specific commands described in this section are:

smem - Get/Set semantic memory parameters and statistics.

epmem - Get/Set episodic memory parameters and statistics.

9.5.1 smem

Controls the behavior of and displays information about semantic memory.

Synopsis

```
=====
-      Semantic Memory Sub-Commands and Options      -
=====
enabled                      off
database                     memory
append                       on
path
```

```
-----
smem [? | help]
smem [--enable | --disable ]
smem [--get | --set]           <option> [<value>]
smem --add                     { (id ^attr value)* }
smem --backup                  <filename>
smem --clear
smem --export                 <filename> [<LTI>]
smem --init
smem --query                  {(cue)* [<num>]}
smem --remove                 { (id [^attr [value]])* }

----- Printing -----
print                                @
print                               <LTI>
smem --history                         <LTI>

----- Activation -----
activation-mode                      recency
activate-on-query                   on
base-decay                           0.5
base-update-policy                  stable
base-incremental-threshes          10
thresh                             100

----- Experimental Spreading Activation -----
spreading                            off
spreading-limit                     300
spreading-depth-limit              10
spreading-baseline                 0.0001
spreading-continue-probability    0.9
spreading-loop-avoidance          off

----- Database Optimization Settings -----
lazy-commit                          on
optimization                         performance
cache-size                           10000
page-size                            8k

----- Timers and Statistics -----
timers <detail>                    off
smem --timers                       [<timer>]
smem --stats                         [<stat>]

-----
Detail: off, one, two, three
Timers: smem_api, smem_hash, smem_init, smem_query,
        smem_ncb_retrieval, three_activation
        smem_storage, _total
Stats:  act_updates, db-lib-version, edges, mem-usage,
       mem-high, nodes, queries, retrieves, stores
-----
```

For a detailed explanation of these settings: `help smem`

9.5.1.1 Summary Output

With no arguments, `smem` will return a quick summary of key aspects of semantic memory.

```
=====
          Semantic Memory Summary
=====
Enabled                      off
Storage                     Memory (append after init)
-----
Nodes                         2
Edges                         1
Memory Usage                  406784 bytes
-----
```

For a full list of `smem`'s sub-commands and settings: `smem ?`

Options :

Option	Description
<code>-e, --enable, --on</code>	Enable semantic memory.
<code>-d, --disable, --off</code>	Disable semantic memory.
<code>-g, --get</code>	Print current parameter setting
<code>-s, --set</code>	Set parameter value
<code>-c, --clear</code>	Deletes all memories
<code>-i, --init</code>	Deletes all memories if append is off
<code>-S, --stats</code>	Print statistic summary or specific statistic
<code>-t, --timers</code>	Print timer summary or specific statistic
<code>-a, --add</code>	Add concepts to semantic memory
<code>-r, --remove</code>	Remove concepts from semantic memory
<code>-q, --query</code>	Print concepts in semantic store matching some cue
<code>-h, --history</code>	Print activation history for some LTI
<code>-b, --backup</code>	Creates a backup of the semantic database on disk

Printing To print from semantic memory, the standard print command can be used, for example, to print a specific LTI:

`p @23`

To print the entire semantic store:

`p @`

Note that such print commands will honor the `--depth` parameter passed in.

The command `trace --smem` displays additional trace information for semantic memory not controlled by this command.

9.5.1.2 Parameters

Due to the large number of parameters, the `smem` command uses the `--get|--set <parameter> <value>` convention rather than individual switches for each parameter. Running `smem` without any switches displays a summary of the parameter settings.

Parameter	Description	Possible values	Default
append	Controls whether database is overwritten or appended when opening or re-initializing	on, off	off
database	Database storage method	file, memory	memory
learning	Semantic memory enabled	on, off	off
path	Location of database file	<i>empty, some path</i>	<i>empty</i>

The `learning` parameter turns the semantic memory module on or off. This is the same as using the enable and disable commands.

The `path` parameter specifies the file system path the database is stored in. When `path` is set to a valid file system path and database mode is set to file, then the SQLite database is written to that path.

The `append` parameter will determine whether all existing facts stored in a database on disk will be erased when semantic memory loads. Note that this affects semantic memory re-initialization also, i.e. if the append setting is off, all semantic facts stored to disk will be lost when a `soar init` is performed. For semantic memory, `append` mode is by default on.

Note that changes to database, `path` and `append` will not have an effect until the database is used after an initialization. This happens either shortly after launch (on first use) or after a database initialization command is issued. To switch databases or database storage types while running, set your new parameters and then perform an `smem --init` command.

Activation Parameters :

Parameter	Description	Possible values	Default
activation-mode	Sets the ordering bias for retrievals that match more than one memory	recency, frequency, base-level	recency
activate-on-query	Determines if the results of queries should be activated	on, off	on
base-decay	Sets the decay parameter for base-level activation computation	> 0	0.5
base-update-policy	Sets the policy for re-computing base-level activation	stable, naive, incremental	stable
base-incremental-thresholds	Sets time deltas after which base-level activation is re-computed for old memories	1, 2, 3, ...	10
thresh	Threshold for activation locality	0, 1, ...	100
base-inhibition	Sets whether or not base-level activation has a short-term inhibition factor.	on, off	off

If `activation-mode` is `base-level`, three parameters control bias values. The `base-decay` parameter sets the free decay parameter in the base-level model. Note that we do implement the (Petrov, 2006) approximation, with a history size set as a compile-time parameter (default=10). The `base-update-policy` sets the frequency with which activation is recomputed. The default, `stable`, only recomputes activation when a memory is referenced (through storage or retrieval). The `naive` setting will update the entire candidate set of memories (defined as those that match the most constraining cue WME) during a retrieval, which has severe performance detriment and should be used for experimentation or those agents that require high-fidelity retrievals. The `incremental` policy updates a constant number of memories, those with last-access ages defined by the `base-incremental-thresholds` set. The `base-inhibition` parameter switches an additional prohibition factor `on` or `off`.

Performance Parameters :

Parameter	Description	Possible values	Default
cache-size	Number of memory pages used in the SQLite cache	1, 2, ...	10000
lazy-commit	Delay writing semantic store changes to file until agent exits	on, off	on
optimization	Policy for committing data to disk	safety, performance	performance
page-size	Size of each memory page used in the SQLite cache	1k, 2k, 4k, 8k, 16k, 32k, 64k	8k
timers	Timer granularity	off, one, two, three	off

When the database is stored to disk, the `lazy-commit` and `optimization` parameters control how often cached database changes are written to disk. These parameters trade off safety in the case of a program crash with database performance. When `optimization` is set to `performance`, the agent will have an exclusive lock on the database, meaning it cannot be opened concurrently by another SQLite process such as SQLiteMan. The lock can be relinquished by setting the database to `memory` or another database and issuing `init-soar/smem --init` or by shutting down the Soar kernel.

9.5.1.3 Statistics

Semantic memory tracks statistics over the lifetime of the agent. These can be accessed using `smem --stats <statistic>`. Running `smem --stats` without a statistic will list the values of all statistics. Unlike timers, statistics will always be updated.

Available statistics are:

Name	Label	Description
act_updates	Activation Updates	Number of times memory activation has been calculated
db-lib-version	SQLite Version	SQLite library version
edges	Edges	Number of edges in the semantic store
mem-usage	Memory Usage	Current SQLite memory usage in bytes
mem-high	Memory Highwater	High SQLite memory usage watermark in bytes

Name	Label	Description
nodes	Nodes	Number of nodes in the semantic store
queries	Queries	Number of times the query command has been issued
retrieves	Retrieves	Number of times the retrieve command has been issued
stores	Stores	Number of times the store command has been issued

9.5.1.4 Timers

Semantic memory also has a set of internal timers that record the durations of certain operations. Because fine-grained timing can incur runtime costs, semantic memory timers are off by default. Timers of different levels of detail can be turned on by issuing `smem --set timers <level>`, where the levels can be `off`, `one`, `two`, or `three`, `three` being most detailed and resulting in all timers being turned on. Note that none of the semantic memory statistics nor timing information is reported by the `stats` command.

All timer values are reported in seconds.

Level one

Timer	Description
_total	Total smem operations

Level two

Timer	Description
<code>smem_api</code>	Agent command validation
<code>smem_hash</code>	Hashing symbols
<code>smem_init</code>	Semantic store initialization
<code>smem_ncb_retrieval</code>	Adding concepts (and children) to working memory
<code>smem_query</code>	Cue-based queries
<code>smem_storage</code>	Concept storage

Level three

Timer	Description
<code>three_activation</code>	Recency information maintenance

9.5.1.5 smem –add

Concepts can be manually added to the semantic store using the `smem --add <concept>` command. The format for specifying the concept is similar to that of adding WMEs to working memory on the RHS of productions. For example:

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

Although not shown here, the common “dot-notation” format used in writing productions can also be used for this command. Unlike agent storage, manual storage is automatically recursive. Thus, the above example will add a new concept (represented by the temporary “arithmetic” variable) with three children. Each child will be its own concept with two constant attribute/value pairs.

9.5.1.6 smem –remove

Part or all of the information in the semantic store of some LTI can be manually removed from the semantic store using the

```
smem --remove <concept>
```

command. The format for specifying what to remove is similar to that of adding WMEs to working memory on the RHS of productions.

For example:

```
smem --remove {
  (@34 ^good-attribute |gibberish value|)
}
```

If `good-attribute` is multi-valued, then all values will remain in the store except `|gibberish value|`. If `|gibberish value|` is the only value, then `good-attribute` will also be removed. It is not possible to use the common “dot-notation” for this command. Manual removal is not recursive.

Another example highlights the ability to remove all of the values for an attribute:

```
smem --remove {
  (@34 ^bad-attribute)
}
```

When a value is not given, all of the values for the given attribute are removed from the LTI in the semantic store.

Also, it is possible to remove all augmentations of some LTI from the semantic store:

```
smem --remove {
    (@34)
}
```

This would remove all attributes and values of `@34` from the semantic store. The LTI will remain in the store, but will lack augmentations.

(Use the following at your own risk.) Optionally, the user can force removal even in the event of an error:

```
smem -r {(@34 ^bad-attribute ^bad-attribute-2)} force
```

Suppose that LTI `@34` did not contain `bad-attribute`. The above example would remove `bad-attribute-2` even though it would indicate an error (having not found `bad-attribute`).

9.5.1.7 smem –query

Queries for LTIs in the semantic store that match some cue can be initialized external to an agent using the

```
smem --query <cue> [<num>]
```

command. The format for specifying the cue is similar to that of adding a new identifier to working memory in the RHS of a rule:

```
smem --query {
    <cue> ^attribute <wildcard> ^attribute-2 |constant|
}
```

Note that the root of the cue structure must be a variable and should be unused in the rest of the cue structure. This command is for testing and the full range of queries accessible to the agent are not yet available for the command. For example, math queries are not supported.

The additional option of `<num>` will trigger the display of the top `<num>` most activated LTIs that matched the cue.

The result of a manual query is either to print that no LTIs could be found or to print the information associated with LTIs that were found in the `print <lti>` format.

9.5.1.8 smem –history

When the activation-mode of a semantic store is set to base-level, some history of activation events is stored for each LTI. This history of when some LTI was activated can be displayed:

```
smem --history @34
```

In the event that semantic memory is not using base-level activation, `history` will mimic `print`.

9.5.1.9 Experimental Spreading Activation

Parameter	Description	Possible values	Default
spreading	Controls whether spreading activation is on or off.	on, off	off
spreading-limit	Limits amount of spread from any LTI	0, 1, ...	300
spreading-depth-limit	Limits depth of spread from any LTI	0, 1, ..., 10	10
spreading-baseline	Gives minimum to spread values.	0, ..., 1	0.0001
spreading-continue-probability	Gives 1 - (decay factor of spread with distance)	0, ..., 1	0.9
spreading-loop-avoidance	Controls whether spread traversal avoids self-loops	on, off	off

Spreading activation has been added as an additional mechanism for ranking LTIs in response to a query. Spreading activation is only compatible with base-level activation. `activation-mode` must be set to `base-level` in order to also use spreading. They are additive. Spreading activation serves to rank LTIs that are connected to those currently instanced in Working Memory more highly than those which are unconnected. Note that spreading should be turned on before running an agent. Also, be warned that an agent which loads a database with spreading activation active at the time of back-up currently has undefined behavior and will likely crash as spreading activation currently maintains state in the database.

Spreading activation introduces additional parameters. `spreading-limit` is an absolute cap on the number of LTIs that can receive spread from a given instanced LTI. `spreading-depth-limit` is an absolute cap on the depth to which a Working Memory instance of some LTI can spread into the SMem network. `spreading-baseline` provides a minimum amount of spread that an element can receive. `spreading-continue-probability` sets the amount of spread that is passed on with greater depth. (It can also be thought of as 1-decay where decay is the loss of spread magnitude with depth.) `spreading-loop-avoidance` is a boolean parameter which controls whether or not any given spread traversal can loop back onto itself.

Note that the default settings here are not necessarily appropriate for your application. For many applications, simply changing the structure of the network can yield wildly different query results even with the same spreading parameters.

9.5.1.10 See Also

[print](#)
[trace](#)
[visualize](#)

9.5.2 epmem

Control the behavior of episodic memory.

Synopsis

```
epmem
epmem -e|--enable|--on
epmem -d|--disable|--off
epmem -i|--init
epmem -c|--close
epmem -g|--get <parameter>
epmem -s|--set <parameter> <value>
epmem -S|--stats [<statistic>]
epmem -t|--timers [<timer>]
epmem -v|--viz <episode id>
epmem -p|--print <episode id>
epmem -b|--backup <file name>
```

Options :

Option	Description
-e, --enable, --on	Enable episodic memory.
-d, --disable, --off	Disable episodic memory.
-i, --init	Re-initialize episodic memory
-c, --close	Disconnect from episodic memory
-g, --get	Print current parameter setting
-s, --set	Set parameter value
-S, --stats	Print statistic summary or specific statistic
-t, --timers	Print timer summary or specific statistic
-v, --viz	Print episode in graphviz format
-p, --print	Print episode in user-readable format
-b, --backup	Creates a backup of the episodic database on disk

Description The `epmem` command is used to change all behaviors of the episodic memory module, except for watch output, which is controlled by the `trace --epmem` command.

9.5.2.1 Parameters

Due to the large number of parameters, the `epmem` command uses the `--get|--set <parameter> <value>` convention rather than individual switches for each parameter. Running `epmem` without any switches displays a summary of the parameter settings.

Main Parameters :

Parameter	Description	Possible values	Default
append	Controls whether database is overwritten or appended when opening or re-initializing	on, off	off
balance	Linear weight of match cardinality (1) vs. working memory activation (0) used in calculating match score	[0, 1]	1
database	Database storage method	file, memory	memory
exclusions	Toggle the exclusion of an attribute string constant	any string	epmem, smem
force	Forces episode encoding/ignoring in the next storage phase	ignore, remember, off	off
learning	Episodic memory enabled	on, off	off
merge	Controls how retrievals interact with long-term identifiers in working memory	none, add	none

Parameter	Description	Possible values	Default
path	Location of database file	<i>empty, some path</i>	<i>empty</i>
phase	Decision cycle phase to encode new episodes and process epmem link commands	output, selection	output
trigger	How episode encoding is triggered	dc, output, none	output

Performance Parameters :

Parameter	Description	Possible values	Default
cache-size	Number of memory pages used in the SQLite cache	1, 2, ...	10000
graph-match	Graph matching enabled	on, off	on
graph-match-ordering	Ordering of identifiers during graph match	undefined, dfs, mcv	undefined
lazy-commit	Delay writing semantic store changes to file until agent exits	on, off	on
optimization	Policy for committing data to disk	safety, performance	performance
page-size	Size of each memory page used in the SQLite cache	1k, 2k, 4k, 8k, 16k, 32k, 64k	8k
timers	Timer granularity	off, one, two, three	off

The `learning` parameter turns the episodic memory module on or off. When `learning` is set to `off`, no new episodes are encoded and no commands put on the epmem link are processed. This is the same as using the enable and disable commands.

The `phase` parameter determines which decision cycle phase episode encoding and retrieval will be performed.

The `trigger` parameter controls when new episodes will be encoded. When it is set to

output, new episodes will be encoded only if the agent made modifications to the output-link during that decision cycle. When set to ‘dc’, new episodes will be encoded every decision cycle.

The **exclusions** parameter can be used to prevent episodic memory from encoding parts of working memory into new episodes. The value of **exclusions** is a list of string constants. During encoding, episodic memory will walk working memory starting from the top state identifier. If it encounters a WME whose attribute is a member of the **exclusions** list, episodic memory will ignore that WME and abort walking the children of that WME, and they will not be included in the encoded episode. Note that if the children of the excluded WME can be reached from top state via an alternative non-excluded path, they will still be included in the encoded episode. The **exclusions** parameter behaves differently from other parameters in that issuing `epmem --set exclusions <val>` does not set its value to `<val>`. Instead, it will toggle the membership of `<val>` in the **exclusions** list.

The **path** parameter specifies the file system path the database is stored in. When **path** is set to a valid file system path and database mode is set to file, then the SQLite database is written to that path.

The **append** parameter will determine whether all existing episodes recorded in a database on disk will be erased when `epmem` loads it. Note that this affects episodic memory re-initialization also, i.e. if the append setting is off, all episodic memories stored to disk will be lost when an init-soar is performed. Note that episodic memory cannot currently append to an in-memory database. If you perform an init-soar while using an in-memory database, all current episodes stored will be cleared.

Note that changes to database, path and append will not have an effect until the database is used after an initialization. This happens either shortly after launch (on first use) or after a database initialization command is issued. To switch databases or database storage types after running, set your new parameters and then perform an `epmem --init`.

The `epmem --backup` command can be used to make a copy of the current state of the database, whether in memory or on disk. This command will commit all outstanding changes before initiating the copy.

When the database is stored to disk, the **lazy-commit** and **optimization** parameters control how often cached database changes are written to disk. These parameters trade off safety in the case of a program crash with database performance. When **optimization** is set to **performance**, the agent will have an exclusive lock on the database, meaning it cannot be opened concurrently by another SQLite process such as SQLiteMan. The lock can be relinquished by setting the database to memory or another database and issuing init-soar/`epmem --init` or by shutting down the Soar kernel.

The **balance** parameter sets the linear weight of match cardinality vs. cue activation. As a performance optimization, when the value is 1 (default), activation is not computed. If this value is not 1 (even close, such as 0.99), and working memory activation is enabled, this value will be computed for each leaf WME, which may incur a noticeable cost, depending upon the overall complexity of the retrieval.

The `graph-match-ordering` parameter sets the heuristic by which identifiers are ordered during graph match (assuming `graph-match` is on). The default, `undefined`, does not enforce any order and may be sufficient for small cues. For more complex cues, there will be a one-time sorting cost, during each retrieval, if the parameter value is changed. The currently available heuristics are depth-first search (`dfs`) and most-constrained variable (`mcv`). It is advised that you attempt these heuristics to improve performance if the `query_graph_match` timer reveals that graph matching is dominating retrieval time.

The `merge` parameter controls how the augmentations of retrieved long-term identifiers (LTIs) interact with an existing LTI in working memory. If the LTI is not in working memory or has no augmentations in working memory, this parameter has no effect. If the augmentation is in working memory and has augmentations, by default (`none`), episodic memory will not augment the LTI. If the parameter is set to `add` then any augmentations that augmented the LTI in a retrieved episode are added to working memory.

9.5.2.2 Statistics

Episodic memory tracks statistics over the lifetime of the agent. These can be accessed using `epmem --stats <statistic>`. Running `epmem --stats` without a statistic will list the values of all statistics. Unlike timers, statistics will always be updated.

Available statistics are:

Name	Label	Description
<code>time</code>	Time	Current episode ID
<code>db-lib-version</code>	SQLite Version	SQLite library version
<code>mem-usage</code>	Memory Usage	Current SQLite memory usage in bytes
<code>mem-high</code>	Memory Highwater	High SQLite memory usage watermark in bytes
<code>queries</code>	Queries	Number of times the <code>query</code> command has been processed
<code>nexts</code>	Nexts	Number of times the <code>next</code> command has been processed
<code>prevs</code>	Prev	Number of times the <code>previous</code> command has been processed
<code>ncb-wmes</code>	Last Retrieval WMEs	Number of WMEs added to working memory in last reconstruction
<code>qry-pos</code>	Last Query Positive	Number of leaf WMEs in the <code>query</code> cue of last cue-based retrieval
<code>qry-neg</code>	Last Query Negative	Number of leaf WMEs in the <code>neg-query</code> cue of the last cue-based retrieval
<code>qry-ret</code>	Last Query Retrieved	Episode ID of last retrieval
<code>qry-card</code>	Last Query Cardinality	Match cardinality of last cue-based retrieval

Name	Label	Description
qry-lits	Last Query Literals	Number of literals in the DNF graph of last cue-based retrieval

9.5.2.3 Timers

Episodic memory also has a set of internal timers that record the durations of certain operations. Because fine-grained timing can incur runtime costs, episodic memory timers are off by default. Timers of different levels of detail can be turned on by issuing `epmem --set timers <level>`, where the levels can be `off`, `one`, `two`, or `three`, `three` being most detailed and resulting in all timers being turned on. Note that none of the episodic memory statistics nor timing information is reported by the `stats` command.

All timer values are reported in seconds.

Level one

Timer	Description
_total	Total epmem operations

Level two

Timer	Description
epmem_api	Agent command validation
epmem_hash	Hashing symbols
epmem_init	Episodic store initialization
epmem_ncb_retrieval	Episode reconstruction
epmem_next	Determining next episode
epmem_prev	Determining previous episode
epmem_query	Cue-based query
epmem_storage	Encoding new episodes
epmem_trigger	Deciding whether new episodes should be encoded
epmem_wm_phase	Converting preference assertions to working memory changes

Level three

Timer	Description
ncb_edge	Collecting edges during reconstruction
ncb_edge_rit	Collecting edges from relational interval tree
ncb_node	Collecting nodes during reconstruction
ncb_node_rit	Collecting nodes from relational interval tree
query_cleanup	Deleting dynamic data structures
query_dnf	Building the first level of the DNF

Timer	Description
query_graph_match	Graph match
query_result	Putting the episode in working memory
query_sql_edge	SQL query for an edge
query_sql_end_ep	SQL query for the end of the range of an edge
query_sql_end_now	SQL query for the end of the now of an edge
query_sql_end_point	SQL query for the end of the point of an edge
query_sql_start_ep	SQL query for the start of the range of an edge
query_sql_start_now	SQL query for the start of the now of an edge
query_sql_start_point	SQL query for the start of the point of an edge
query_walk	Walking the intervals
query_walk_edge	Expanding edges while walking the intervals
query_walk_interval	Updating satisfaction while walking the intervals

Visualization When debugging agents using episodic memory it is often useful to inspect the contents of individual episodes. Running `epmem --viz <episode id>` will output the contents of an episode in graphviz format. For more information on this format and visualization tools, see <http://www.graphviz.org>. The `epmem --print` option has the same syntax, but outputs text that is similar to using the `print` command to get the substructure of an identifier in working memory, which is possibly more useful for interactive debugging.

9.5.2.4 See Also

[trace](#)

[wm](#)

9.6 Other Debugging Commands

This section describes the commands used primarily for debugging or to configure the trace output printed by Soar as it runs. Many of these commands provide options that simplify or restrict runtime behavior to enable easier and more localized debugging. Users may specify the content of the runtime trace output, examine the backtracing information that supports generated justifications and chunks, or request details on Soar's performance.

The specific commands described in this section are:

trace - Control the information printed as Soar runs. (*was watch*)

output - Controls sub-commands and settings related to Soar's output.

output enabled - Toggles printing at the lowest level.

output console - Redirects printing to the terminal. Most users will not change this.

output callbacks - Toggles standard Soar agent callback-based printing.

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

output command-to-file - Dump the printed output and results of a command to a file.

output print-depth - Set how many generations of an identifier's children that Soar will print

output warnings - Toggle whether or not warnings are printed.

output verbose - Control detailed information printed as Soar runs.

output echo-commands - Set whether or not commands are echoed to other connected debuggers.

explain - Provides interactive exploration of why a rule was learned.

visualize - Creates graph visualizations of Soar's memory systems or processing.

stats - Print information on Soar's runtime statistics.

debug - Contains commands that provide access to Soar's internals. Most users will not need to access these commands

debug allocate - Allocate additional 32 kilobyte blocks of memory for a specified memory pool without running Soar.

debug port - Returns the port the kernel instance is listening on.

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

debug internal-symbols - Print information about the Soar symbol table.

Of these commands, **trace** is the most often used (and the most complex). **output print-depth** is related to the **print** command. **stats** is useful for understanding how much work Soar is doing.

9.6.1 trace

Control the run-time tracing of Soar.

Synopsis

```
=====
Soar Trace Messages
=====
----- Level 1 -----
```

Operator decisions and states	on	-d
<hr/> ----- Level 2 -----		
Phases	off	-p
State removals caused by GDS violation	off	-g
<hr/> ----- Level 3: Rule firings -----		
Default rules	off	-D
User rules	off	-u
Chunks	off	-c
Justifications	off	-j
Templates	off	-T
Firings inhibited by higher-level firings	off	-W
<hr/> ----- Level 4 -----		
WME additions and removals	off	-w
<hr/> ----- Level 5 -----		
Preferences	off	-r
<hr/> ----- Additional Trace Messages -----		
Chunking dependency analysis	off	-b
Goal dependency set changes	off	-G
Episodic memory recording and queries	off	-e
Numeric preference calculations	off	-i
Learning Level	off	-L 0-2
Reinforcement learning value updates	off	-R
Semantic memory additions	off	-s
Working memory activation and forgetting	off	-a
WME Detail Level	none	-n, -t, -f

9.6.1.1 Trace Levels

trace 0-5

Use of the --level (-l) flag is optional but recommended.

Option	Description
0	trace nothing; equivalent to -N
1	trace decisions; equivalent to -d
2	trace phases, gds, and decisions; equivalent to -dp
3	trace productions, phases, and decisions; equivalent to -dpP
4	trace wmes, productions, phases, and decisions; equivalent to -dpwP
5	trace preferences, wmes, productions, phases, and decisions; equivalent to -dpwPw

It is important to note that trace level 0 turns off ALL trace options, including backtracking, indifferent selection and learning. However, the other trace 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 trace level is set to 0.

9.6.1.2 Options

`trace [options]`

Option Flag	Argument to Option	Description
<code>-l, --level</code>	0 to 5 (see <i>Trace Levels</i> below)	This flag is optional but recommended. Set a specific trace level using an integer 0 to 5, this is an inclusive operation
<code>-N, --none</code>	No argument	Turns off all printing about Soar's internals, equivalent to <code>--level 0</code>
<code>-b, --backtracing</code>	<code>remove</code> (optional)	Print backtracing information when a chunk or justification is created
<code>-d, --decisions</code>	<code>remove</code> (optional)	Controls whether state and operator decisions are printed as they are made
<code>-e, --epmem</code>	<code>remove</code> (optional)	Print episodic retrieval traces and IDs of newly encoded episodes
<code>-g, --gds</code>	<code>remove</code> (optional)	Controls printing of warnings when a state is removed due to the GDS
<code>-G, --gds-wmes</code>	<code>remove</code> (optional)	Controls printing of warnings about wme changes to GDS
<code>-i, --indifferent-selection</code>	<code>remove</code> (optional)	Print scores for tied operators in random indifferent selection mode

Option Flag	Argument to Option	Description
<code>-p, --phases</code>	<code>remove</code> (optional)	Controls whether decisions cycle phase names are printed as Soar executes
<code>-r, --preferences</code>	<code>remove</code> (optional)	Controls whether the preferences generated by the traced productions are printed when those productions fire or retract
<code>-P, --productions</code>	<code>remove</code> (optional)	Controls whether the names of productions are printed as they fire and retract, equivalent to <code>-Dujc</code>
<code>-R, --rl</code>	<code>remove</code> (optional)	Print RL debugging output
<code>-s, --smem</code>	<code>remove</code> (optional)	Print log of semantic memory storage events
<code>-w, --wmes</code>	<code>remove</code> (optional)	Controls the printing of working memory elements that are added and deleted as productions are fired and retracted.
<code>-a, --wma</code>	<code>remove</code> (optional)	Print log of working memory activation events
<code>-A, --assertions</code>	<code>remove</code> (optional)	Print assertions of rule instantiations and the preferences they generate.

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.

Tracing Productions By default, the names of the productions are printed as each production fires and retracts (at trace levels 3 and higher). However, it may be more helpful to trace only a specific *type* of production. The tracing of firings and retractions of productions can be limited to only certain types by the use of the following flags:

Option Flag	Argument to Option	Description
-D, --default	remove (optional)	Control only default-productions as they fire and retract
-u, --user	remove (optional)	Control only user-productions as they fire and retract
-c, --chunks	remove (optional)	Control only chunks as they fire and retract
-j, --justifications	remove (optional)	Control only justifications as they fire and retract
-T, --template	remote (optional)	Soar-RL template firing trace

Note: The [production watch](#) command is used to trace individual productions specified by name rather than trace a type of productions, such as `--user`.

Additionally, when tracing 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 traced.

Option Flag	Description
-n, --nowmes	When tracing productions, do not print any information about matching wmes
-t, --timetags	When tracing productions, print only the timetags for matching wmes
-f, --fullwmes	When tracing productions, print the full matching wmes

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

Tracing Learning As Soar is running, it may create justifications and chunks which are added to production memory. The trace command allows users to monitor when chunks and justifications are created by specifying one of the following arguments to the `--learning` command:

Argument	Alias	Effect
noprint	0	Print nothing about new chunks or justifications (default)
print	1	Print the names of new chunks and justifications when created
fullprint	2	Print entire chunks and justifications when created

9.6.1.3 Description

The `trace` command controls the amount of information that is printed out as Soar runs. The basic functionality of this command is to trace various *levels* of information about Soar's internal workings. The higher the *level*, the more information is printed as Soar runs. At the lowest setting, 0 (`--none`), nothing is printed. The levels are cumulative, so that each successive level prints the information from the previous level as well as some additional information. The default setting for the *level* is 1, (`--decisions`).

The numerical arguments *inclusively* turn on all levels up to the number specified. To use numerical arguments to turn off a level, specify a number which is less than the level to be turned off. For instance, to turn off tracing of productions, specify `--level 2` (or 1 or 0). Numerical arguments are provided for shorthand convenience. For more detailed control over the trace settings, the named arguments should be used.

With no arguments, this command prints information about the current trace 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`.

9.6.1.4 Examples

The most common uses of `trace` are by using the numeric arguments which indicate trace levels. To turn off all printing of Soar internals, do any one of the following (not all possibilities listed):

```
trace --level 0
trace -l 0
trace -N
```

Note: You can turn off printing at an even lower level using the `output` command.

Although the `--level` flag is optional, its use is recommended:

```
trace --level 5 ## OK
```

```
trace 5          ## OK, avoid
```

Be careful of where the level is on the command line, for example, if you want level 2 and preferences:

```
trace -r -l 2 ## Incorrect: -r flag ignored, level 2 parsed after it and overrides the setting
trace -r 2    ## Syntax error: 0 or remove expected as optional argument to -r
trace -r -l 2 ## Incorrect: -r flag ignored, level 2 parsed after it and overrides the setting
trace 2 -r    ## OK, avoid
trace -l 2 -r ## OK
```

To turn on printing of decisions, phases and productions, do any one of the following (not all possibilities listed):

```
trace --level 3
trace -l 3
trace --decisions --phases --productions
trace -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):

```
trace --level 1 --wmes
trace -l 1 -w
trace --decisions --wmes
trace -d --wmes
trace -w --decisions
trace -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):

```
trace --level 4 --phases remove
trace -l 4 -p remove
trace -l 4 -p 0
trace -d -P -w -p remove
```

To trace the firing and retraction of decisions and *only* user productions, do any one of the following (not all possibilities listed):

```
trace -l 1 -u
trace -d -u
```

To trace 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):

```
trace --decisions --phases --productions --user remove --justifications remove --fullwmes
trace -d -p -P -f -u remove -j 0
trace -f -l 3 -u 0 -j 0
```

9.6.1.5 Default Aliases

```
v      trace -A
```

w	trace
watch	trace

9.6.1.6 See Also

[epmem](#)
[production](#)
[output](#)
[print](#)
[run](#)
[wm](#)

9.6.2 output

Controls settings related to Soar's output

Synopsis

```
=====
-          Output Sub-Commands and Options      -
=====
```

output	[? help]	
agent-trace	<channel-number> [on off]	Controls agent-trace

enabled	on	Global toggle
console	off	For debugging
callbacks	on	Standard printing

output log	[--append -A] <filename>	Log output to file
output log	--add <string>	
output log	[--close]	

output command-to-file	[-a] <file> <cmd> [args]	Log single command

print-depth	1	Default print depth
agent-writes	on	Print RHS output
warnings	on	Print all warnings
echo-commands	off	Echo to debugger

To view/change a setting: output <setting> [<value>]		
For a detailed explanation of these settings:		help output

9.6.2.1 Summary Screen

Using the `output` command without any arguments will display some key output settings:

```
=====
-          Output Status      -
=====
Printing enabled           Yes
Print warnings             Yes
Print verbose output       No
```

To enable specific types of trace messages, use the '`trace`' command.

Use '`output ?`' for a command overview or '`help output`' for the manual page.

9.6.2.2 `output` command-to-file

This command logs a single command. It is almost equivalent to opening a log using [clog](#), running the command, then closing the log, the only difference is that input isn't recorded.

Running this command while a log is open is an error. There is currently not support for multiple logs in the command line interface, and this would be an instance of multiple logs.

This command echoes output both to the screen and to a file, just like `clog`.

Options :

Option	Description
<code>-a</code> , <code>--append</code>	Append if file exists.
<code>filename</code>	The file to log the results of the command to
<code>command</code>	The command to log
<code>args</code>	Arguments for command

9.6.2.3 `output log`

The `output 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 `output log` with no arguments to query the current logging status. Pass a filename to start logging to that file (relative to the command line interface's home directory). Use the `close` option to stop logging.

Usage

```
output log [-A] filename
output log --add string
output log --close
```

Options :

Option	Description
filename	Open filename and begin logging.
-c, --close	Stop logging, close the file.
-a, --add string	Add the given string to the open log file.
-A, --append	Opens existing log file named <code>filename</code> and logging is added at the end of the file.

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

```
output log foo.log
```

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

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

To terminate logging and close the open log file:

```
output log -c
```

Known Issues with log Does not log everything when structured output is selected.

9.6.2.4 General Output Settings

Invoke a sub-command with no arguments to query the current setting. Partial commands are accepted.

Option	Valid Values	Default
echo-commands	yes or no	off
print-depth	≥ 1	1
verbose	yes or no	no
warnings	yes or no	yes

output echo-commands `output echo-commands` will echo typed commands to other connected debuggers. Otherwise, the output is displayed without the initiating command, and this can be confusing.

output print-depth The `print-depth` command reflects the default depth used when working memory elements are printed (using the `print`). The default value is 1. This default depth can be overridden on any particular call to the `print` command by explicitly using the `--depth` flag, e.g. `print --depth 10 args`.

By default, the `print` command prints *objects* in working memory, not just the individual working memory element. To limit the output to individual working memory elements, the `--internal` flag must also be specified in the `print` command. Thus when the print depth is 0, by default Soar prints the entire object, which is the same behavior as when the print depth is 1. But if `--internal` is also specified, then a depth of 0 prints just the individual WME, while a depth of 1 prints all WMEs which share that same identifier. This is true when printing timetags, identifiers or WME patterns.

When the depth is greater than 1, the identifier links from the specified WME's will be followed, so that additional substructure is printed. For example, a depth of 2 means that the object specified by the identifier, wme-pattern, or timetag will be printed, along with all other objects whose identifiers appear as values of the first object. This may result in multiple copies of the same object being printed out. If `--internal` is also specified, then individuals WMEs and their timetags will be printed instead of the full objects.

output verbose The `verbose` command enables tracing of a number of low-level Soar execution details during a run. The details printed by `verbose` are usually only valuable to developers debugging Soar implementation details.

output warnings The `warnings` command enables and disables the printing of warning messages. 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.

9.6.2.5 Default Aliases

<code>ctf</code>	<code>output command-to-file</code>
<code>clog</code>	<code>output log</code>
<code>default-wme-depth</code>	<code>output print-depth</code>
<code>echo-commands</code>	<code>output echo-commands</code>
<code>verbose</code>	<code>output verbose</code>

warnings	output warnings
----------	-----------------

9.6.3 explain

Allows you to explore how rules were learned.

Synopsis

```
===== Explainer Commands and Settings =====
explain ?                                     Print this help listing
----- What to Record -----
all                                         [ on | OFF ]      Record all rules learned
justifications                            [ on | OFF ]      Record justifications
record <chunk-name>                      Record specific rule
list-chunks                                List all rules learned
list-justifications                        List all justifications
----- Starting an Explanation -----
chunk [<chunk name> | <chunk id> ]        Start discussing chunk
formation                                  Describe formation
----- Browsing an Explanation -----
instantiation <inst id>                   Explain instantiation
explanation-trace                         Switch explanation trace
wm-trace                                    Switch to WM trace
----- Supporting Analysis -----
constraints                                 Display extra transitive
                                              constraints required by
                                              problem-solving
identity                                    Display identity to
                                              identity set mappings
stats                                       Display statistics about
                                              currently discussed chunk
----- Settings -----
after-action-report                         [ on | OFF ]      Print statistics to file
                                              on init and exit
only-chunk-identities                      [ ON | off ]      Identity analysis only
                                              prints identities sets
                                              found in chunk
-----
To change a setting:                         explain <setting> [<value>]
For a detailed explanation of these settings: help explain
```

9.6.3.1 Summary Screen

Using the `explain` command without any arguments will display a summary of which rule firings the explainer is watching for learning. It also shows which chunk or justification the user has specified is the current focus of its output, i.e. the chunk being discussed.

Tip: This is a good way to get a chunk id so that you don't have to type or paste in a chunk name.

```
=====
Explainer Summary
=====
Watch all chunk formations           Yes
Explain justifications            No
Number of specific rules watched   0

Chunks available for discussion:
                                         chunkx2*apply2 (c 14)
                                         chunk*apply*o (c 13)
                                         chunkx2*apply2 (c 12)
                                         chunk*apply*d (c 11)
                                         chunkx2*apply2 (c 6)
                                         chunk*apply* (c 15)
                                         chunkx2*apply (c 8)
                                         chunk*apply*c (c 5)
                                         chunkx2*apply (c 10)
                                         chunk*apply (c 1)

* Note: Printed the first 10 chunks. 'explain list' to see other 6 chunks.

Current chunk being discussed:      chunk*apply*down-gripper(c 3)

Use 'explain chunk [ <chunk-name> | id ]' to discuss the formation of that chunk.
Use 'explain ?' to learn more about explain's sub-command and settings.
```

9.6.3.2 explain chunk

This starts the process.

Tip: Use `c`, which is an alias to `explain chunk`, to quickly start discussing a chunk, for example:

```
soar % c 3
Now explaining chunk*apply*move-gripper-above*pass*top-state*OpNoChange*t6-1.
- Note that future explain commands are now relative
  to the problem-solving that led to that chunk.

Explanation Trace                               Using variable identity IDs                                Shortest Path to Result Instantiation
sp {chunk*apply*move-gripper-above*pass*top-state*OpNoChange*t6-1
1:  (<s1> ^top-state <s2>)                      ([140] ^top-state [162])
  -{
2:  (<s1> ^operator <o1>)                      ([140] ^operator [141])
3:  (<o1> ^name evaluate-operator)                ([141] ^name evaluate-operator)
  }
4:  (<s2> ^gripper <g1>)                      ([162] ^gripper [156])
5:  (<g1> ^position up)                         ([156] ^position up)
6:  (<g1> ^holding nothing)                     ([156] ^holding nothing)
7:  (<g1> ^above <t1>)                          ([156] ^above [157])
8:  (<s2> ^io <i2>)                            ([162] ^io [163])
9:  (<i2> ^output-link <i1>)                  ([163] ^output-link [164])
10:  (<i1> ^gripper <g2>)                     ([164] ^gripper [165])
```

```

11:  (<s2> ^clear { <> <t1> <b1> })
      ([162] ^clear { <> [161] [161] })
      i 30 -> i 31
12:  (<s1> ^operator <o1>)
      ([140] ^operator [149])
13:  (<o1> ^moving-block <b1>)
      ([149] ^moving-block [161])
14:  (<o1> ^name pick-up)
      ([149] ^name pick-up)

-->
1:   (<g2> ^command move-gripper-above +)
      ([165] ^command move-gripper-above +)
2:   (<g2> ^destination <c1> +)
      ([165] ^destination [161] +)
}

```

9.6.3.3 explain formation

`explain formation` provides an explanation of the initial rule that fired which created a result. This is what is called the ‘base instantiation’ and is what led to the chunk being learned. Other rules may also be base instantiations if they previously created children of the base instantiation’s results. They also will be listed in the initial formation output.

```

soar % explain formation
-----
The formation of chunk 'chunk*apply*move-gripper-above*pass*top-state*OpNoChange*t6-1' (c 1)
-----

Initial base instantiation (i 31) that fired when apply*move-gripper-above*pass*top-state matched at level 3 at time 6:
Explanation trace of instantiation # 31
(produced chunk result)
-----
```

	(match of rule apply*move-gripper-above*pass*top-state at level 3)	Identities instead of variables	Operational	Creator
1:	([159] ^operator [160])	No	i 30 (pick-up*propose*move-gripper-above)	
2:	([160] ^name move-gripper-above)	No	i 30 (pick-up*propose*move-gripper-above)	
3:	([160] ^destination <des>)	No	i 30 (pick-up*propose*move-gripper-above)	
4:	([159] ^top-state <t1>)	No	i 27 (elaborate*state*top-state)	
5:	([162] ^io <i1>)	Yes	Higher-level Problem Space	
6:	([163] ^output-link <o1>)	Yes	Higher-level Problem Space	
7:	([164] ^gripper <gripper>)	Yes	Higher-level Problem Space	

```

-->
1:   (<gripper> ^command move-gripper-above +)
2:   (<gripper> ^destination <des> +)
-----
```

This chunk summarizes the problem-solving involved in the following 5 rule firings:

```

i 27 (elaborate*state*top-state)
i 28 (elaborate*state*operator*name)
i 29 (pick-up*elaborate*desired)
i 30 (pick-up*propose*move-gripper-above)
i 31 (apply*move-gripper-above*pass*top-state)

```

9.6.3.4 explain instantiation

This is probably one of the most common things you will do while using the explainer. You are essentially browsing the instantiation graph one rule at a time.

Tip: Use `i`, which is an alias to `explain instantiation`, to quickly view an instantiation, for example:

```

soar % i 30
Explanation trace of instantiation # 30
- Shortest path to a result: i 30 -> i 31
-----
```

	(match of rule pick-up*propose*move-gripper-above at level 3)	Identities instead of variables	Operational	Creator
1:	([152] ^name pick-up)	No	i 28 (elaborate*state*operator*name)	
2:	([152] ^desired <d1>)	No	i 29 (pick-up*elaborate*desired)	
3:	([152] ^moving-block <mblock>)	No	i 29 (pick-up*elaborate*desired)	
4:	([152] ^top-state <ts>)	No	i 27 (elaborate*state*top-state)	
5:	([155] ^clear <mblock>)	Yes	Higher-level Problem Space	
6:	([155] ^gripper <g>)	Yes	Higher-level Problem Space	
7:	([156] ^position up)	Yes	Higher-level Problem Space	
8:	([156] ^holding nothing)	Yes	Higher-level Problem Space	

```

9:   (<g> ^above { <> <mblock> <a1> })      ([156] ^above { <> [154] [157] })      Yes      Higher-level Problem Space
-->
1:   (<s> ^operator <op1> +)                  ([152] ^operator [158] +)
2:   (<op1> ^name move-gripper-above +)       ([158] ^name move-gripper-above +)
3:   (<op1> ^destination <mblock> +)           ([158] ^destination [154] +)

```

9.6.3.5 explain explanation-trace and wm-trace

In most cases, users spend most of their time browsing the explanation trace. This is where chunking learns most of the subtle relationships that you are likely to be debugging. But users will also need to examine the working memory trace to see the specific values matched.

To switch between traces, you can use the `explain e` and the `explain w` commands.

Tip: Use `et` and ‘`wt`’, which are aliases to the above two commands, to quickly switch between traces.

```

soar % explain w
Working memory trace of instantiation # 30      (match of rule pick-up*propose*move-gripper-above at level 3)
1:   (S9 ^name pick-up)                         No        i 28 (elaborate*state*operator*name)
2:   (S9 ^desired D6)                           No        i 29 (pick-up*elaborate*desired)
3:   (D6 ^moving-block B3)                      No        i 29 (pick-up*elaborate*desired)
4:   (S9 ^top-state S1)                         No        i 27 (elaborate*state*top-state)
5:   (S1 ^clear B3)                            Yes      Higher-level Problem Space
6:   (S1 ^gripper G2)                          Yes      Higher-level Problem Space
7:   (G2 ^position up)                         Yes      Higher-level Problem Space
8:   (G2 ^holding nothing)                     Yes      Higher-level Problem Space
9:   (G2 ^above { <> B3 T1 } )                Yes      Higher-level Problem Space
-->
1:   (S9 ^operator O9) +
2:   (O9 ^name move-gripper-above) +
3:   (O9 ^destination B3) +

```

9.6.3.6 explain constraints

This feature explains any constraints on the value of variables in the chunk that were required by the problem-solving that occurred in the substate. If these constraints were not met, the problem-solving would not have occurred.

Explanation-based chunking tracks constraints as they apply to identity sets rather than how they apply to specific variables or identifiers. This means that sometimes constraints that appear in a chunk may have been a result of conditions that tested sub-state working memory element. Such conditions don’t result in actual conditions in the chunk, but they can provide constraints. `explain constraints` allows users to see where such constraints came from.

This feature is not yet implemented. You can use `explain stats` to see if any transitive constraints were added to a particular chunk.

9.6.3.7 explain identity

`explain identity` will show the mappings from variable identities to identity sets. If available, the variable in a chunk that an identity set maps to will also be displayed. (Requires

a debug build because of efficiency cost.)

Variable identities are the ID values that are displayed when explaining an individual chunk or instantiation. An identity set is a set of variable identities that were unified to a particular variable mapping. The null identity set indicates identities that should not be generalized, i.e. they retain their matched literal value even if the explanation trace indicates that the original rule had a variable in that element.

By default, only identity sets that appear in the chunk will be displayed in the identity analysis. To see the identity set mappings for other sets, change the `only-chunk-identities` setting to `off`.

```
soar % explain identity
=====
-          Variablization Identity to Identity Set Mappings      -
=====

-== NULL Identity Set ==-
```

The following variable identities map to the null identity set and will not be generalized: 282 301 138 291 355 336 227 309 328 318 128 218 345

-== How variable identities map to identity sets ==-

Variablization IDs	Identity	CVar	Mapping Type
Instantiation 36:			
125 -> 482	IdSet 12 <s>		New identity set
126 -> 493	IdSet 11 <o>		New identity set
Instantiation 38:			
Instantiation 41:			
146 -> 482	IdSet 12 <s>		New identity set
147 -> 493	IdSet 11 <o>		New identity set
Instantiation 42:			
151 -> 180	IdSet 1 <ss>		New identity set
149 -> 482	IdSet 12 <s>		New identity set
150 -> 493	IdSet 11 <o>		New identity set
307 -> 180	IdSet 1 <ss>		Added to identity set
187 -> 180	IdSet 1 <ss>		Added to identity set
334 -> 180	IdSet 1 <ss>		Added to identity set
173 -> 180	IdSet 1 <ss>		Added to identity set
280 -> 180	IdSet 1 <ss>		Added to identity set
Instantiation 53:			
219 -> 489	IdSet 15 		New identity set
Instantiation 61:			
Instantiation 65:			
319 -> 492	IdSet 20 <t>		New identity set

9.6.3.8 explain stats

`explain stats` prints statistics about the chunk being discussed.

```
=====
Statistics for 'chunk*apply*move-gripper-above*pass*top-state*OpNoChange*t6-1' (c 1):
=====

Number of conditions           14
Number of actions               2
Base instantiation             i 31 (apply*move-gripper-above*pass*top-state)
=====

===== Generality and Correctness =====

Tested negation in local substate      No
LHS required repair                   No
RHS required repair                   No
Was unrepairable chunk                No
=====

===== Work Performed =====

Instantiations backtraced through      5
Instantiations skipped                 6
Constraints collected                1
Constraints attached                 0
Duplicates chunks later created       0
Conditions merged                     2
```

9.6.3.9 After-Action Reports

The explainer has an option to create text files that contain statistics about the rules learned by an agent during a particular run. When enabled, the explainer will write out a file with the statistics when either Soar exits or a `soar init` is executed. This option is still considered experimental and in beta.

9.6.3.10 Visualizing an Explanation

Soar's `visualize` command allows you to create images that represent processing that the explainer recorded. There are two types of explainer-related visualizations.

(1) The visualizer can create an image that shows the entire instantiation graph at once and how it contributed to the learned rule. The graph includes arrows that show the dependencies between actions in one rule and conditions in others. This image is one of the most effective ways to understand how a chunk was formed, especially for particularly complex chunks. To use this feature, first choose a chunk for discussion. You can then issue the `visualize` command with the appropriate settings.

(2) The visualizer can also create an image that shows how identities were joined during identity analysis. This can be useful in determining why two elements were assigned the same variable.

9.6.3.11 Default Aliases

```
c    explain chunk  
i    explain instantiation  
  
ef   explain formation  
ei   explain identities  
es   explain stats  
  
et   explain explanation-trace  
wt   explain wm-trace
```

9.6.3.12 See Also

chunk
visualize

9.6.4 visualize

Creates visualizations of Soar's memory systems or processing.

Synopsis

9.6.4.1 Description

The `visualize` command will generate graphical representations of either Soar memory structure or the analysis that explanation-based chunking performed to learn a rule.

This command can be instructed to automatically launch a viewer to see the visual representation. If you have an editor that can open graphviz files, you can have Soar launch that automatically as well. (Such editors allow you to move things around and lay out the components of the visualization exactly as you want them.)

9.6.4.2 Visualizing Memory

```
visualize [wm | smem | epmem] [id] [depth]
```

The first argument is the memory system that you want to visualize.

The optional id argument allows you to specify either a root identifier from which to start working memory or semantic memory visualizations, or an episode ID for episodic memory visualization.

The depth argument specifies how many levels of augmentation that will be printed.

9.6.4.3 Visualizing How a Rule was Learned

```
visualize [ identity_graph | ebc_analysis]
```

`visualize identity_graph` will create a visualization of how the final identities used in a learned rule were determined. This shows all identities involved and how the identity analysis joined them based on the problem-solving that occurred.

`visualize ebc_analysis` will create a visualization of the chunk that was learned and all rules that fired in a substate that contributed to a rule being learned. In addition to all of the dependencies between rules that fired, this visualization also shows which conditions in the instantiations tested knowledge in the superstate and hence contributed to a conditions in the final learned rule.

9.6.4.4 Presentation Settings

`rule-format`: This setting only applies to visualizing EBC processing. The `full` format will print all conditions and actions of the rule. The `name` format will only print a simple object with the rule name.

`memory-format`: This setting only applies to visualizing memory systems. The `node` format will print a single graphical object for every symbol, using a circle for identifiers and a square for constants. The `record` format will print a database-style record for each identifier with all of its augmentations as fields. Links to other identifiers appear as arrows.

`line-style` is a parameter that is passed to Graphviz and affects how lines are drawn between objects. See the Graphviz documentation for legal values.

`separate-states` is a parameter that determines whether a link to a state symbol is drawn. When this setting is on, Soar will not connect states and instead will represent it as a constant. This setting only applies to visualizing memory systems.

`architectural-wmes` is a parameter that determines whether working memory elements created by the architecture, for example I/O and the various memory sub-system links, will be included in the visualization. This setting only applies to visualizing memory systems.

9.6.4.5 File Handling Settings

`file-name` specifies the base file name that Soar will use when creating both graphviz data files and images. You can specify a path as well, for example “`visualization/soar_viz`”, but make sure the directory exists first!

`use-same-file` tells the visualizer to always overwrite the same files for each visualization. When off, Soar will create a new visualization each time by using the base file name and adding a new number to it each time. Note that this command does not yet handle file creation as robustly as it could. If the file already exists, it will simply overwrite it rather than looking for a new file name.

`generate-image` specifies whether the visualizer should render the graphviz file into an image. This setting is overridden if the viewer-launch setting is enabled.

`image-type` specifies what kind of image that visualizer should create. Graphviz is capable of rendering to a staggering number of different image types. The default that the visualizer uses is SVG, which is a vector-based format that can be scaled without loss of clarity. For other legal formats, see the Graphviz or DOT documentation.

9.6.4.6 Post Action Settings

After the data and image files are generated, the visualizer can automatically launch an external program to view or edit the output.

`viewer-launch` specifies whether to launch an image viewer. Most web browser can view SVG files.

`editor-launch` specifies whether to launch whatever program is associated with `.gv` files. For example, on OSX, the program OmniGraffle can be used to great effect.

`print-debug` specifies whether to print the raw Graphviz output to the screen. If you are having problems, you may want to use this setting to see what it is generating for your agent.

Note that your operating system chooses which program to launch based on the file type. This feature has not been tested extensively on other platforms. Certain systems may not allow Soar to launch an external program.

9.6.4.7 See Also

[explain](#)
[epmem](#)
[smem](#)
[chunk](#)

9.6.5 stats

Print information on Soar’s runtime statistics.

Synopsis

`stats [options]`

9.6.5.1 Options

Option	Description
<code>-m, --memory</code>	report usage for Soar’s memory pools
<code>-l, --learning</code>	report statistics about rules learned via explanation-based chunking
<code>-r, --rete</code>	report statistics about the rete structure
<code>-s, --system</code>	report the system (agent) statistics (default)
<code>-M, --max</code>	report the per-cycle maximum statistics (decision cycle time, WM changes, production fires)
<code>-R, --reset</code>	zero out the per-cycle maximum statistics reported by <code>--max</code> command
<code>-t, --track</code>	begin tracking the per-cycle maximum statistics reported by <code>--max</code> for each cycle (instead of only the max value)
<code>-T, --stop-track</code>	stop and clear tracking of the per-cycle maximum statistics
<code>-c, --cycle</code>	print out collected per-cycle maximum statistics saved by <code>--track</code> in human-readable form
<code>-C, --cycle-csv</code>	print out collected per-cycle maximum statistics saved by <code>--track</code> in comma-separated form
<code>-S, --sort N</code>	sort the tracked cycle stats by column number <code>N</code> , see table below

`--sort parameters` :

Option	Description
0	Use default sort
1, -1	Sort by decision cycle (use negative for descending)
2, -2	Sort by DC time (use negative for descending)
3, -3	Sort by WM changes (use negative for descending)
4, -4	Sort by production firings (use negative for descending)

9.6.5.2 Description

This command prints Soar internal statistics. The argument indicates the component of interest, `--system` is used by default.

With the `--system` flag, the `stats` command lists a summary of run statistics, including the following:

- **Version** — The Soar version number, hostname, and date of the run.
- **Number of productions** — The total number of productions loaded in the system, including all chunks built during problem solving and all default productions.
- **Timing Information** — Might be quite detailed depending on the flags set at compile time. See note on timers below.
- **Decision Cycles** — The total number of decision cycles in the run and the average time-per-decision-cycle in milliseconds.
- **Elaboration cycles** — The total number of elaboration cycles that were executed during the run, the average number of elaboration cycles per decision cycle, and the average time-per-elaboration-cycle in milliseconds. This is not the total number of production firings, as productions can fire in parallel.
- **Production Firings** — The total number of productions that were fired.
- **Working Memory Changes** — This is the total number of changes to working memory. This includes all additions and deletions from working memory. Also prints the average match time.
- **Working Memory Size** — This gives the current, mean and maximum number of working memory elements.

The `stats` argument `--memory` provides information about memory usage and Soar's memory pools, which are used to allocate space for the various data structures used in Soar.

The `stats` argument `--learning` provides information about rules learned through Soar's explanation-based chunking mechanism. This is the same output that `chunk stats` provides. For statistics about a specific rule learned, see the `explain` command.

The `stats` argument `--rete` provides information about node usage in the Rete net, the large data structure used for efficient matching in Soar.

The `--max` argument reports per-cycle maximum statistics for decision cycle time, working memory changes, and production fires. For example, if Soar runs for three cycles and there

were 23 working memory changes in the first cycle, 42 in the second, and 15 in the third, the `--max` argument would report the highest of these values (42) and what decision cycle that it occurred in (2nd). Statistics about the time spent executing the decision cycle and number of productions fired are also collected and reported by `--max` in this manner. `--reset` zeros out these statistics so that new maximums can be recorded for future runs. The numbers are also zeroed out with a call to `init-soar`.

The `--track` argument starts tracking the same stats as the `--max` argument but records all data for each cycle instead of the maximum values. This data can be printed using the `--cycle` or `--cycle-csv` arguments. When printing the data with `--cycle`, it may be sorted using the `--sort` argument and a column integer. Use negative numbers for descending sort. Issue `--stop-track` to reset and clear this data.

A Note on Timers The current implementation of Soar uses a number of timers to provide time-based statistics for use in the stats command calculations. These timers are:

- total CPU time
- total kernel time
- phase kernel time (per phase)
- phase callbacks time (per phase)
- input function time
- output function time

Total CPU time is calculated from the time a decision cycle (or number of decision cycles) is initiated until stopped. Kernel time is the time spent in core Soar functions. In this case, kernel time is defined as the all functions other than the execution of callbacks and the input and output functions. The total kernel timer is only stopped for these functions. The phase timers (for the kernel and callbacks) track the execution time for individual phases of the decision cycle (i.e., input phase, preference phase, working memory phase, output phase, and decision phase). Because there is overhead associated with turning these timers on and off, the actual kernel time will always be greater than the derived kernel time (i.e., the sum of all the phase kernel timers). Similarly, the total CPU time will always be greater than the derived total (the sum of the other timers) because the overhead of turning these timers on and off is included in the total CPU time. In general, the times reported by the single timers should always be greater than than the corresponding derived time. Additionally, as execution time increases, the difference between these two values will also increase. For those concerned about the performance cost of the timers, all the run time timing calculations can be compiled out of the code by defining `NO_TIMING_STUFF` (in `kernel.h`) before compilation.

9.6.5.3 Examples

Track per-cycle stats then print them out using default sort:

```
stats --track
run
```

```
stop  
stats --cycle
```

Print out per-cycle stats sorting by decision cycle time

```
stats --cycle --sort 2
```

Print out per-cycle stats sorting by firing counts, descending

```
stats --cycle --sort -4
```

Save per-cycle stats to file `stats.csv`

```
ctf stats.csv stats --cycle-csv
```

Default Aliases

```
st      stats
```

9.6.5.4 See Also

[timers](#)
[init-soar](#)
[command-to-file](#)

9.6.6 debug

Contains commands that provide access to Soar's internals. Most users will not need to access these commands.

Synopsis

```
=====  
          Debug Commands and Settings  
=====  
allocate [pool blocks]           Allocates extra memory to a memory pool  
internal-symbols                 Prints symbol table  
port                                Prints listening port  
time <command> [args]            Executes command and prints time spent
```

debug allocate

```
debug allocate [pool blocks]
```

This `allocate` command allocates additional blocks of memory for a specified memory pool. Each block is 32 kilobyte.

Soar allocates blocks of memory for its memory pools as it is needed during a run (or during other actions like loading productions). Unfortunately, this behavior translates to an increased run time for the first run of a memory-intensive agent. To mitigate this, blocks can be allocated before a run by using this command.

Issuing the command with no parameters lists current pool usage, exactly like `stats` command's memory flag.

Issuing the command with part of a pool's name and a positive integer will allocate that many additional blocks for the specified pool. Only the first few letters of the pool's name are necessary. If more than one pool starts with the given letters, which pool will be chosen is unspecified.

Memory pool block size in this context is approximately 32 kilobytes, the exact size determined during agent initialization.

9.6.6.1 debug internal-symbols

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.

9.6.6.2 debug port

The `port` command prints the port the kernel instance is listening on.

9.6.6.3 debug time

```
debug time command [arguments]
```

The `time` command uses a system clock timer to record the time spent while executing a command. The most common use for this is to time how long an agent takes to run.

9.6.6.4 See Also

[stats](#)

9.7 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:

cd - Change directory.

dirs - List the directory stack.

load - Loads soar files, rete networks, saved percept streams and external libraries.

load file - Sources a file containing soar commands and productions. May also contain Tcl code if Tcl mode is enabled.

load library - Loads an external library that extends functionality of Soar.

load rete-network - Loads a rete network that represents rules loaded in production memory.

load library - Loads soar files, rete networks, saved percept streams and external libraries.

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.

save - Saves chunks, rete networks and percept streams.

save agent - Saves the agent's procedural and semantic memories and settings to a single file.

save chunks - Saves chunks into a file.

save percepts - Saves future input link structures into a file.

save rete-network - Saves the current rete networks that represents rules loaded in production memory.

echo - Prints a string to the current output device.

(See also the output command in Section 9.6.2 on page 265.)

The **load file** command, previously known as **source**, 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. Saving and loading percept streams are used mainly when Soar needs to interact with an external environment. Soar applications that include a graphical interface or other simulation environment will often require the use of **echo**. Users might take advantage of these commands when debugging agents, but care should be used in adding and removing WMEs this way as they do not fall under Soar's truth maintenance system.

9.7.1 File System

Soar can handle the following Unix-style file system navigation commands

9.7.1.1 **pwd**

Print the current working directory.

9.7.1.2 **ls**

List the contents of the current working directory.

9.7.1.3 **cd**

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.

9.7.1.4 **dirs**

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.

9.7.1.5 **pushd**

Push the directory on to the stack. Can be relative path name or a fully specified one.

9.7.1.6 **popd**

Pop the current working directory off the stack and change to the next directory on the stack. Can be relative pathname or a fully specified path.

Default Aliases

chdir	cd
dir	ls
topd	pwd

9.7.2 load

Loads soar files, rete networks, saved percept streams and external libraries.

Synopsis

```
=====
-           Load Sub-Commands and Options
=====
load          [? | help]
-----
load file      [--all --disable] <filename>
load file      [--verbose]      ]
-----
load library   <filename> <args...>
-----
load rete-network --load <filename>
-----
load percepts  --open <filename>
load percepts  --close
-----
```

9.7.2.1 load file

Load and evaluate the contents of a file. The **filename** can be a relative path or a fully qualified path. The 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. This is traditionally known as the source command.

Options :

Option	Description
filename	The file of Soar productions and commands to load.
-a, --all	Enable a summary for each file sourced
-d, --disable	Disable all summaries
-v, --verbose	Print excised production names

Summaries After the source completes, the number of productions sourced and excised is summarized:

```
agent> source demos/mac/mac.soar
*****
Total: 18 productions sourced.
```

```
Source finished.
agent> source demos/mac/mac.soar
#####
Total: 18 productions sourced. 18 productions excised.
Source finished.
```

This can be disabled by using the `-d` flag.

Multiple Summaries A separate summary for each file sourced can be enabled using the `-a` flag:

```
agent> source demos/mac/mac.soar -a
_firstload.soar: 0 productions sourced.
all_source.soar: 0 productions sourced.
**
goal-test.soar: 2 productions sourced.
 ***
monitor.soar: 3 productions sourced.
 ****
search-control.soar: 4 productions sourced.
top-state.soar: 0 productions sourced.
elaborations_source.soar: 0 productions sourced.
_readme.soar: 0 productions sourced.
**
initialize-mac.soar: 2 productions sourced.
*****
move-boat.soar: 7 productions sourced.
mac_source.soar: 0 productions sourced.
mac.soar: 0 productions sourced.
Total: 18 productions sourced.
Source finished.
```

Listing Excised Productions

```
agent> source demos/mac/mac.soar -d
*****
Source finished.
agent> source demos/mac/mac.soar -d
#####
Source finished.
```

A list of excised productions is available using the `-v` flag:

```
agent> source demos/mac/mac.soar -v
#####
Total: 18 productions sourced. 18 productions excised.
```

```

Excised productions:
mac*detect*state*success
mac*evaluate*state*failure*more*cannibals
monitor*move-boat
monitor*state*left
...

```

Combining the `-a` and `-v` flags add excised production names to the output for each file.

9.7.2.2 load rete-network

The `load rete-network` command loads a Rete net previously saved. The Rete net is Soar's internal representation of production 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.

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.

Usage :

```
load rete-network -l <filename>
```

9.7.2.3 load percepts

Replays input stored using the capture-input command. The replay file also includes a random number generator seed and seeds the generator with that.

Synopsis

```

load percepts --open filename
load percepts --close

```

Option	Description
<code>filename</code>	Open filename and load input and random seed.
<code>-o, --open</code>	Reads captured input from file in to memory and seeds the random number generator.
<code>-c, --close</code>	Stop replaying input.

Options

9.7.2.4 load library

Load a shared library into the local client (for the purpose of, e.g., providing custom event handling).

Options :

Option	Description
<code>library_name</code>	The root name of the library (without the .dll or .so extension; this is added for you depending on your platform).
<code>arguments</code>	Whatever arguments the library's initialization function is expecting, if any.

Technical Details Sometimes, a user will want to extend an existing environment. For example, the person may want to provide custom RHS functions, or register for print events for the purpose of logging trace information. If modifying the existing environment is cumbersome or impossible, then the user has two options: create a remote client that provides the functionality, or use `load library`. `load library` creates extensions in the local client, making it orders of magnitude faster than a remote client.

To create a loadable library, the library must contain the following function:

```
#ifdef __cplusplus
extern "C" {
#endif

EXPORT char* sml_InitLibrary(Kernel* pKernel, int argc, char** argv) {
    // Your code here
}

#ifndef __cplusplus
} // extern "C"
#endif
```

This function is called when `load library` loads your library. It is responsible for any initialization that you want to take place (e.g. registering custom RHS functions, registering for events, etc).

The `argc` and `argv` arguments are intended to mirror the arguments that a standard SML client would get. Thus, the first argument is the name of the library, and the rest are whatever other arguments are provided. This is to make it easy to use the same codebase

to create a loadable library or a standard remote SML client (e.g. when run as a standard client, just pass the arguments main gets into `sml_InitLibrary`).

The return value of `sml_InitLibrary` is for any error messages you want to return to the load-library call. If no error occurs, return a zero-length string.

An example library is provided in the `Tools/TestExternalLibraryLib` project. This example can also be compiled as a standard remote SML client. The `Tools/TestExternalLibraryExe` project tests loading the `TestExternalLibraryLib` library.

Load Library Examples To load `TestExternalLibraryLib`:

```
load library TestExternalLibraryLib
```

To load a library that takes arguments (say, a logger):

```
load library my-logger -filename mylog.log
```

9.7.2.5 Default aliases

source	load file
rete-net, rn	load rete-network
replay-input	load input
load-libarary	load library

9.7.2.6 See Also

[file system](#)
[decide](#)
[production](#)
[save](#)

9.7.3 save

Saves chunks, rete networks and percept streams.

Synopsis

```
=====
-           Save Sub-Commands and Options      -
=====
save [? | help]
```

```
-----
save agent <filename>
save chunks <filename>
-----
save percepts --open <filename>
save percepts [--close --flush]
-----
save rete-network --save <filename>
-----
```

For a detailed explanation of sub-commands: help save

9.7.3.1 save agent

The `save agent` command will write all procedural and semantic memory to disk, as well as many commonly used settings. This command creates a standard `.soar` text file, with semantic memory stored as a series of `smem --add` commands.

9.7.3.2 save chunks

The `save chunks` command will write all chunks in memory to disk. This command creates a standard `.soar` text file.

9.7.3.3 save rete-network

The `save rete-network` command saves the current Rete net to a file. The Rete net is Soar's internal representation of production 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.

Note that justifications cannot be present when saving the Rete net. Issuing a [production excise -j](#) before saving a Rete net will remove all justifications.

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.

Usage :

```
save rete-network -s <filename>
```

9.7.3.4 save percepts

Store all incoming input wmes in a file for reloading later. Commands are recorded decision cycle by decision cycle. Use the command [load percepts](#) to replay the sequence.

Note that this command seeds the random number generator and writes the seed to the capture file.

Options :

Option	Description
<code>filename</code>	Open filename and begin recording input.
<code>-o, --open</code>	Writes captured input to file overwriting any existing data.
<code>-f, --flush</code>	Writes input to file as soon as it is encountered instead of storing it in RAM and writing when capturing is turned off.
<code>-c, --close</code>	Stop capturing input and close the file, writing captured input unless the flush option is given.

Usage

```
save percepts -o <filename>
...
save percepts -c
```

9.7.3.5 Default Aliases

<code>capture-input</code>	<code>save percepts</code>
----------------------------	----------------------------

9.7.3.6 See Also

[production](#)
[soar](#)
[load](#)

9.7.4 echo

Print a string to the current output device.

9.7.4.1 Synopsis

```
echo [--newline] [string]
```

9.7.4.2 Options

Option	Description
string	The string to print.
-n, --newline	Supress printing of the newline character

9.7.4.3 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 `--newline` 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.

9.7.4.4 Example

This example will add these comments to the screen and any open log file.

```
echo This is the first run with disks = 12
```

9.7.4.5 See Also

[clog](#)

Index

- !, *see* preference, 22, 69
- !@, 53
- +, *see* preference, 23, 59, 69
- , (comma), 70
- , *see* preference, 23, 56, 69
- <, *see* preference, 53, 69
- <<, >>, 55, 61
- <=, 53
- <=>, 53
- <>, 53
- =, *see* preference, 69
- >, *see* preference, 53, 69
- >=, 53
- @, 53
- @+, 53
- @-, 53
- ^, (carat symbol), 43
- ~, *see* preference, 69
- acceptable preference, *see* preference
- actions, *see* production
- alias (command), 198
- arithmetic operations, 73
- attribute, 6, 8, 14, 43, 44
 - multi-valued, *see* multi-valued attribute tests, 60
- attribute (attribute), 84
- augmentation, *see* working memory element
- best preference, *see* preference
- better preference, *see* preference
- choices (attribute), 84
- chunk, 32, 91, 108
 - overgeneral, 32
- chunk (command), 118, 232
 - add-osk, 235
 - allow-local-negations, 234
 - max-chunks, 234
 - max-dupes, 235
 - naming-style, 235
- chunking, 32, 91
- backtracing, 94, 101, 103, 115, 120
- correctness, 101, 105, 109, 110, 117, 119
- disjunctive context conflation, 113
- ebc-components, 98
- explanation-based chunking, 94, 110
- identity, 95, 99, 104
- inhibition, 108
- learning from instruction, 114
- literalization, 95, 96, 106, 116
- negated conditions, 104, 112
- NULL identity, 105
- over-general, 105, 109, 110, 112, 114, 117
- over-specialization, 110, 111
- relevant operator selection knowledge, 101
- repair, 107
- RHS functions, 81, 117
- singleton, 120
- usage, 118
- comments, 51
- conditions, *see* production
- conflict impasse, *see* impasse
- conjunctive
 - conditions, 55
 - negation, 58
- constant, 44, 83
- constraint-failure impasse, *see* impasse
- debug (command), 281
 - allocate, 281
 - internal-symbols, 282
 - port, 282
 - time, 282
- decide (command), 194
 - indifferent-selection, 138, 195
 - numeric-indifferent-mode, 195
 - predict, 197
 - select, 197
 - set-random-seed, 197
- decision cycle, 7, 24, 25
- decision procedure, 7, 11, 15, 19, 22, 27, 84
- disjunction

of attributes, 61
 of constants, 55
 dot notation, 62, 65
 echo (command), 291
 elaboration, 7, 12
 elaboration cycle, 24, 85
 episodic memory, 155
 epmem, 155
 performance, 161
 retrieve, 157
 storage, 156
 structures, 160
 epmem (command), 251
 exhaustion, 85
 explain (command), 269
 chunk, 270
 constraints, 272
 explanation-trace-and-wm-trace, 272
 formation, 271
 identity, 272
 instantiation, 271
 stats, 274
 floating-point constants, 44
 forgetting, 48
 goal, 5
 examples, 85
 representation, 8, 16
 result, *see* result
 stack, 29
 subgoal, *see* subgoal
 termination, 34, 84
 Goal Dependency Set, 35, 37
 gp (command), 139, 202
 grammar, 82, 83
 help (command), 193
 i-support, 15, 18, 31, 32, 38
 I/O, 11, 15, 25, 86
 input functions, 25, 86
 input links, 87
 io attribute, 87
 output functions, 25, 86
 output links, 87
 identifier, 14, 43, 44, 46
 impasse, 7, 27, 28, 84
 conflict, 28, 34, 84
 constraint-failure, 22, 28, 34, 84
 elimination, 35
 examples, 85
 no-change, 22, 28, 84
 operator no-change, 28, 35
 resolution, 34, 84
 state no-change, 28, 34
 tie, 28, 34, 84
 types, 84
 impasse (attribute), 84
 indifferent preference, *see* preference
 indifferent-selection, 20
 input functions, *see* I/O
 input links, *see* I/O
 instantiation, *see* production, 93, 99
 integer, 44
 interface, 183
 io attribute, *see* I/O
 item (attribute), 85
 item-count (attribute), 85
 justification, 31, 32, 108
 link, 14, 46, 52
 Linux, 4
 load (command), 285
 file, 285
 library, 288
 library-examples, 289
 percepts, 287
 rete-network, 287
 LTI
 comparisons, 53
 definition, 144
 Macintosh, 4
 math-query, 150
 motor commands, *see* I/O
 multi-valued attribute, 14, 45, 58, 63
 negated conditions, 56
 negated conjunctions, 58
 no-change impasse, *see* impasse
 non-numeric (attribute), 85
 non-numeric-count (attribute), 85
 not equal test, 53
 numeric comparisons, 53
 numeric-indifferent preference, *see* preference
 o-support, 18, 31, 32

- object, 6, 14, 44, 46
operator, 5
 application, 11
 comparison, *see* preferences
 proposal, 9
 representation, 8
 selection, 11
 support, 18
operator no-change impasse, *see* impasse
Operator Selection Knowledge (OSK), *see* Context-Dependent Preference Set
output (command), 265
 command-to-file, 266
 echo-commands, 267
 log, 266
 print-depth, 268
 verbose, 268
 warnings, 268
output functions, *see* I/O
output links, *see* I/O

path notation, 62
persistence, 18, 32
predicates, 53
preference, 9, 19, 21, 46, 69
 acceptable as condition, 59
 acceptable(+), 20, 21, 23, 46
 best(>), 20, 23
 better(>*val*), 20, 23
 binary indifferent(= *val*), 20
 numeric-indifferent, 131
 numeric-indifferent(= *num*), 20
 prohibit, 21, 23
 reject(-), 15, 20, 23
 require(!), 21, 22
 syntax, 48
 unary indifferent(=), 20, 24
 worse(< *val*), 20, 23
 worst(<), 20, 24
preference memory, 9, 19
 syntax, 48
preferences (command), 226
print (command), 48, 215
problem solving, 7, 11
problem space, 8, 12
production, 7, 16, 18
 action side (RHS), 17, 18, 67, 83
 coding conventions, 49, 50
 comments, *see* comments
condition side (LHS), 17, 52, 83
conjunctions, *see* conjunctive disjunction
disjunction, *see* disjunction
firing, 16
flags, 50, 71, 140
grammar, 82
instantiation, 17, 32, 38, 45, 52
match, 7
structured value notation, 65
syntax, 48
templates, 140
production (command), 204
 break, 205
 excise, 206
 find, 207
 find-examples, 208
 firing-counts, 208
 matches, 209
 memory-usage, 211
 optimize-attribute, 212
 print-formatting, 216
 printing-options, 215
 watch, 213
production memory, 6, 16, 48
prohibit preference, *see* preference

quiescence, 24, 112, 117
quiescence t (augmentation), 85

reinforcement learning, 131
reject preference, *see* preference
require preference, *see* preference
result, 27, 29, 91, 99, 102
 support, 31
reward-link, 133
RHS Function, 71, 132
 @, 76
 abs, 73
 atan2, 73
 capitalize-symbol, 75
 carriage return, line feed (crlf), 72
 cmd, 81
 compute-heading, 75
 compute-range, 75
 concat, 75
 cos, 73
 dc, 76
 deep-copy, 76
 div, 73

dont-learn, 81, 117
 exec, 80
 float, 74
 floating-point calculations, 73
 force-learn, 82, 117
 halt, 71
 ifeq, 74
 int, 74
 interrupt, 71
 link-stm-to-ltm, 77
 log, 72
 make-constant-symbol, 77
 max, 74
 min, 74
 mod, 73
 rand-float, 78
 rand-int, 78
 round-off, 78
 round-off-heading, 79
 sin, 73
 size, 79
 sqrt, 73
 strlen, 79
 timestamp, 79
 trim, 80
 wait, 71
 write, 72
 RHS of production, *see* production
 RL, 131
 chunking, 141
 eligibility trace settings, 138
 learning-policy, 134
 operator, 131
 reward-link, 133
 rule, 131
 rule generation, 139
 substates, 137
 temporal gaps, 136
 rl (command), 236
 discount-rate, 135
 eligibility-trace-decay-rate, 138
 eligibility-trace-tolerance, 138
 hrl-discount, 137
 learning-policy, 134, 139
 learning-rate, 135
 statistics, 239
 temporal-extension, 136
 trace, 240
 update-logging, 240
 run (command), 191
 save (command), 289
 agent, 290
 chunks, 290
 percepts, 291
 rete-network, 290
 scene graph, 165, 166
 Scene Graph Edit Language, 168
 semantic memory, 143
 singleton, 100, 120
 smem, 143
 activation, 150
 neg-query, 150
 performance, 153
 prohibit, 149
 query, 148
 retrieve, 147
 storage, 147
 store, 145
 store-new, 146
 smem (command), 241
 add, 146, 248
 history, 249
 query, 249
 remove, 248
 SML, 72, 80, 86, 168
 soar (command), 185
 init, 147, 186
 keep-all-top-oprefs, 187
 max-dc-time, 188
 max-elaborations, 188
 max-goal-depth, 188
 max-gp, 188
 max-memory-usage, 189
 max-nil-output-cycles, 189
 stop, 187
 stop-phase, 189
 tcl, 189
 timers, 189
 version, 187
 wait-snc, 190
 sp (command), 49, 200
 Spatial Visual System, 15, 165
 filters, 173
 stack, *see* goal
 state, *see* goal
 state no-change impasse, *see* impasse
 state representation, 8, 16, 84

stats (command), 278
structured value notation, 65
subgoal, *see* goal, 28, 29, 32, 84, 91, 99
 augmentations, 84
 termination, 84
substate, *see* subgoal
superstate, *see* goal, 29
superstate (attribute), 84
support, 32
SVS, *see* Spatial Visual System
svs (command), 229
symbol, 44
symbolic constant, 44
syntax
 preferences, *see* preference
 productions, *see* production
 WMEs, *see* working memory element

templates, 140
tie impasse, *see* impasse
timetag, 45
top-state
 for I/O, 89
trace (command), 258
 levels, 259
type (attribute), 84
type comparisons, 53

Unix, 4

value, 14, 43, 44
variable, 83
variables, 17, 53, 67, 83
visualize (command), 275

Windows, 4

wm (command), 219
 activation, 220
 add, 223
 remove, 224
 watch, 225

WME, *see* working memory element

working memory, 6, 14
 acceptable preference, 46
 object, *see* object
 syntax, 43

working memory activation, 48, 152, 158

working memory element, 14
 syntax, 43

timetag, *see* timetag

worse preference, *see* preference
worst preference, *see* preference

Summary of Soar Aliases and Functions

Predefined Aliases

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

Alias	Command	Page
?	help	193
a	alias	198
add-wme	wm add	223
allocate	debug allocate	281
aw	wm add	223
c	explain chunk	270
capture-input	save percepts	291
chdir	cd	284
chunk-name-format	chunk naming-style	232
cli	soar tcl	189
clog	output log	266
command-to-file	output command-to-file	266
cs	chunk stats	233
cts	output command-to-file	266
d	run -d 1	191
dir	ls	284
e	run -e 1	191
echo-commands	output echo-commands	267
ef	explain formation	271
ei	explain identities	272
es	explain stats	274
et	explain explanation-trace	272
excise	production excise	206
fc	production firing-counts	208
firing-counts	production firing-counts	208
gds_print	print --gds	215
gp-max	soar max-gp	188
h	help	193
i	explain instantiation	271
indifferent-selection	decide indifferent-selection	195
inds	decide indifferent selection	195
init	soar init	186
internal-symbols	debug internal-symbols	282
interrupt	soar stop	187
is	soar init	186
learn	chunk	232
load-library	load library	288
man	help	193
matches	production matches	209
max-chunks	chunk max-chunks	234
max-dc-time	soar max-dc-time	188
max-elaborations	soar max-elaborations	188
max-goal-depth	soar max-goal-depth	188
max-memory-usage	soar max-memory-usage	189
max-nil-output-cycles	soar max-nil-output-cycles	189
memories	production memory-usage	211
multi-attributes	production optimize-attribute	212

numeric-indifferent-mode	decide numeric-indifferent-mode	195
p	print	215
pbreak	production break	205
pc	print --chunks	215
port	debug port	282
predict	decide predict	197
production-find	production find	207
ps	print --stack	215
pw	production watch	213
pwatch	production watch	213
quit	exit	193
r	run	191
remove-wme	wm remove	224
replay-input	load percepts	287
rete-net	load rete-network	287
rn	load rete-network	287
rw	wm remove	224
s	run 1	191
select	decide select	197
set-default-depth	output print-depth	268
set-stop-phase	soar stop-phase	189
soarnews	soar	185
source	load file	285
srand	decide srand	197
ss	soar stop	187
st	stats	278
step	run -d 1	191
stop	soar stop	187
stop-soar	soar-stop	187
tcl	soar tcl	189
time	debug time	282
timers	soar timers	189
topd	pwd	284
un	alias -r	198
unalias	alias -r	198
varprint	print -v -d 100	215
verbose	trace -A	258
version	soar version	187
w	trace	258
waitsnc	soar wait-snc	190
warnings	output warnings	268
watch	trace	258
watch-wmes	wm watch	225
wma	wm activation	220
wmes	print -depth 0 -internal explain wm-trace	215 272

Summary of Soar Functions

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

Command	Summary	Page
<code>alias</code>	Controls aliases for Soar procedures.	198
<code>chunk</code>	Controls parameters for chunking.	232
<code>debug</code>	Accesses Soars internals.	281
<code>decide</code>	Controls operator-selection settings.	194
<code>echo</code>	Echoes arguments to the output stream.	291
<code>epmem</code>	Controls behavior of episodic memory.	251
<code>explain</code>	Explores how rules were learned.	269
<code>gp</code>	Defines a production template.	202
<code>help</code>	Gets information about Soar commands.	193
<code>load</code>	Loads files and libraries.	285
<code>output</code>	Controls Soar output settings.	265
<code>preferences</code>	Examines WME support.	226
<code>print</code>	Prints items in working or production memory.	215
<code>production</code>	Manipulates or analyzes Soar rules.	204
<code>rl</code>	Controls RL preference update settings.	236
<code>run</code>	Begins Soars execution cycle.	191
<code>save</code>	Saves various aspects of Soar memory.	289
<code>smem</code>	Controls behavior of semantic memory.	241
<code>soar</code>	Controls settings for running Soar.	185
<code>sp</code>	Defines a Soar production.	200
<code>stats</code>	Prints information on Soar agent statistics.	278
<code>svs</code>	Controls behavior of the Spatial Visual System.	229
<code>trace</code>	Controls the run-time tracing of Soar.	258
<code>visualize</code>	Creates visualizations of memory or processing.	275
<code>wm</code>	Controls settings related to working memory.	219