The Soar Coloring Book: A Tutorial[1]

Peter M. Hastings
Artificial Intelligence Lab
1101 Beal Avenue
The University of Michigan
Ann Arbor, MI 48109
(313)747-0969

July 1997
Version 1.1[2]

[1]This work was supported by a gift from the McDonnell Foundation.

[2]Version 1.1 includes new and extended examples in comparison to the original Coloring Book (Version 1.0)

Contents

1 Introduction

This document[1] is a practical guide to learning Soar. It assumes no prior knowledge of Soar, but it will be a little easier to follow if you know some basics about Artificial Intelligence. Soar is introduced as an architecture which supports problem solving. Soar programs start with descriptions of three things: the problem, the desired solution, and the actions that can be performed. In the later sections of this document, the learning capabilities and cognitive aspects of Soar are described.

This document is not intended to serve as a reference for Soar. Instead, it is meant as a companion to the manual [Congdon1996]. We will suggest ways to explore Soar and new things to try. We strongly encourage you to make heavy use of the manual to find out different parameters for the various commands, and to find out new things that you can do with them.

The goals of this document are to:

- Introduce the user to the basic operating principals of Soar
- Show the user examples of specific Soar programs
- Teach the user how to run these examples and understand what they do
- Teach the user how to make their own Soar programs.

The document takes the form of a series of lessons which introduce concepts and give you a chance to try them out. Each section builds on the previous ones, so it is best to read them in order. In order to make the best use of this tutorial, we strongly suggest you think of a specific task (different from our examples) and after each lesson, analyze how you would address the current topics in your task. Your task could be anything from a model of human reading performance, to a system that did robot navigation. Trying to apply the concepts taught in this tutorial to a different domain will make them much more clear for you.

What is Soar? Soar can be viewed in three ways:

- It is a problem-solving system.
- It is a learning system.
- It is a candidate Unified Theory of Cognition.

[1]The Soar Coloring Book would not have been possible without the help of many people. John Laird taught me Soar and read many drafts to make sure that the information is technically correct. Randy Jones answered many knocks on his door and stupid questions. Tony Kalus, Karl Schwamb, and Frank Ritter gave me many good comments on early drafts. Doug Pearson gave me lots of hints on how to avoid the pitfalls of chunking. And Clare Congdon graciously tolerated my millions of pestering questions about Soar commands.

This document starts by focusing on the the first viewpoint. After the problem solving aspects of Soar are described, we will introduce the practical aspects of learning and cognitive modeling in Soar. For further information on these and other aspects of Soar, please see these helpful references [Congdon1996, Laird *et al.*1987, Newell1990].

Important note The coloring book assumes that you have version 7 of Soar. You can obtain the current version of Soar via Soar's World Wide Web page:

`http://www.isi.edu/soar/soar.html`

Important files The main file you need to be aware of is the file that contains the executable for Soar. If you already have Soar installed at your site, you can ask the system administrator where the Soar files are. If you don't have it, you will have to install it. See the Soar web page (mentioned above) for information on how to install the most recent Soar release.

When you do run Soar, you will need to set up several environment variables to let Soar know where some of the related files are. The critical variables are `TCL_DIR`, `TCL_LIBRARY`, `TK_LIBRARY`, and `SOAR_LIBRARY`. In the Unix shell, these variables are specified with all uppercase letters, as shown here. Within Soar, these variables use all lowercase letters, for example, `soar_library`. This last variable is especially important because all of the examples files for this tutorial are stored in a subdirectory of the `soar_library` directory called `tutorial`. For details on specifying these variables, please see the installation instructions.

As mentioned above, the `tutorial` subdirectory of the `soar_library` contains the examples which are used in this document. There is a file called README in this directory which describes these files. The README file is also included as an appendix at the end of this tutorial for your convenience.

Another important set of files can be found in the `demos` subdirectory of the `soar_library` directory. These files demonstrate various capabilities of Soar. Although they are not described in the Coloring Book, these programs can be a very useful resource for you in exploring Soar.

2 The Soar approach to problem solving

This section provides a high-level view of how Soar approaches problem solving. We will start with a description of the goals of the section — i.e. what the reader should know after reading it. Then we'll describe the current topic; in this case, the topic is a broad view of problem solving with Soar. Next, we'll tell you how to actually run Soar and see it in action. At the end of each section will be a set of questions or exercises to let you see how well you've understood the material. We will follow this structure throughout the *Coloring Book*.

The goals of this section are to:

- introduce the concepts of states and operators
- show how Soar uses them to solve problems
- start actually running Soar

2.1 States and Operators

To solve a problem, Soar uses knowledge about the initial state of the world (the current situation) and knowledge about the desired state or states which are to be achieved. Soar also uses knowledge about the operators that can be used in the world. The operators transform states. The goal of the Soar program is to select and apply operators which transform the initial state into one of the desired or "goal" states.

For example, a Soar program for playing chess would have an initial state that described the board and the positions of the pieces. Its operators would supply knowledge about how to transform a state by moving a piece. Knowledge about the desired states would describe what it means to win the game: checkmate.

A Soar program for traveling through a maze would have a state that described the maze (or maybe just the area nearby), and your initial location. The operators would describe the possible moves (avoiding bumping into walls, of course), and would transform the state by updating your position. Desired state information would allow the recognition that you have gotten to the goal position

A program for modeling human reading performance would include information about letters and words on the state. Depending on the depth of problem solving, it might also include segments of letters. The operators would resolve any ambiguities in letters, and combine them into words. The goal state could be a complete representation of a sentence.

For our initial examples in the *Coloring Book*, we will use a simple blocks world. This world consists of 3 named blocks sitting on a table and a gripper that can move the blocks. The state information provided by the Soar program will describe what objects are in the world and their relative positions to each other. The desired state will be specified as an arrangement of the blocks. There will be just one operator: moving a block. We choose this example because although it is relatively simple and easy to picture in your head, it provides us with a rich enough environment to show the basic aspects of writing Soar programs.

Problem Definition:

> There is a world with three blocks sitting on a table, as shown on the left side of Figure 1. The goal is produce the configuration shown on the right.

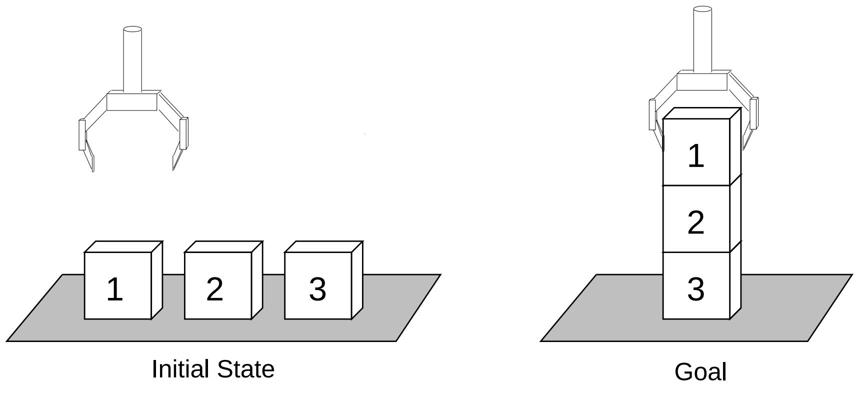


Figure 1: The initial state and goal of the example “blocks-world” task.

A *state* in Soar consists of a set of statements which describe the objects in the world — in this case: 3 blocks and a table. The state contains information about these objects: each of the blocks is clear, and there is a clear space on the table. The state can also contain information about relationships between objects, such as 1, 2, and 3 are all on the table initially, and they are all clear. Other information, such as the color of each block, may or may not be represented in the Soar state.

Soar's operators modify states. An operator in the blocks world moves a block. It specifies that to move block 1 onto block 2 makes the following transformations to the original state:

- Remove the statement that 1 is on the table
- Remove the statement that 2 is clear
- Add the statement that 1 is on 2

Soar also has knowledge about when to propose operators and when they're completed. An operator to move 1 on top of 2 is only proposed when 1 and 2 are both clear. It is completed when 1 is on 2. This separation of knowledge about action (applying operators) and knowledge about control (selecting operators) is a key concept for Soar that we will return to in section 3. For all tasks, problem solving in Soar reduces to selecting and applying operators. The challenge in using Soar is decomposing a task into its component operators and states.

As mentioned above, Soar attempts to transform the initial state into a desired or goal state. We will describe these steps in the next section, following a more hands-on look at this example.

2.2 Soar programs and the Soar architecture

It is very important to understand the distinction between the Soar architecture and Soar programs. The Soar architecture provides a framework for problem solving. It supports the creation and updating of states. It tries to select an operator when one is not currently selected. It does not know anything about particular problems however. The knowledge about the states and operators of a problem comes from a Soar program. Programs specify how a problem state will be represented, what the operators are, and what the operators do. Programs contain the knowledge about particular problems. Soar provides a framework for problem solving. The next section shows Soar in action, and includes a delineation of what Soar does, and what the program does.

It is also very important to understand the distinction between a Soar program and what we generally think of as a program. The classical notion of program consists of a sequence of instructions, like a recipe, which are to be followed in their specified order. A Soar program consists of individual distinct pieces of knowledge, each of which specifies *when* it applies, and *what* it does. This will be further explained in section 3. For convenience sake and for the want of a better term, we'll continue to call the knowledge provided to Soar a "program," but you should bear in mind that it means something a little different from what you expect.

2.3 Running Soar: Simple Blocks World

In this section, Soar will be described via hands-on experience. You are strongly urged to experiment with Soar directly. This section assumes that you have access to a running version of Soar. We will describe how to start Soar, how to load a program, and run it, and to look at what is happening. We will also describe the important step of halting Soar. You will need to know how to find Soar at your site (e.g., file manager in windows interfaces or change directory command in Unix).

```
unix% soar                                          Start soar

Soar> source $soar_library/tutorial/blocks.soar
                                                    Load example file
Loading blocks.soar
**********
Soar> run

      0:  ==>S: S1                                  S1 is the initial state
Initial state has 1, 2, and 3 on the table.
      1:  O: O5 (move-block 1 to 3)                 Operators being chosen
      2:  O: O8 (move-block 1 to table)
      3:  O: O12 (move-block 2 to 3)
      4:  O: O15 (move-block 1 to 2)
Achieved 1, 2, 3                                    The goal was achieved
System halted.

Soar> exit                                          The command to stop Soar
unix%
```

After starting Soar, at the command prompt, you load the example program. (Remember, it's not like a recipe.) Then the `run` command tells Soar to begin the problem solving process.

The text output from Soar as shown above is called a "trace." In other words, we would call this example a "trace" of the blocks-world task. The execution of a Soar program consists of a number of *decision cycles* (described in detail in section 9) each of which results in the selection of an operator; in the solution to the blocks-world task shown above, Soar completes four decision cycles (the trace from each decision cycle is preceded by a number):

1. The zeroth decision cycle is not really a decision cycle at all. In this step, Soar initializes itself.
2. In the first decision cycle, Soar (randomly) selected operator `O5` to move block `1` on top of block `3`.
3. In the second decision cycle, Soar (randomly) selected operator `O8` to move block `1` to the table.
4. In the third decision cycle, Soar (randomly) selected operator `O12` to move block `2` on top of block `3`.
5. In the fourth decision cycle, Soar (randomly) selected operator `O15` to move block `1` on top of block `2`.

After the application of this operator, the Soar program recognizes that this is the tower it was trying to build; it has achieved the goal of building the 1-2-3 tower.

Therefore the program tells Soar to `halt` — to stop doing decision cycles. In the example run above, Soar just happened to build the tower quickly; when you run this program yourself, it is likely that it will take many more decision cycles to randomly build a tower.

At the `Soar>` prompt, enter the `exit` command to stop Soar and return to the operating system.

2.4 Exercises

1. Color the blocks in figure 1. Use whatever colors you wish.
2. Think about your example task. Answer these questions:
   (a) What would be the representation of the state?
   (b) What would the initial and final states be?
   (c) What would the operators be?

3 Operator Knowledge: Proposal, Comparison, Selection, Application, and Termination

In the previous lesson, we describe Soar as pursuing problem solving by using the selection and application of operators. In this section, we break down the steps of selection and application into smaller, more specific parts.

The specific actions which accomplish the higher level concept of *selection* are:

- Operator proposal
- Operator comparison
- Operator selection

The specific actions which accomplish the higher level concept of *application* are:

- Operator application
- Operator termination

Soar separates operator proposal, comparison, selection, application, and termination into different pieces of knowledge. With standard programming languages, the focus is on application – what the program does. Control of the application is accomplished through the use of various control structures and usually the implicit assumption that the next statement in a program should be executed next. By separating these functions, Soar maximizes flexibility. (It also allows it to learn more completely, but we will address learning later.)

Because Soar approaches problem solving differently, it is necessary to think about programs from a different point of view. Instead of being just a series of steps that computes the desired result, Soar programs are individual, independent bits of information – or *knowledge* – about the various aspects of Soar's problem solving process.

The goals of this section are to:

- Describe the key actions associated with operators:
  - Selection actions:
    * proposal
    * comparison
    * selection
  - Application actions:
    * application

    * termination

- introduce the *Working Memory* of Soar which describes the state
- introduce rules which perform actions based on the current state

3.1 Description

In Soar, there are 5 basic program events for operators:

- **Proposal:** An operator is proposed when the conditions are appropriate for it to be selected. Usually, this is when it can be applied to the current situation. For example, a `move-block` operator to move block 1 onto block 2 requires that both blocks are *clear*[2] and that 1 isn't already on top of 2.
- **Comparison:** Any operator whose conditions are met will be proposed. Comparison rules help Soar choose one operator over another. In the simple blocks world, there are no comparison rules, but we will see how comparison knowledge can be used in a later section.
- **Selection:** The point at which Soar chooses a unique operator is called selection. The selection action uses the results of operator proposals and comparisons to determine what proposed operator should be pursued.
- **Application:** Application knowledge tells the selected operator what it should do. For the `move-block` operator, it updates the state to show the new position of the block and to remove the old information about the block's position. For tasks involving interaction with an external environment, this would include initiation of motor commands.
- **Termination:** Separate knowledge informs Soar when an operator is finished. Without this information, Soar won't know when it can select a new operator. For the `move-block` operator, the operator terminates when the block is in its new position.

3.2 Working Memory and Productions

In the next section, we will demonstrate these concepts in action, by running Soar. But before that, we must describe a little bit about how states and operators are implemented in Soar. We introduce below *working memory* and *productions* which will both be described in more detail in section 4. We provide this brief overview here to help you interpret the results of running Soar.

Soar's description of a state is called working memory. Working memory consists of a set of statements each of which describe the world in terms of an object, an attribute

[2]A block is "clear" when no other block is sitting on top of the block.

of that object, and that attribute's value. The term object here can either be Soar's reference to a physical object, or something abstract, for example an `ontop` relation between two objects. Soar uses *identifiers* to refer to objects. Identifiers consist of a letter and an integer, e.g. `S1`. The examples of statements in working memory that you see below collect all of the attributes and values of a single identifier into a list. This list comprises the description of an object. For example, the object below describes the object `S1` with type `state`, and says that it has 4 different `thing` attributes, 2 `clear` attributes, and one `ontop` attribute. The values for all but the first attribute are objects themselves. Object `I1` is a block, and is named "1".[3]

```
(S1 ^type state
    ^thing B1 ^thing B2 ^thing B3 ^thing T1
    ^clear B1 ^clear B2
    ^ontop O1)
(B1 ^type block ^name 1)
```

This representation is not the most powerful possible knowledge representation language. It doesn't represent disjunction, for example. But it is good for describing things, for making models. And that's what we want to use it for.

Operators are represented as a collection of rules. Each rule is called a *production* and consists of an "if" side, specifying the conditions under which the rule should be used, and a "then" or *action* side, specifying what the rule should do.. The conditions of a production rule are constantly compared to the statements in working memory. When all the conditions of a production rule can be satisfied by statements in working memory, the production "fires" – makes changes to the working memory as specified by its action side.

Each of these rules is an independent piece of knowledge in Soar. For example, Soar programs generally use at least one production for each step of proposing, applying, and terminating an operator. There may or may not be productions which help Soar select an operator. In the program trace below, you will see a line which tells when each particular production fires. The names of Soar's productions are generally very descriptive of the role that they play in the overall program. For example, the production `propose*move-block` fires when the `move-block` operator should be proposed in the blocks world.

3.3 Running Soar: Proposal, comparison, selection, application, and termination

In this abbreviated example run of Soar, we will start Soar as before and load the same program. This time, however, we will look more closely at the process of proposing,

[3]In practice, Soar's working memory usually contains much more information than is being shown here. For example, working memory includes statements that describe inputs from external environments and outputs to them.

selecting, applying, and terminating operators. As previously mentioned, this simple blocks-world program has no operator comparison knowledge. Soar is told to randomly choose between proposed operators.

We will run the program one "decision cycle" at a time. A decision cycle is the basic unit of execution for Soar programs and consists of applying a selected operator, terminating it, and proposing one or more new operators.[4] A decision cycle completes when the selected operator is terminated, and a new operator is selected. The decision cycle will be more completely described in lesson 9.

Programming hint : If you want to save the program trace of your run to a file (to examine at a later time), use the `log` command. At the beginning of your run, use the command: `log -open` *pathname* `w` to open a file that your output will be written to. The `w` at the end specifies that a new file will be opened, or an existing file will be overwritten. Use `a` if you want to append to an existing file. When you are finished running Soar, close the file you opened above with the command: `log -close`

```
unix% soar                                      start Soar
Soar> source $soar_library/tutorial/blocks.soar

Loading blocks.soar
**********                                      One * for each production loaded
Soar> watch 1 productions -on                   Set the watch level
Soar> run 1 d                                   Run one decision cycle

      0:  ==>S: S1                              Default initial state

Firing elaborate*initial-state                  Firing a production
 2
Initial state has 1, 2, and 3 on the table.     Output from production

Firing propose*move-block                       Firing operator proposals
Firing propose*move-block
Firing propose*move-block
Firing propose*move-block
Firing propose*move-block
Firing propose*move-block

      1:  O: O4 (move-block 3 to 2)             Operator O4 randomly selected

Soar>
```

Most of the information in this example can be ignored for now. The important parts are the commands `watch 1 productions -on`, and `run 1 d`, and the lines that start

[4]Usually, operators make changes that, in turn, lead to new operators. However, exceptions exist which we will discuss in later in the tutorial.

with "Firing". The `watch` command sets the level of information that is presented by the trace. Level "1" tells Soar to print information for individual decisions. "Productions -on" indicates that Soar should print information about individual production firings. The `run` command tells Soar to begin matching and firing productions. The "1 d" arguments simply indicate that Soar should run for a single decision cycle. Each "firing" line indicates a production that is firing. Here, the productions are proposing `move-block` operators. As we will explain further in section 4, a single production can fire multiple times with different variable bindings.[5] Each set of variable bindings for a production defines an *instantiation* of the production.

However, more important for the moment, notice that there are 6 operators proposed, one for each possible block movement: 1 to 2, 1 to 3, 2 to 1, 2 to 3, 3 to 1, and 3 to 2. We can see this by finding out some information from the state, where the operators get attached. To do this, use the `print` command, to show the state, `S1`, which was created in the first line of the trace (`0:  ==>S: S1`). The `-depth 2` indicator tells Soar to print the attributes and values of the specified object, and if any of the values are objects themselves, to print out those object descriptions. The information below is slightly abbreviated and formatted differently than the actual printout for clarity. The listing shows that there are 6 `move-block` operators, `O4` through `O9` attached to the state, and it shows the Soar objects for the blocks that they are proposing to move and the destinations.

```
unix%  print -depth 2 s1                              Print s1 and its attributes
(S1 ^type state                                       S1 is a state
    ^clear B2 ^clear T1 ^clear B1 ^clear B3           S1 has 4 clear objects
    ^ontop O1 ^ontop O2 ^ontop O3                     S1 has 3 ontop relations
    ^operator O7                                      O7 is the selected operator
    ^operator O9 + ^operator O8 + ^operator O7 +      These operators
    ^operator O4 + ^operator O5 + ^operator O6 +      are proposed
    ^thing B3 ^thing B2 ^thing B1 ^thing T1)          S1 has 4 things
(B1 ^name 1 ^type block )                             Soar object B1
(B2 ^name 2 ^type block)                              Object B2
(B3 ^name 3 ^type block)                              Object B3
(T1 ^name table ^type table)
(O1 ^bottom-thing T1 ^top-block B1)
(I6 ^bottom-thing T1 ^top-block B2)
(I7 ^bottom-thing T1 ^top-block B3 )
(O9 ^destination B1 ^moving-block B2 ^name move-block) Operator O7
(O8 ^destination B2 ^moving-block B1 ^name move-block) proposes moving
(O7 ^destination B2 ^moving-block B3 ^name move-block) B3 (block 3)
(O6 ^destination B1 ^moving-block B3 ^name move-block) to B2 (block 2)
(O5 ^destination B3 ^moving-block B2 ^name move-block)
(O4 ^destination B3 ^moving-block B1 ^name move-block )
Soar>
```

[5]Although it looks like the productions fire sequentially, they are conceptually fired in parallel. The decision cycle (see section 9) includes separate steps for the firing of productions, and the evaluation of their results.

At the end of the decision cycle just displayed, Soar had selected operator O7. Soar selected this particular operator from the 6 proposed operators because the productions specify that all of these operators are *indifferent* with respect to the others. Indifferent statements[6] indicate to Soar that it should pick at random among the proposed operators when several are proposed.

Going one step further in the example (i.e., `run 1 d`), we can see three productions which fire to apply the operator, and one which terminates it. In addition, new `move-block` operators are proposed to move 3 back to the table, and to move 3 to 2.

```
Soar> run 1 d                               Run one decision cycle

Firing apply*move-block*remove-old-clear    Firing apply rules
Firing apply*move-block*remove-old-ontop
Firing apply*move-block*add-new-on

Firing terminate*move-block                 Suggest reconsider
Firing propose*move-block                   Propose new moves
Firing propose*move-block
Retracting propose*move-block               Retracting invalid moves
Retracting propose*move-block
Retracting propose*move-block
Retracting propose*move-block
Retracting propose*move-block

      2:  O: O12 (move-block 3 to table)    O12 is randomly selected

Soar>
```

The three `apply` productions do precisely those actions that we specified in section2, removing statements from the state that are no longer true, and adding new statements for new relations. We can use the print command again to verify the changes. Notice that there is no longer a statement that B2 (Soar's object for block 2) is clear, there is no longer a statement that 3 is on the table, and there is a new statement that 3 is `ontop` of 2 (O10). Because the blocks have been moved in working memory, the conditions for some of the previous `move-block` operator proposals are not longer met. For instance, block 2 is no longer clear and the proposal for moving some block requires that the block is clear. Thus, the two operator proposals that were proposed to move block 2 (O5 and O9) not longer match. Soar keeps track of these conditions and *retracts* productions whose conditions do not match current working memory contents. Retractions lead to the removal of any changes to working memory that came from those productions.[7]

[6]These indifferent statements, called preferences (see section 5), are not stored in Working Memory, so they do not show up on the traces.

[7]Again, these changes are not sequential, but are performed in waves of parallel firings and retractions as will be discussed in section 9. In this case, the retractions come as a result of the

```
Soar> p -depth 2 s1

(S1 ^type state
    ^clear B1 ^clear B3 ^clear T1                block 1, block 3,
    ^ontop O1 ^ontop O2                          and table are clear
    ^ontop O10                                   New ontop O10
    ^operator O12
    ^operator O4 + ^operator O11 + ^operator O12 +
    ^thing I2 ^thing I3 ^thing I4 ^thing T1)
(I2 ^name 1 ^type block)
(I3 ^name 2 ^type block)
(I4 ^name 3 ^type block)
(T1 ^name table ^type table)
(O10 ^bottom-thing B2 ^top-block B3)             block3 ontop of block2
(O1 ^bottom-thing T1 ^top-block B1)
(O2 ^bottom-thing T1 ^top-block B2)
(O11 ^destination B1 ^moving-block B3 ^name move-block)
(O12 ^destination T1 ^moving-block B3 ^name move-block)
(O4 ^destination B3 ^moving-block B1 ^name move-block)
Soar>
```

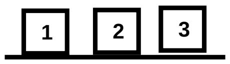
Soar will continue this process of proposing, selecting, applying, and terminating operators until it is halted by the user, or reaches a halt command in some production.

3.4 Exercise

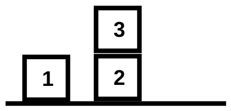
In the last step that we traced, Soar had applied operator `O7` and terminated it. Then it selected operator `O12` to move block 3 to the table. What steps of operator Proposal, Selection, Application, and Termination can bring this example to a successful finish; that is, one in which the goal of a 1-2-3 tower is accomplished? Use figure 3.4 to show the steps and the states that result from the operator applications (draw the boxes on the table, and color them as desired).

changes made to working memory by the earlier firing of the operator application productions.

Initial state



Propose O4, O5, O6, O7, O8, O9
Select O7 (3−>table)
Apply O7
Terminate O7



Propose O11, O12 (O4 still proposed)
Select O12 (3−>table)
Apply ___
Terminate ___

Propose _______
Select ______
Apply ___
Terminate ___

Propose ______
Select ______
Apply ___
Terminate ___

4 Productions 1: Instantiation

This section introduces productions, the basic unit of Soar programs. As previously mentioned, productions (or rules) are independent pieces of code that specify when they apply ("conditions") and what they do ("actions"). Whenever the conditions of some production match descriptions of the world in working memory, the production fires, and its actions propose changes to working memory.

This section starts with a description of the structure of working memory. We describe in general terms the syntax of productions and then introduce the process of 'matching and firing productions. A discussion of the precise syntax of productions will be postponed until section 7.

4.1 Working Memory

As suggested previously, Soar's state description contains knowledge about the objects in the world, and about their attributes and relations. For the initial state of the blocks world task we have been using, Soar contains these facts:

- There is a thing that is a block, whose name is 1.
- There is a thing that is a block, whose name is 2.
- There is a thing that is a block, whose name is 3.
- There is a thing that is a table, whose name is table.
- 1 is clear.
- 2 is clear.
- 3 is clear.
- table is clear (meaning there is space to put down a block on it).

To represent this information, Soar stores information about the world in its working memory, abstractly depicted in figure 2. Working memory is composed of a set of working memory elements (WMEs) each of which consists of an identifier, an attribute, and a value. The collection of WMEs with a single identifier is said to be the description of an object. In the figure, the boxes show the eight objects which describe the initial blocks world configuration. Notice that although some of the objects correspond to real physical entities in the world, some (S1, O1 — O3) are abstract objects representing relationships. For example, O1 says that there is an `ontop` relationship, with B1 as the block on top, and T1 as the thing on the bottom.

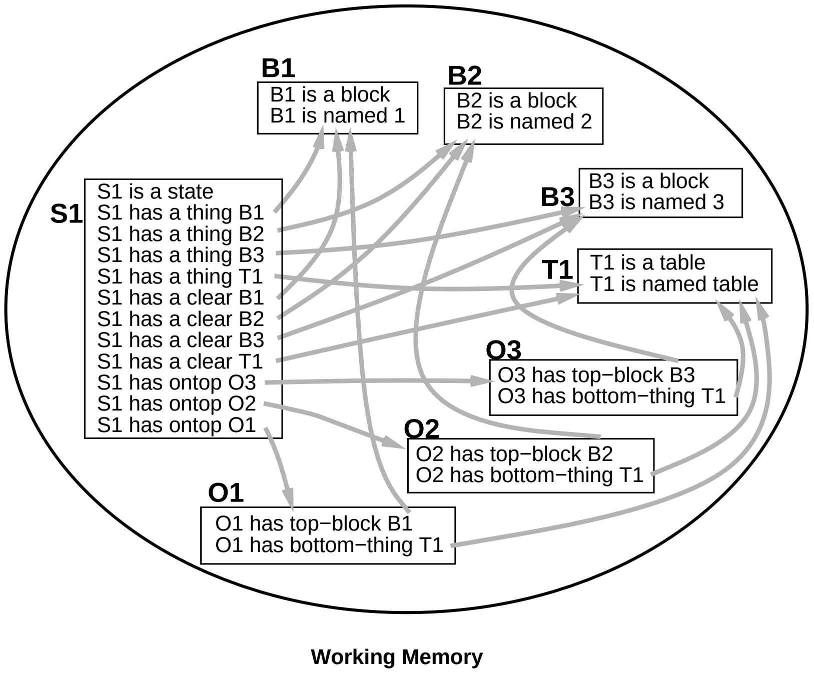


Figure 2: An abstract view of working memory, showing 26 working memory elements grouped into eight different objects.

4.2 Matching Productions

As we mentioned in the beginning of this section, the production is the basic unit of a Soar program. Productions consist of a set of conditions that denote when the production can fire, and a set of actions that are performed when the production fires. Figure 3 shows a schematic version of a Soar production that proposes the `move-block` operator.

<state1> is a state
<state1> has a thing <thing1>
<state1> has a thing <thing2>
<thing1> ≠<thing2>
<thing1> is clear
<thing2> is clear
<state1> has ontop <ontop1>
<ontop1> has top−block <thing1>
<ontop1> has bottom−thing ≠<thing2>
$\longrightarrow$
propose an operator to move <thing1>
ontop of <thing2>

Figure 3: A schematic Soar production for proposing the move-block operator

The nine lines above the arrow are the conditions of the production, and the line below the arrow is the action. The items between angle brackets (e.g. `<state1>`) are variables. They can match any object in working memory, but the different occurrences of a variable in a production must all match the same object. Please note that this production does not use the syntax of a Soar production, but it does have the main elements of one. The exact syntax of Soar productions will be described in section 7.

In order for a production to *fire*, its conditions must match elements of working memory. A production matches working memory if there is a consistent set of variable substitutions that satisfies all of its conditions. Figure 4 shows one match, or *instantiation*, of the example production. The set of variable substitutions is shown in the box with dotted lines. The dark, arrowed lines originate from the working memory elements which satisfy the conditions to which they point. These satisfied conditions form the instantiated production, represented in the solid box on the right.

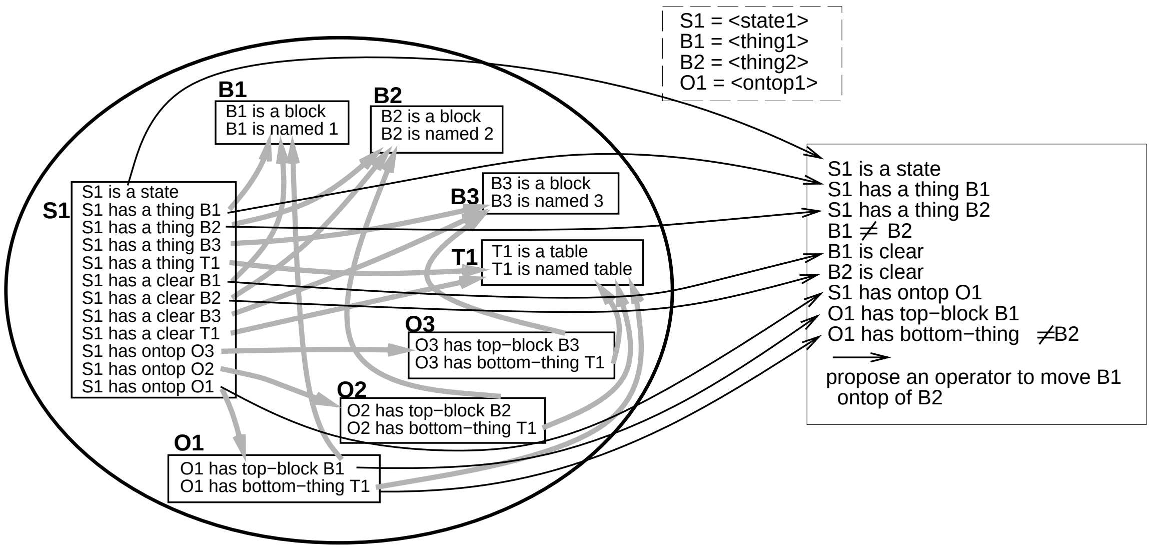


Figure 4: Variable substitutions instantiate a production.

The illustrated instantiation suggests an operator to move block `B1` on top of `B2`. In the initial state of our blocks world example, five additional instantiations are created from the `propose*move-block` production: one to move block `B1` on top of `B3` and to move block `B2` on top of `B1` and to move block `B2` on top of `B3` and to move block `B3` on top of `B1` and to move block `B3` on top of `B2`. (There are three blocks that are clear and an operator will be suggested to move each block on top of each other block.)

4.3 Running Soar: Instantiation

In this example, we will run the same program we ran in the previous section. However, now we will look at some new information in order to see the production instantiations as they occur. Start Soar and load the program in the same way that you did last time. (If you still have Soar running, you can just type the `init-soar` command to re-initialize Soar, as shown in the example.)

After loading the productions (if you started over) or initializing Soar, we tell Soar to fire one round of production firings, an *elaboration cycle*[8]. In this elaboration cycle, only an initialization production that establishes the initial state fires.

```
Soar> init-soar                               Restart Soar

Soar> run 1 e                                 Run one elaboration cycle

      0:  ==>S: S1                            Default initial state

Initial state has 1, 2, and 3 on the table.   Output from production

Soar> watch 1 productions -on -fullwmes       Set the watch level
Soar> matches                                 Show productions that match
Assertions:
  propose*move-block (6)                      6 different instantiations
Retractions:                                  for this production
Soar> run 1 d
Firing propose*move-block
 (5:  S1 ^thing B3)                           -fullwmes shows
 (12:  S1 ^clear B3)                          variable bindings for each
 (14:  S1 ^clear B1)                          production that fires
 (7:  S1 ^thing B1)
 (10:  S1 ^ontop O1)
 (23:  O1 ^top-block B1)
 (24:  O1 ^bottom-thing T1)
Firing propose*move-block                     Six total production firings
....                                          (one for each production instantiation)
Soar>
```

Now we want to look at some production instantiations. First, set the watch level as you did in the previous example, only add the flag `-fullwmes` to the `watch` command. This flag tells Soar to print the working memory elements (or "WMEs") that are bound to each production instantiation.

We can see which productions are ready to fire using the `matches` command. This command lists the productions whose conditions match working memory under `Assertions`, and those with conditions that no longer match under `Retractions`. At this point, six instantiations of the `move-block` proposal production are ready to fire. Next, tell Soar to perform one decision cycle so we can see the results.

Because we have added the `-fullwmes` flag to the `watch` command, now when Soar tells which production is firing, it also prints the WMEs which satisfied the production's conditions. In the example, we only show the variable bindings for the proposal which will create the operator to move block 1 to block 3 (`O4` in the last section).

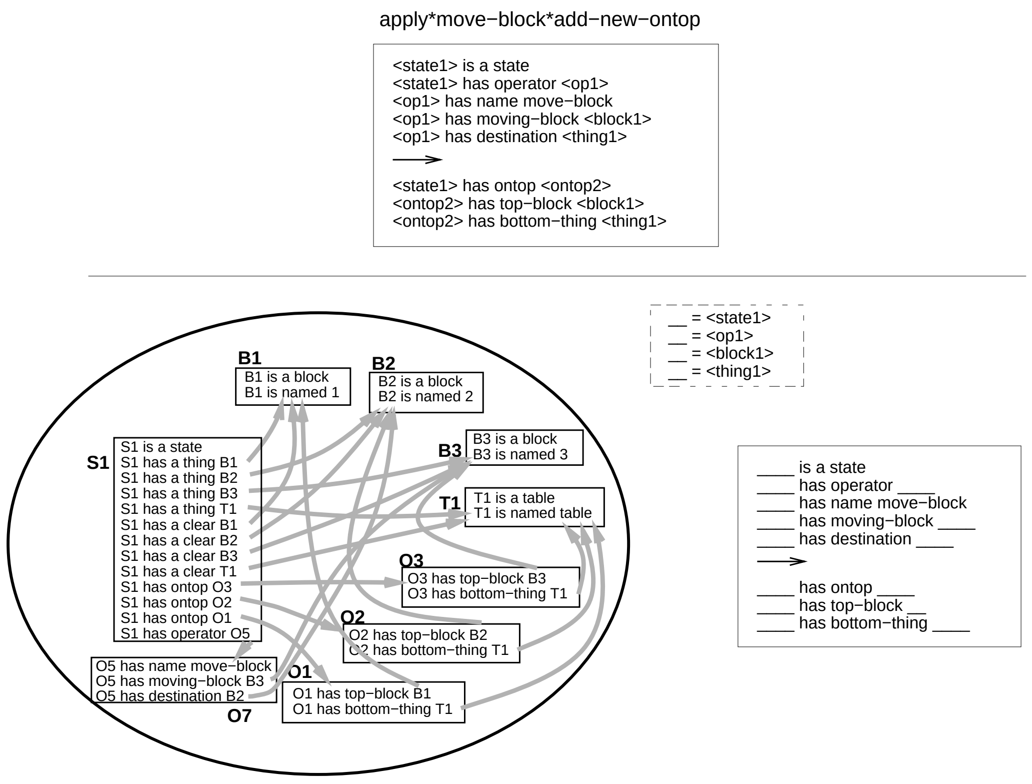
[8]Elaboration cycles are described in detail in section 9

Compare these bindings to those we showed in figure 4, for the operator to move block 1 onto block 2.[9] Each line lists the WME number, the identifier, the augmentation (attribute), and its value, which is often another identifier.

You can proceed with the blocks world program (`run 1 d` for each step) to view how the other productions are instantiated. You can also use the print function (`p -depth 2 s1`) to see the current state.

4.4 Exercise: Variable binding

Figure 5 shows working memory after an operator has been proposed and selected to move block 3 onto block 2. It also shows an application production for the `move-block` operator which adds the new `ontop` statement to show the moving block's new position. Fill in the blanks in the variable substitution box to show how this production will be instantiated. Draw lines from the WME's to the instantiated production to show that it can fire.



```
apply*move−block*add−new−ontop

<state1> is a state
<state1> has operator <op1>
<op1> has name move−block
<op1> has moving−block <block1>
<op1> has destination <thing1>
──>
<state1> has ontop <ontop2>
<ontop2> has top−block <block1>
<ontop2> has bottom−thing <thing1>
```

```
__ = <state1>
__ = <op1>
__ = <block1>
__ = <thing1>
```

```
____ is a state
____ has operator ____
____ has name move−block
____ has moving−block ____
____ has destination ____
──>
____ has ontop ____
____ has top−block __
____ has bottom−thing ____
```

```
B1 is a block
B1 is named 1
B2 is a block
B2 is named 2
B3 is a block
B3 is named 3
T1 is a table
T1 is named table
S1 is a state
S1 has a thing B1
S1 has a thing B2
S1 has a thing B3
S1 has a thing T1
S1 has a clear B1
S1 has a clear B2
S1 has a clear B3
S1 has a clear T1
S1 has ontop O3
S1 has ontop O2
S1 has ontop O1
S1 has operator O5
O3 has top−block B3
O3 has bottom−thing T1
O2 has top−block B2
O2 has bottom−thing T1
O1 has top−block B1
O1 has bottom−thing T1
O5 has name move−block
O5 has moving−block B3
O5 has destination B2
```

Figure 5: An instantiation exercise

[9]If you are checking carefully, you'll notice that Soar doesn't print out the binding for the state (`<state1> is a state`), or the inequality (`<thing1> not equal <thing2>`). The first is an automatic part of the Soar matcher, and the second doesn't require testing a WME — just the variable bindings.

5 Preferences

In the previous sections, we have lied to you. We have suggested that when Soar fires productions, the actions of the productions make changes directly to working memory. This suggestion is over-simplified. In this section, we describe an additional Soar memory (called Preference Memory) that stores *suggested* changes to memory that come from production firings.

These suggestions are called *preferences*, which we alluded to earlier when discussing indifferent operators in section 3. The use of preferences is a key distinguishing feature between Soar and other production systems. Preferences allow Soar programs to deliberately reason about every aspect of what they do. For example, a program can prefer one operator over another in a particular situation. A program can have several different ways of deciding what value the application of an operator gives. Also, when a Soar program can not resolve the preferences for operators into a single selection, Soar can invoke subgoaling (section 13) and learn about the results (section 15).

In contrast, other production systems (e.g., OPS-5) use a static conflict resolution mechanism that chooses between competing rules based on some metric (e.g., "weights"). This style of conflict resolution is simpler, but doesn't allow for reasoning or learning about conflict resolution. In addition, it requires that the programmer keep track of the various weights used in various parts of the program.

This section describes:

- what preference memory is
- the relationship between preference memory and working memory
- operator preferences and attribute preferences

5.1 Preference memory and working memory

A Soar program *suggests* changes to working memory by placing *preferences* in *preference memory*. Preferences are quite similar to WMEs, except that they contain an additional element which specifies how Soar should treat them when considering what to put into working memory. `Acceptable` preferences, indicated by `+`, tell Soar that the suggested WME change is allowed. Although there are other kinds of preferences, an acceptable preference is required for each WME additional. Another example of a preference is a parallel preference, indicated by `&`. This preference tells Soar that it can have multiple values for that attribute (if each of the values for that attribute has the parallel preference). Figure 6 shows some of the preferences that exist in our blocks world example after the initializing production has fired.

Preferences are suggestions about the information that Soar should have in its working memory. Different productions can create preferences for conflicting working memory elements. For example, one preference might suggest that "B1 is named 1" be added

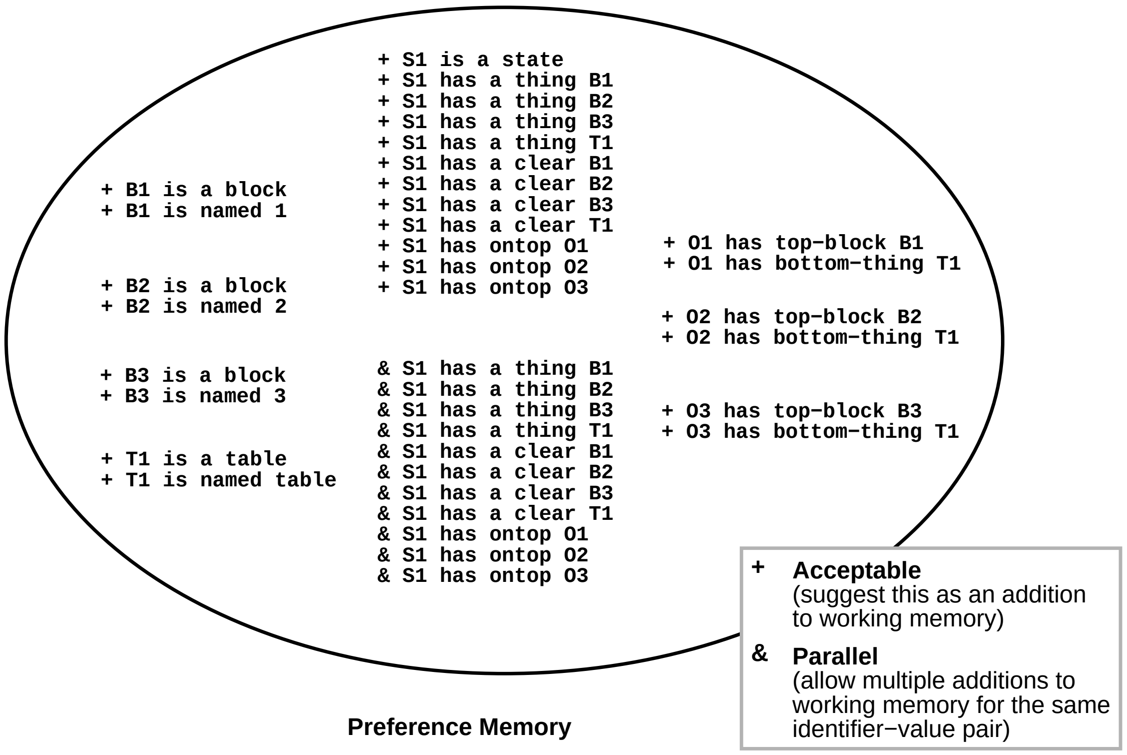


Figure 6: An abstract view of preference memory. In this illustration, there are `acceptable` and `parallel` preferences for the `thing`, `clear`, and `ontop` attributes of `S1`. All other attributes have only `acceptable` preferences.

to working memory, but another may suggest that “B1 is named 2” be added to working memory. In most cases, an identifier-attribute pair may have only one value in working memory; Soar must *resolve* the preferences so that working memory is not contradictory. When the preferences cannot be resolved, Soar is said to have reached an *impasse*: additional progress can not be made based on the current state of preference and working memory. In general, Soar responds to impasses by creating subgoals in an attempt to bring additional knowledge to bear. Impasses are more fully discussed in section 13.

For operators, preferences are used functionally to say when an operator should be selected, and when it should be terminated. When an operator is given an acceptable preference (`+`), it is available for selection by Soar. If there are multiple acceptable operators available, preferences can be used to tell Soar which operator it should select. For example, one preference might say that if you don’t know what else to do, it’s *acceptable* to pick an operator to move block `1` to the table; a second preference might say that it’s *acceptable* to pick an operator to move block `2` on top of block `3`; and a third preference might say that in this case it’s *best* to pick an operator to move block `2` on top of block `3`. These three preferences would be resolved to pick an operator to move block `2` on top of block `3`. Preferences are created by rules which test the current state, so they can be used to control what Soar does in specific situations.

Operators are terminated by productions which recognize when the operator's actions have been completed. These productions issue a reconsider (`@`) preference for the operator, telling Soar to make another choice on what the currently selected operator should be.

For states, Soar uses preferences primarily to add attributes and values to working memory, and to take them out. The acceptable preference for an attribute/value pair allows Soar to put that pair into working memory. The parallel preference (`&`) allows Soar to store multiple values for the specified attribute. The reject preference (`-`) indicates to Soar that the WME should no longer be in memory, by overriding the acceptable preference.

There are other preference that are commonly used in Soar that we will not describe in detail here. For additional information on these, consult the Soar manual.

5.2 Running Soar: Preferences

For this demonstration, re-initialize or restart Soar as before and run an elaboration cycle. Now we tell Soar to not only tell us which productions are firing (the `productions -on` flag) but what preferences each of these productions is creating with the `preferences -on` flag. Continue running the decision and you will see preferences created for each of the six `move-block` operators we have seen in the previous runs. The arrow (`-->`) below the firing production simply indicates that the preferences are created via the action side of the production. For each production firing, an acceptable preference is created for the ^`destination`, ^`moving-block`, and ^`name` augmentations of the operator.[10] An acceptable preference is also created for the operator augmentation. Finally, an unary indifferent preference is created for the operator. This preference is "unary" because it doesn't refer to another operator; it says that this operator is indifferent to all other operators. "Binary" preferences can be used to make a preference between two specific operators.

Another way to examine preferences is with the *preferences* command. Check the preferences for the operator augmentation of S1 (`preferences S1 operator`, as shown in the example). Soar tells us that there are acceptable preferences for six operators and that all of them are indifferent. Now proceed one more decision cycle (you can ignore the intervening output), and check the operator preferences again. There will now be a reconsider preference (`@`) for the operator selected in the first decision. Look back through the trace and determine what production created this preference.[11]

[10]The [O] indicates that these preferences are persistent or "operator-supported." We will discuss o-support is section 6. For now, just ignore the [O].

[11]The production that issued the reconsider preferences was `terminate*move-block`.

```
Soar> init-soar                                   Restart Soar

Soar> run 1 e                                     Run one elaboration cycle

      0:  ==>S: S1                                Default initial state

Initial state has 1, 2, and 3 on the table.       Output from production

Soar> watch 1 productions -on -preferences -on   (Set the watch level)
Soar> run 1 d
Firing propose*move-block
 -->                                              Indicates action side of a production
 (O4 ^B3 + [O] )                                  Acceptable preference for destination
 (O4 ^moving-block B1 + [O] )                     Acceptable preference for moving block
 (O4 ^name move-block + [O] )                     Acceptable preference for operator's name
 (S1 ^operator O4 =)                              This operator is indifferent
 (S1 ^operator O4 +)                              This operator is acceptable
Firing propose*move-block                         Six sets of new preferences
....                                              (one for each production firing)
Soar> preferences S1 operator                     Print specific preferences
Preferences for S1 ^operator:
acceptables:
 O4 (move-block 1 to 3) +
 O5 (move-block 2 to 3) +
 O6 (move-block 3 to 1) +
 O7 (move-block 3 to 2) +
 O8 (move-block 1 to 2) +
 O9 (move-block 2 to 1) +

unary indifferents:
 O4 (move-block 1 to 3) =
 O5 (move-block 2 to 3) =
 O6 (move-block 3 to 1) =
 O7 (move-block 3 to 2) =
 O8 (move-block 1 to 2) =
 O9 (move-block 2 to 1) =
Soar>
```

5.3 Exercises: Preferences

Figure 7 shows the translation of production `apply*move-block*add-new-on` which fires when the current operator is a move-block and adds the information that the moving block is `ontop` of the destination. Assume that `<state1>` is bound to `S1`, `<ontop>` is bound to `O3`, `<block1>` is bound to `B2`, and `<thing2>` is bound to `T1`. On the blank lines, enter the preferences that will be put into preference memory for the object/attribute/value combinations shown. **Hint:** Remember acceptable preferences are assumed if no preferences is specified in the action side of a production.

Psuedo−production
apply*move−block*add−new−on

```
 <state1> is a state
 <state1> has operator <op1>
 <op1> has name move−block
 <op1> has moving−block <block1>
 <op1> has destination <thing2>
 ——>
<state1> has ontop <ontop> with
  acceptable and parallel preferences
<ontop> has top−block <block1>
<ontop> has bottom−thing <thing2>
```

Preferences

```
S1 ^ontop O3 ___  ___
O3 ^top−block B2 ___
O3 ^bottom−thing T1 ___
```

Figure 7: Attribute preferences: fill in the blanks

Exercise 2: Recite this mantra 10 times:

> Productions match working memory elements and add preferences to preference memory.

This point is important to remember. The left-hand sides of productions and the right-hand sides of productions look similar, but they are doing quite different things.

6 Persistence

Preferences let Soar determine what goes into working memory and what comes out. How long a preference stays active depends on how the preference was created. Preferences can be separated into two classes: persistent preferences and dependent preferences or elaborations. This can be confusing because both types are created by productions. So for persistent preferences, we have a rule like:

> If A, then B

And we want B to remain true until some other rule changes it.

But for dependent preferences, we have the same type of rule:

> If C, then D

But in this case, D is dependent on C, so that if C is removed (from working memory), D is automatically removed too.[12]

This section explains:

- How operator application creates persistent preferences
- How elaborations create dependent preferences
- How persistence determines how long a preference stays in preference memory

6.1 Persistence and Elaboration

In Soar, when an operator suggests a change in memory, that change stays in memory indefinitely; i.e., until the WME is explicitly rejected as being no longer appropriate. For example, when the `move-block` operator is applied, it changes which blocks are clear, and which are ontop of others. This modification persists until another operator changes the state again. Preferences that come from applying operators are called persistent changes.

Another class of changes to memory, which we have not seen in the blocks world example, are called elaborations. Elaborations refine Soar's state knowledge by using production knowledge to "elaborate" the current situation. For example, suppose that our Soar program needed to invoke a particular action when one block was above — and not necessarily directly on — another block. This information doesn't show up directly on the state. One solution would have an operator proposal production that

[12] Because the action sides of productions produce items in preference memory, this is a removal of D from preference memory which will result in the corresponding WME being removed from working memory if there is no other acceptable preference for it.

checks if block A is on block B, and another to check if block A is on block C, which is on block B. However, this solution may become more unworkable for large towers of blocks. Instead, a Soar program can add additional information to the state through elaboration. In this case, we can elaborate the state with an `above` attribute. The following rules could make the addition and will create above attributes for towers of any size:

> If block `<A>` is **`ontop`** block `<B>`, then block `<A>` is **`above`** block `<B>`.
>
> If block `<A>` is **`ontop`** block `<B>`, and block `<B>` is **`above`** block `<C>`, then block `<A>` is **`above`** block `<C>`.

The important thing to realize is the difference between the `above` attribute, and those attributes that are created by operators. Changes to memory made by operators remain intact until altered by other operators because they reflect changes to the actual "environment." Elaborations, on the other hand, are dependent only on what is currently in working memory. When a WME that was used to make an elaboration is removed from working memory, the elaboration instantiation no longer matches and is retracted. We saw examples of retractions in section 3. When the elaboration is retracted, the preferences it created are removed as well. This retraction then "reverses" the effect of the production that created the elaboration, resulting in the removal of any WME changes it made.

In this example, if block A is moved off of block B, the operator will remove the `ontop` relation. This change causes the retraction of the first rule (because A is no longer on top of B). Because there is no longer an acceptable preference for the `above` relation, Soar automatically removes it. Thus, operators need not keep track of elaborations.

Consider another example. The set of relevant operators should change as the state changes. By proposing operators as elaborations of the state, operator suggestions will be retracted when they no longer apply to the current situation. If proposals were made through the action of other operators, then proposals would persist even after state changes made the proposed operators irrelevant (or, worse, bad things to do).

Thus by using refinements, all the effects and side-effects of an operator do not have to be explicitly listed as part of the application of the operator. Refinements provide detail to the state when appropriate and are automatically retracted when no longer appropriate.

Technically speaking, we say that attributes which are created by operator application have *O*-support (the O comes from Operator). Those preferences with O-support are marked in memory with `[O]`, as shown in the example below. Elaborations are said to have *I*-support (from Instantiation), and these preferences have no special marker in listings of preference memory.

6.2 Running Soar: Persistence

```
Soar> init-soar                                      Restart Soar
Soar> run 1 d                                        Run one decision cycle
      0:  ==>S: S1
Initial state has 1, 2, and 3 on the table.
      1:  O: O7 (move-block 3 to 2)
Soar> preferences S1 ontop
Preferences for S1 ^ontop:
acceptables:
 O1 +
 O2 +
 O3 +
unary parallels:
 O1 &
 O2 &
 O3 &
Soar> matches                                        Print matching productions
Assertions:
 apply*move-block*add-new-ontop                      Applications ready to fire
 apply*move-block*remove-old-ontop
 apply*move-block*remove-old-clear
Retractions:
Soar> watch 1 productions -on preferences -on
Soar> run 1 e                                        Fire the operator applications
Firing apply*move-block*remove-old-clear
 -->
 (S1 ^clear B2 - [O] )                               Remove (reject) B2 clear
Firing apply*move-block*remove-old-ontop
 -->
 (S1 ^ontop O3 - [O] )                               Remove 3 ontop of table
Firing apply*move-block*add-new-ontop
 -->
 (S1 ^ontop O10 + [O] )                              Add new ontop
 (S1 ^ontop O10 & [O] )                              with parallel preference
 (O10 ^bottom-thing B2 + [O] )                       block 2 on bottom
 (O10 ^top-block B3 + [O] )                          block 3 on top
Soar> preferences S1 ontop
Preferences for S1 ^ontop:
acceptables:
 O10 + [O]                                           O10 has operator support
 O2 +                                                O2 has instantiation support
 O3 +
unary parallels:
 O10 & [O]
 O2 &
 O3 &
Soar>
```

Start Soar and load the `blocks.soar` productions. Run one decision. Let's look at the preferences for the `ontop` relation on `S1`. An initializing production created the initial state with the `ontop` relations `O1` (block 1 on the table), `O2` (block 2 on the table), and `O3` (block 3 on the table). Because this production tested only the state (no operators were selected until the first decision), this instantiation is s-supported.

Now look at what is ready to fire with the `matches` command. The three operator applications for `move-block` are ready to fire. Set the watch level to see the production firings and the preference changes, and then run an elaboration cycle. Each application makes a change to the state to "move" block 3 on top of block 2. First, *apply*move-block*remove-old-clear* removes the clear augmentation for block 2. It accomplishes this removal by issuing a reject preference (`-`) for the B2's clear augmentation.[13] Similarly, *apply*move-block*remove-old-ontop* issues a reject preference for `O3` (block 3 on the table). Finally, *apply*move-block*add-new-ontop* creates a new `ontop` relation in which block 3 is on top of block 2.

Notice that the preferences for all these actions also included a `[O]` after the preference itself. This symbol simply denotes that the preference has operator support, as we noted above. For instance, if you now look at the preferences for the `ontop` relation, as we did earlier, you will see that the new `ontop` relation, `O10`, has o-support. Remember that any preference without the `[O]` has instantiation support (i.e., i-support is the default support type).

6.3 Exercise: Persistence

1. Are the `ontop` and `clear` relations created by the operator application productions persistent or are they elaborations?

2. There is a production which fires at the beginning of the blocks example which puts on the state the information about the initial positions of the blocks, and their relations. The condition of this production is that there is a state. Are the `ontop` and `clear` relations created by this production persistent or are they elaborations?

3. Are operator proposals persistent or elaborations?

4. When can persistent preferences be removed?

5. At what point will elaborations be removed?

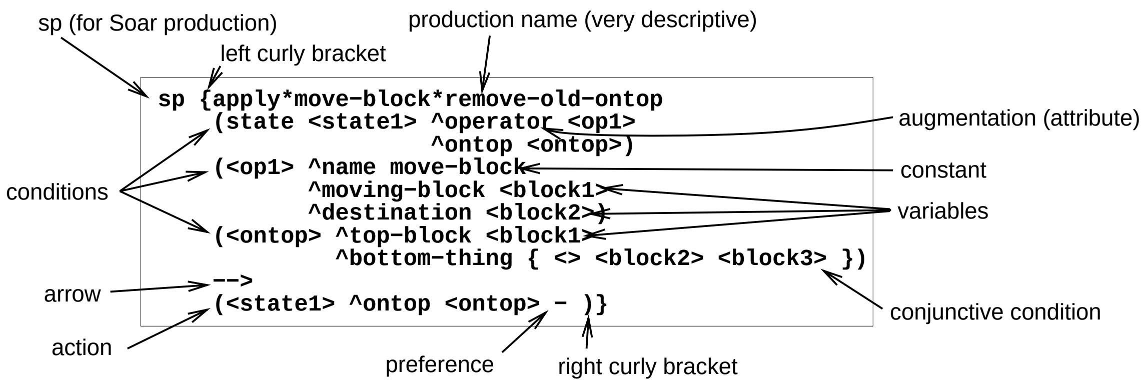
[13] Remember the action side of productions never directly change memory. When Soar evaluates the preferences for the `^clear` augmentation, it will determine that the clear augmentation for B2 has been rejected, and then remove it from memory.

7 Productions 2: Syntax

In the previous sections, we have described the basic components of a production, but we have left out most of the detail of how the production is structured. In this section, we will delve into the specifics of production syntax. The goals of this section are to:

- give the overall structure of a production, including conditions (left-hand side or LHS) and actions (right-hand side or RHS)
- introduce variables and symbols
- show how they are used in the creation of production clauses
- show the components of the condition and action clauses
- show some of the shortcuts for writing simpler productions

Figure 8 shows the basic components of a Soar production. This production applies the `move-block` operator to remove the old `ontop` relation, as we looked at in section 6. When the current operator is a `move-block` operator to move block `<block1>` and `<block1>` is currently `ontop` of `<block3>` (which is not the block that it is moving to), it removes that `ontop` relation from the state. It does that by creating a reject (`-`) preference for that augmentation. This section describes the components of productions and how they fit together.



```
sp {apply*move-block*remove-old-ontop
   (state <state1> ^operator <op1>
                   ^ontop <ontop>)
   (<op1> ^name move-block
          ^moving-block <block1>
          ^destination <block2>)
   (<ontop> ^top-block <block1>
            ^bottom-thing { <> <block2> <block3> })
   -->
   (<state1> ^ontop <ontop> - )}
```

Figure 8: The anatomy of a production

To give you a feeling for how the syntax of the production maps into its semantics, figure 9 shows a translation of this same production into the pseudo-English style used previously in this document.

Soar Production

```
sp {apply*move-block*remove-old-ontop
    (state <state1> ^operator <op1>
                    ^ontop <ontop>)
    (<op1> ^name move-block
           ^moving-block <block1>
           ^destination <block2>)
    (<ontop> ^top-block <block1>
             ^bottom-thing { <> <block2> <block3> })
    -->
    (<state1> ^ontop <ontop> - )}
```

Translation

```
<state1> is a state
<state1> has an operator <op1>
<state1> has an ontop <ontop>
<op1> has name move-block
<op1> has moving-block <block1>
<op1> has destination <block2>
<ontop> has top-block <block1>
<ontop> has bottom-thing <block3>
            different from <block2>
⟶
<state1> rejects ontop <ontop>
```

Figure 9: A translated Soar production

7.1 Basic production structure

The basic elements that exist in all productions are:

- `sp`: to designate this as a Soar production
- left curly bracket ({): marks the beginning of the production body
- production name: Productions are name by the person developing a Soar model. By convention, production names are usually descriptive.
- conditions: a set of condition clauses which match working memory
- arrow (`-->`): separates condition side and action side
- actions: a set of action clauses for preferences to be added to preference memory
- right curly bracket (}): marks the end of the production

The complex parts of the production syntax are the condition and action clauses. These will be described below. However, they make frequent use of *variables* and *symbols*, so we will define these first.

7.2 Variables

Syntactically, a variable is a symbol that begins with a left angle-bracket (`<`), ends with a right angle-bracket (`>`), and contains at least one alphanumeric symbol in between. As shown in the preceding sections, variables allow conditions and actions to match against a variety of WME's by binding the variables to specific WMEs and then replacing additional variable references with the appropriate substitution.

Figure 4 gave one example of variable substitutions being used to instantiate a production. Figure 10 shows how the same condition can match several different WMEs

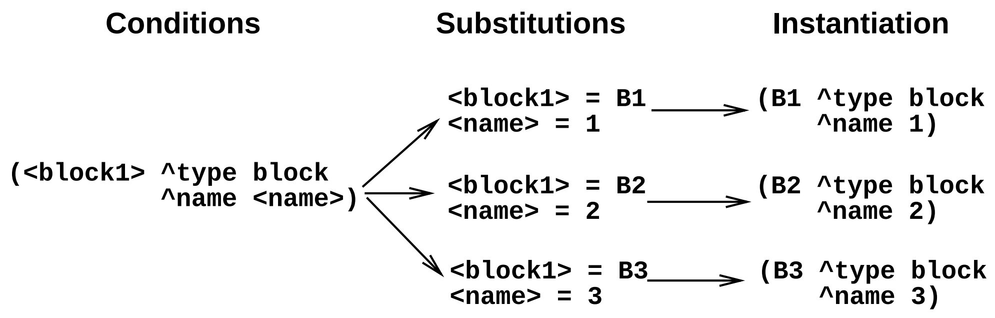


Figure 10: Variable substitutions allow multiple instantiations

using variables bound to different values. (The exact syntax of conditions is described below.)

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

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

7.3 Symbols

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

Identifiers A Soar identifier represents an object in working memory. The set of WMEs that start with the same identifier constitute the description of the object which that identifier represents. For example, the WME's `(O2 ^top-block B2)` and `(O2 ^bottom-thing T1)` are both augmentations of the abstract object O2, which in our examples is an `ontop` relation. When Soar prints an object in traces, it combines the augmentations, for example `(O2 ^top-block B2 ^bottom-thing T1)`. The exact symbol used as an identifier is arbitrary, and *cannot be matched in the conditions or specified in the actions of a production except by a variable.* When an object is created (either by a production or input), the identifier is generated automatically by Soar. The format of an identifier is a single upper-case letter followed by an integer, such as `B4`.

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

- Integer constants (numbers). In the blocks world example, the names of the blocks are integers.

- Floating-point constants (numbers). Floating point number are not used in the blocks world but are used by Soar.
- Symbolic constants. Symbolic constants are symbols with arbitrary alphanumeric strings; for example: `table`. They are designated in productions by their names.

7.4 Condition side

The condition side of a production, also called the left-hand side (abbreviated: LHS) of the production, describes one or more objects that may exist in working memory. When all of the conditions of a production match elements in working memory, the production is instantiated, leading the production to fire in the next elaboration cycle.

In order to make our description of the conditions more clear, we'll take the condition side of our previous example production, apply*move-block*remove-old-ontop, as an example. The conditions for this production are:

```
(state  <state1> ^operator <op1>
                 ^ontop <ontop>)
(<op1> ^name move-block
       ^moving-block <block1>
       ^destination  <block2>)
(<ontop> ^top-block <block1>
         ^bottom-thing { <> <block2> <block3> })
```

Each condition is a description of an object that may exist in working memory. At its most simple, a condition will describe an individual WME, such as `(O3 ^top-block B2)` or `(O3 ^bottom-thing T1)`, but multiple WME's of the same object are usually combined, such as `(O3 ^top-block B2 ^bottom-thing T1)`.

Conditions are not literal descriptions of working memory elements. They must contain variables (as described above) to specify the identifiers of the objects to which they refer. Variables allow conditions to specify the identifiers that link the conditions together without knowledge of the literal identifiers that appear in working memory. They will most frequently appear as the identifier or value in a condition. Less commonly, variables may also be used for the attribute in a condition.

For a production match, all occurrences of the same variable must bind to the same symbol. In the example above (figure 10), there are two occurrences of each of `<block1>` and `<block2>`. Both occurrences of `<block1>` must bind to the same identifier, such as `B1`; and both occurrences of `<block2>` must bind to the same identifier, such as `B2`. The variable `<block3>`, used in a conjunctive condition with an inequality to `<block2>`, is described below.

7.4.1 Conditions must be "linked"

An object is *linked* to a second object if its identifier is the value of an augmentation of the second object. One of the conditions in a production must specify a state or an impasse in working memory, and the rest of the conditions must be linked to this condition. All conditions must be linked to the state (or impasse), but the linkage may be indirect, through other conditions in the production.

The condition that specifies the state or impasse must start with the symbol `state` or `impasse` to identify the state or impasse which is the root of the links.

For example, a condition of a production could look like this:

```
(state <s> ^superstate nil)
```

The matcher will ensure that this condition matches either states or impasses, whichever is specified. (A useful convention is to have the state or impasse in the first condition.) If there is not a state or impasse condition, or if all conditions are not linked, a warning will be printed when the production is loaded. (The production will be stored in production memory, but it may never fire, or may match in situations that you didn't intend.)

Important Note: You you might think that the simple condition:

```
(state <s>)
```

by itself on the LHS of a production would just fire once when a state is created; however, this is not necessarily true. This condition matches every WME with a state as its first item, such as `(S1 ^type state)`, `(S1 ^superstate nil)`, and so on. Thus, multiple instantiations of this rule would be created with the result that it would fire once for each WME with the identifier state. To ensure a production just fires once for the top-state, use the condition: `(state <s> ^superstate nil)`.[a]

[a] `^superstate` is created by the architecture whenever a new state is created. The value of this WME is a pointer to the immediately higher state (which is NIL for the first or top state). See section 13 for more information about substates and superstates.

In order to examine some more advanced syntactic features of productions, consider the following conditions:

> one block is blue or red;
> another block is red;
> the red block weighs less than 5;
> the blue or red block is not clear;
> and the texture of the covering of the red block is rough

These conditions are simply for illustration; the blocks world example we have been using does not use these features. We want to use these condition to introduce the syntax for:

- disjunctive conditions
- predicates
- conjunctive conditions
- negations

We will *not* cover these techniques in great detail. For additional information, please refer to the manual.

7.4.2 Disjunctive conditions

The condition `(<block1> ^color red)` will allow you to match against any red block. However, what kind of condition must you write to if you want to match against red *or blue* blocks? Soar allows you to write disjunctive conditions that can match against one of many possible values. In this example, the disjunctive condition `(<block1> ^color << red blue >>)` would match for any WMEs that had red or blue values for a color attribute.

Disjunctions specify that a WME must match one of a set of constants. No variables can be used in a conjunction, and *there must be spaces separating the brackets from the constants.*

The following table provides examples of legal and illegal disjunctions:

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

7.4.3 Predicates

Predicates are used to specify that an attribute's value satisfies some relation to something else. In our example, we need predicates for two reasons: to specify that the second block is different from the first, and to show that the weight is less than 5. Let's solve the second problem first.

We have already seen how to specify that an attribute has a particular value when we identified the "names" of the blocks. In order to specify that an attribute's value is less than another value, we can just use the `less than` predicate, `<`, like this:

```
(<block2> ^weight < 5)
```

It's very important to view the `< 5` as a unit which specifies a condition that the value must satisfy, just as the condition `(<block2> ^name <name>)` enforces the condition that the value of the `name` attribute bind with the variable `<name>`. We'll return to this requirement below.

We can use the `not-equal`, `<>`, predicate to specify different blocks. This predicate can be used to contrast any symbol, not just numbers. We will use it in the current example to specify that the first block is distinct from the second. To do this, we will use the condition:

```
(<state> ^thing <> <block1>)
```

This specifies that the value of this `thing` attribute is different — a different identifier — from the one which `<block1>` points to. This condition is not quite complete yet, because we also want to bind a variable to this other block. We'll see how to do this in the next section.

There are other predicates available that compare numbers and one that compares types. See the manual for details.

7.4.4 Conjunctive conditions

To complete the "another block is not red" condition, we need to say that the value of the `thing` attribute is not the same as the value of `<block1>` *and* that it binds to `<block2>`. We use a conjunction, indicated by curly brackets {}, to complete this condition. To specify a set of conjunctive constraints, we just put all of the constraints within curly brackets where a single constraint would be. In this case, we have the two constraints: `<> <block1>` and `<block2>`. The resulting condition is:

```
(<state> ^thing { <> <block1> <block2> })
```

Note that this is a little non-intuitive because there is nothing to separate the conjuncts. We could also write:

```
(<state> ^thing { <block2> <> <block1> })
```

This second example means exactly the same thing as the first, even though it looks like something different. The brackets specify two conjunctive conditions. One condition says that the value of the thing attribute will bind with `<block2>`, and the other says that it is not the same as the value of `<block1>`.

Any number of conjuncts are allowed in a conjunctive condition. Conjunctions can be used in *negated clauses* also, but such conjunctions are not discussed in this document. See the manual for more information.

7.4.5 Simple Negations

Negations can be used in many ways, and here we will show just one. Again, for details, see the manual. In order to specify that `<block1>` is not clear, we write:

```
- (<state> ^clear <block1>)
```

This condition requires that there is *no* `clear` augmentation of state with the value the same as that bound to the variable `<block1>`.

A negation can also be used inside a condition, with the same meaning, for example: `(<state> -^clear <block1>)`. Negating within a condition is useful for specifying a condition with multiple augmentations, because the negation applies only to the negated augmentation, and not to the others. For example,

```
(<state> -^clear <block1> ^thing <block1>)
```

indicates that variable `<block1>` must be bound to a `thing` augmentation but that it can not also be bound to a `clear` augmentation.

7.4.6 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 (or "dot" notation) provides a shorthand to avoid the inclusion of these intermediate variables.

For example, in order to specify that the texture of the covering of the red block is rough, we could write (assuming that `covering` is a structured attribute):

```
(<block2> ^covering <cov>)
(<cov> ^texture rough)
```

Dot notation allows us to combine both of these into one equivalent condition:

```
(<block2> ^covering.texture <rough>)
```

Not that if we had to check another attribute of the covering, `density` for example, we would want to keep the clauses split:

```
(<block2> ^covering <cov>)
(<cov> ^texture rough ^density hard)
```

The completed production left-hand side, for the conditions we introduced earlier, is presented below. Take a close look at the conditions and make sure you understand the syntax of the individual conditions. Also note how the variable bindings constrain the possible matches for the conditions. For instance, `<block1>` must be a thing, must be color red or blue, and can not be clear.

```
(<state> ^thing <block1>
         ^thing { <block2> <> <block1> }
         -^clear <block1>)
(<block1> ^color << blue red >>)
(<block2> ^color red ^weight < 5)
(<block2> ^covering.texture rough)
```

7.5 Action side

As we mentioned in section 5, clauses on the action side (or RHS for right-hand side) of productions look very similar to those on the condition side, but they do completely different things. There are two types of actions: *preference clauses* and `functions`. The most commonly used function is `write`, which prints out messages to the terminal. For example, in our blocks world example, `write` is used to tell us that the state has been initialized, or that the goal state has been reached.

The most important actions are the ones that add preferences to preference memory. When the conditions of a production can be satisfied, the production is instantiated, and fires during the next elaboration cycle. Production firings perform the actions indicated in the right-hand side, *using the same variable bindings* that formed the instantiation.

There are 13 different preferences that can be created on a production RHS. As mentioned in section 5, we will only concentrate on a subset of them in this document. The preferences we will discuss are shown with the syntax of their respective clauses below:

RHS preferences	**Semantics**
`(id ^attribute value)`	acceptable
`(id ^attribute value +)`	acceptable
`(id ^attribute value -)`	reject
`(id ^attribute value > value2)`	better
`(id ^attribute value >)`	best
`(id ^attribute value =)`	unary indifferent
`(id ^attribute value = value2)`	binary indifferent
`(id ^attribute value &)`	unary parallel
`(id ^attribute value & value2)`	binary parallel
`(id ^attribute value @)`	reconsider

Any of the identifier, attribute, or value(s) may be variables, such as `(<s1> ^operator <o1> > <o2>)`.

Notice that the preference notation appears similar to the predicate tests that appear on the left-hand side of productions. However, the meanings are wholly different. Predicates can not be used on the right-hand side of a production and you can not 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 on the right-hand side of a production, although using this symbol will make the semantics of your productions more clear in many instances. The `+` symbol will always appear when you inspect preference memory (with the `preferences` command). Because this symbol is optional, it is sometimes easy to forget that productions create preferences, and do not create working memory elements directly.

Multiple preferences for the same value of an attribute may be included at the same time, but when they are, the acceptable preference must be made explicit. For example:

```
...
-->
(<s> ^ontop <on1> + <on1> &)
(<s> ^ontop <on2> + <on2> &)
(<on1> ^top-block <b1> ^bottom-thing <b2>)
(<on2> ^top-block <b2> ^bottom-thing <b3>))
```

Thirdly, multiple preferences for the same value may be listed without repeating the value, as in:

```
...
-->
(<s> ^ontop <on1> + & ^ontop <on2> + &)
(<on1> ^top-block <b1> ^bottom-thing <b2>)
(<on2> ^top-block <b2> ^bottom-thing <b3>))
```

Finally, multiple preferences for different values of the same attribute may be included at the same time (separated by commas if necessary for clarity or to disambiguate unary and binary preferences), such as:

```
...
-->
(<s> ^ontop <on1> + &, <on2> + &)
(<on1> ^top-block <b1> ^bottom-thing <b2>)
(<on2> ^top-block <b2> ^bottom-thing <b3>))
```

Note that 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 an up-arrow, a closing parenthesis, or a comma follows it.

Other functions available in the action side of a production include a limited set of arithmetic functions. These functions provide the only way to do arithmetic in Soar.

For example, if you want to check if a variable that takes integer values is equal to another variable minus one, you must place the value of the variable minus one on the state as an elaboration, and then check the first variable against that elaboration.

7.6 Exercises: Syntax

1. Figure 11 shows another pseudo-English operator application production for the `move-block` operator. This production adds the new information that the moving-block is on the destination block. Fill in the blanks to complete the Soar production.

Translation

```
<state1> is a state
<state1> has an operator <op1>
<op1> has name move−block
<op1> has moving−block <block1>
<op1> has destination <thing2>
⟶
<state1> has ontop <ontop> with
    accept and parallel preferences
<ontop> has ^top−block <block1>
<ontop> has ^bottom−thing <thing2>
```

Soar Production

```
sp {apply*move−block*add−new−on
    (state <state1> ^operator _______)
    (<op1> ^name _____________
           ______________  <block1>
           ^destination  ________)
    −−>
    (<state1> ^ontop <ontop>  ___  ___)
    (________ ^top−block _________
              ____________  <thing2> )}
```

Figure 11: Translate this Soar production

2. Next, write the following productions that might be needed for a slightly different task:

```
propose*destroy-block:
        if there is a block,
        and another (different) block,
        and the first block is red,
        and the red block is ontop the other block
        then
        propose the operator named destroy-block
        with the destroyed-block the other block

apply*destroy-block*add-clear
        if there is an operator called destroy block,
        and the destroyed-block is block1,
        and block1 is ontop some other block,
        and no block is ontop block1
        then
        add clear block2
```

```
apply*destroy-block*change-texture
        if there is an operator named destroy block,
        and the destroyed block is block1,
        and block1 is ontop another block,
        and the density of the covering of the other block is
          hard or medium,
        then
        reject that covering density
        and add the density soft
```

8 A Soar Program

This section contains the entire Soar program that we have been constructing and shows how the parts fit together and what they do. This should allow you to see the "big picture" of a Soar program, and what the basic components are. Of course, this example has been very simple, and most Soar programs will be considerably larger than this. However, the fundamental components of a Soar program are all here.

The goals of this section are to:

- identify the basic components of a Soar program
- help the reader imagine what it would take to make his or her own Soar program

8.1 The components, and the program

All of the code that constitutes the example blocks world program is included below, with the exception of comments at the beginning of the file which identify the file and the authors, and a command at the end which alters the print format.

The minimum components of a Soar program are:

- a production which describes the initial state of the world, including working memory elements that describe the environment in which problem solving is to occur (Note: This initialization may not be necessary when the world description is being provided by an external environment, as described in section 10.)
- for each operator:
  - one or more productions to propose the operator (one in this case)
  - one or more productions to apply the operator (three in this case)
  - one or more productions to terminate the operator (one in this case)
- one or more productions to recognize when the desired state has been achieved

Most Soar programs also include a set of elaborations which reformulate knowledge. Actual elaboration productions will be shown in section 10.

The actual Soar program productions are included below. The boxes indicate actual parts of the Soar program. The comments from the actual code files have been replaced by descriptive paragraphs in the text. However, the actual code and comments can be found in the `$soar_library/tutorial/blocks.soar` file. The comments there detail clause by clause what is in each production. However, you should first try to understand each production just by interpreting it along with the accompanying textual description.

The first uncommented line of the program is not a production at all. Instead, it instructs the Soar architecture to choose at random among indifferent attribute preferences. When there are multiple choices for an attribute (including the operator) all of which have been specified *indifferent*, Soar can either select one of the possibilities, or ask the user which value it should use. (See the manual for a full discussion of this command.)

```
indifferent-selection -random
```

The first production creates the initial state. Soar automatically creates the WME object (usually `S1`) for this state during the "zeroth" decision cycle, and includes the fact that it has no superstate (^`superstate nil`). The following production fires after that WME is created and attaches the information about the "world" to it. It creates the structure that was shown in figure 2 (with the names changed to protect the innocent). This includes the 4 `things` (3 of them blocks), the `ontop` relations, and the `clear` relations. It also creates the substructure for the `things` and the `ontops`. Because they are not created by operator application, these additions are elaborations. However, because the initial state (which is the only condition) never goes away, the attributes and their values stay in memory until they are rejected by an operator application. The `thing`, `ontop`, and `clear` attributes all get the `parallel` (`&`) as well as the `acceptable` (`+`) preference so that there can be multiple values for the same attribute name.

```
sp {elaborate*initial-state
    (state <s> ^superstate nil)
    -->
    (<s> ^thing <block1> + &
         ^thing <block2> + &
         ^thing <block3> + &
         ^thing <T> + &
         ^ontop <O1> + &
         ^ontop <O2> + &
         ^ontop <O3> + &
         ^clear <block1> + &
         ^clear <block2> + &
         ^clear <block3> + &
         ^clear <T> + &)
    (<block1> ^type block ^name 1)
    (<block2> ^type block ^name 2)
    (<block3> ^type block ^name 3)
    (<T> ^type table ^name table)
    (<O1> ^top-block <block1> ^bottom-thing <T>)
    (<O2> ^top-block <block2> ^bottom-thing <T>)
    (<O3> ^top-block <block3> ^bottom-thing <T>)
    (write (crlf) |Initial state has 1, 2, and 3 on the table.|)}
```

The following production proposes all of the move-block operators. Remember that we want to be able to move a block to another block or a block to the table. This is easily accomplished because both blocks and tables are `things`. This production checks 1) if there are two things (the second different from the first), 2) if they are both clear, and 3) if the first thing is `ontop` of something besides the second thing. This last clause is necessary because the first block could be on the table which is always clear. We don't want to propose moving it to the table if it's already there, so we make sure the second thing is a block. The operator is proposed by giving it the `acceptable` and `indifferent` preferences, allowing Soar to choose randomly between the possibilities. The operator object is given a name, a `moving-block`, and a `destination`.

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

The next three productions perform the operator application. So why four separate operator application productions? Two reasons:

1. It is generally best in Soar to limit production actions to atomic events, i.e. a single action or a group of closely related actions.

2. In this specific case, there are four different actions which need to be done, and their conditions differ slightly from each other. They are:

   (a) Remove the statement that the `moving-block` is on whatever it was on before. This requires checking the state to see what the previous `ontop` relation was.

   (b) Add the statement that the block that was formerly under the `moving-block` is now clear. This is not done for the table, because of the convention that it is always clear.

   (c) Add the statement that the `moving-block` is on the `destination`. This only requires checking the operator, because it specifies both the `moving-block` and `destination`.

   (d) Remove the fact that the destination block is clear. This does not occur when the `destination` is the table.

The first operator application production, *apply*move-block*remove-old-ontop*, is the one that rejects the old `ontop` relation. It checks to see if the current operator is a `move-block`, and then checks the state to see what `ontop` relation is there with the moving-block as its top-block. Then it removes that ontop relation by giving it the reject (-) preference.

```
sp {apply*move-block*remove-old-ontop
    (state <s> ^operator <o>
               ^ontop <ontop>)
    (<o> ^name move-block
         ^moving-block <block1>
         ^destination <block2>)
    (<ontop> ^top-block <block1>
             ^bottom-thing { <> <block2> <block3> })
    -->
    (<s> ^ontop <ontop> -)}
```

*apply*move-block*add-new-clear* makes clear the block that the moving-block was ontop of prior to moving. The LHS conditions check the operator to see what block is moving, and check the state to see what the block was on. They must also check to make sure that that thing is a block (as opposed to the table). Then it adds a new clear relation to the state, giving it acceptable and parallel preferences.

```
sp {apply*move-block*add-new-clear
    (state <s> ^operator <o>
               ^ontop <ontop>)
    (<o> ^name move-block
         ^moving-block <block1>
         ^destination <thing2>)
    (<ontop> ^top-block <block1>
             ^bottom-thing { <> <thing2> <block3> })
    (<block3> ^type block)
    -->
    (<s> ^clear <block3> + &)}
```

*apply*move-block*add-new-ontop* adds the information that the moving-block is now ontop of the destination. It does this by simply checking the operator and its arguments. Then it adds a new ontop relation to the state with the moving-block as the top-block and the destination as the bottom-thing.

```
sp {apply*move-block*add-new-ontop
    (state <s> ^operator <o>)
    (<o> ^name move-block
         ^moving-block <block1>
         ^destination <thing2>)
    -->
    (<s> ^ontop <ontop> + &)
    (<ontop> ^top-block <block1>
             ^bottom-thing <thing2>)}
```

The fourth operator application production, *apply*move-block*remove-old-clear*, removes the information that the destination is clear. The conditions check to see if the destination was on the state as clear, and that it is a block, and not the table. Notice that it doesn't check the `moving-block` because that is irrelevant to the purposes of this production. Then it rejects that `clear` relation.

```
sp {apply*move-block*remove-old-clear
    (state <s> ^operator <o>
               ^clear <block2>)
    (<o> ^name move-block
         ^destination <block2>)
    (<block2> ^type block)
    -->
    (<s> ^clear <block2> -)}
```

*terminate*move-block* is responsible for determining that the operator has been completed and then terminating it by giving it the reconsider preference. It does this by checking the state to see if it has the `moving-block` `ontop` of the `destination`. Then it tells Soar to reconsider the operator. *Notice that in some slightly more benevolent world where some external action could stack the block before the operator is applied, this production would still fire. This is a good example of how Soar programs gain a lot by having the conditions of their productions be very specific to particular situations.*

```
sp {terminate*move-block
    (state <s> ^operator <o>
               ^ontop <ontop>)
    (<o> ^name move-block
         ^moving-block <thing1>
         ^destination <thing2>)
    (<ontop> ^top-block <thing1>
             ^bottom-thing <thing2>)
    -->
    (<s> ^operator <o> @)}
```

One final production wraps things up. It checks the state to see if there are three `ontop` relations: 1 on 2, 2 on 3, and 3 on the table. If so, it writes out a statement, and calls the halt function, returning control to the user.

```
sp {detect*goal
    (state <s> ^ontop <12>
               ^ontop <23>
               ^ontop <3T>)
    (<12> ^top-block <1>
          ^bottom-thing <2>)
    (<23> ^top-block <2>
          ^bottom-thing <3>)
    (<3T> ^top-block <3>
          ^bottom-thing <T>)
    (<1> ^name 1 ^type block)
    (<2> ^name 2 ^type block)
    (<3> ^name 3 ^type block)
    (<T> ^name table ^type table)
    -->
    (write (crlf) |Achieved 1, 2, 3|)
    (halt)}
```

8.2 Exercise: Running the Soar program

Start Soar, load the program, and set the watch level to an appropriate level, and step through the execution of the program, making sure that you understand at each step what is happening and why. See if you can predict which productions will fire next, and what changes they will make to preference memory.

9 Decision Cycle

In section 3, we described Soar's problem solving process as a sequence of operator proposals, selections, applications, and terminations. However, we didn't tell you any of the specifics of this process. In this section, we describe in broad terms the underlying algorithm that processes Soar programs. We'll see how and when Soar fires and retracts productions, resulting in changes to preference memory, and thereby to working memory.

By the end of this section, you should understand:

- the differences and relationships between production memory, preference memory, and working memory
- the basic operation of the decision cycle
- the basic operation of the elaboration cycle

9.1 The big picture

If you were a therapist analyzing Soar's behavior, you might say that Soar is obsessed with selecting and applying operators. This behavior is the basic (not the smallest) unit of execution in Soar. Soar measures its progress by selecting operators (as you can see in the traces), where each numbered step is an operator selection.

The process of applying the currently selected operator, making any associated changes, terminating that operator, and selecting a new operator is known as the *decision cycle.* The key boundary in this process is the point at which Soar selects an operator. This selection occurs in the *decision phase.* At this point, Soar compares all of its preferences for operators for the current state, and selects one to try next. After an operator is selected, Soar applies that operator. Once the applications have fired, in normal processing, a reconsider termination for the production is issued and a new operator is proposed.

Figure 12 depicts the decision cycle as a ladder which is built dynamically downwards from an initial state to a goal state.[14] Although it may seem strange to view a ladder as going downwards, it is shown this way because of the explicit similarities with a trace of Soar. (Try running Soar on the example at watch level one. Imagine the decision phases as the rungs on the ladder, and you will see what we mean.) Each rung on the ladder is a decision by Soar. At this point, that is, at the end of elaborations, Soar must be able to reconsider the current operator, and select an operator to work on next.

The spaces between the rungs on this ladder, labeled `Elaborate`, have a similar substructure, as shown in figure 13. These spaces are called *elaboration phases*, and

[14]For a different view of the decision cycle, see the decision cycle section in the manual.

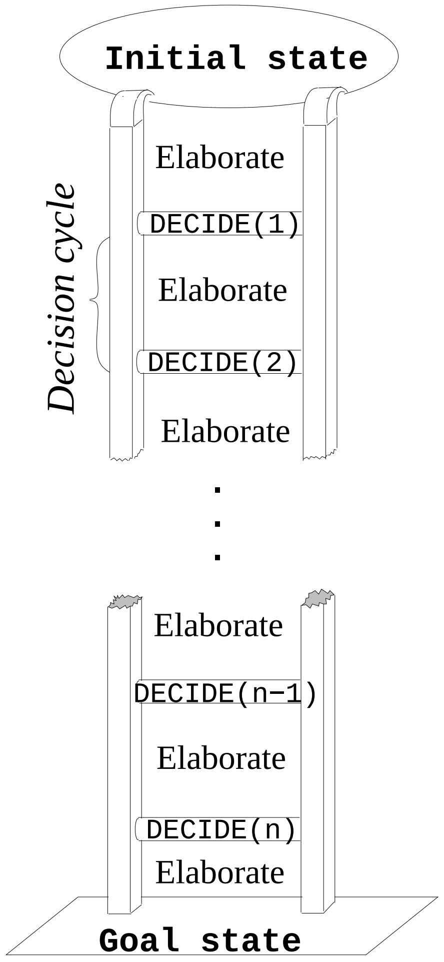


Figure 12: Decision cycles building a ladder

consists of one or more *elaboration cycles.* An elaboration cycle consists of a *preference phase*, and a *working memory phase.* During the preference phase, all of the productions that match newly created working memory elements are fired in parallel, adding preferences to preference memory. At the same time, any production instantiations whose conditions no longer match are retracted, resulting in deletions from preference memory of those preferences created by the instantiation. When all the productions have fired that matched originally, Soar makes another type of decision: it decides which working memory elements should be added or retracted — but only for non-operator attributes. These working memory decisions are the rungs on the elaboration ladder (labeled "Working Memory"). Of course, when additional elements are added to or removed from working memory, more productions may match, and others may need to be retracted. Thus, the elaboration phase continues until no more changes are made to non-operator attributes. When no new instantiations are created or retract, Soar has reached *quiescence.* At this point, the elaboration phase ends, and the decision cycle makes its operator decision.

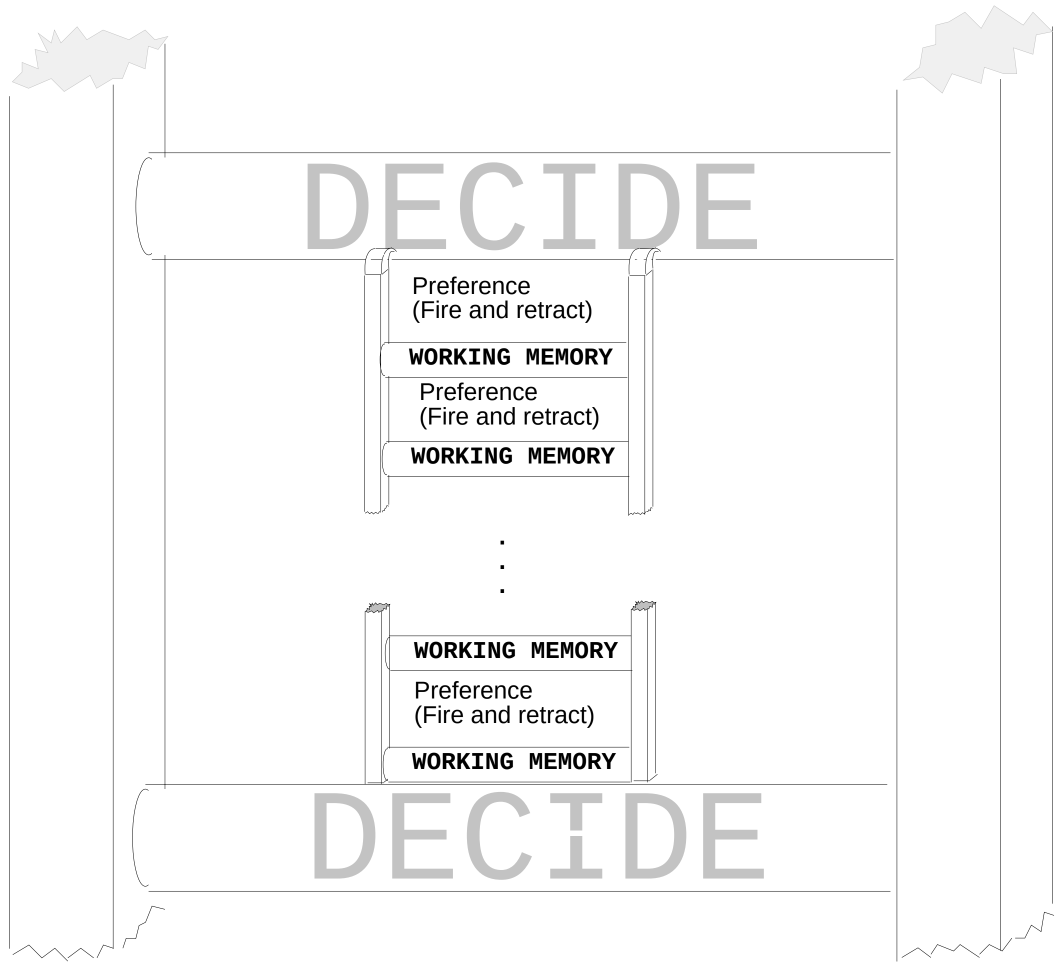


Figure 13: Elaboration cycles building a ladder between decisions

9.2 Memories and the decision cycle

It is very important to understand the relationship between the phases of the decision cycle and the three memories of Soar: production memory, preference memory, and working memory.

- During preference phases, productions (from production memory) are instantiated, the conditions bound to working memory elements. When productions fire, they add preferences to preference memory.
- When working memory elements have been removed, any productions that were based on those elements are retracted, and their preferences are retracted from preference memory. Retraction also occurs during the preference phase.
- During the working memory phase (which occurs after each preference phase), Soar collects non-operator preferences, and decides which elements to add to working memory, and which elements should be removed.

- During the decision phase (at quiescence, the end of the elaboration phase), Soar collects operator preferences, and attempts to select a new operator.

If at any point, Soar is unable to make a decision, an *impasse* is reached. Impasses and subgoaling will be described in lesson 13.

It is also important to remember why Soar uses preferences and parallel production firings, instead of making changes directly to working memory. The use of preferences allows Soar to reason about all of its decisions, even about which operator to choose in case of a tie. The preference mechanism allows Soar to fire all matched productions in parallel. This parallelism means that there is no order-dependence in Soar programs.

9.3 Running Soar

Start up Soar, load the blocks world example, and set the watch level to 1. Run 1 decision cycle. Now set the watch level to 5 and run a single elaboration cycle. At this level of detail, you can see all the Soar phases. You can ignore the input and output phases for now, although we'll return to them in the next section.

An elaboration cycle consists of a preference and working memory phase. In the preference phase, you can see productions firing and making changes to preference memory. In the working memory phase, you can see items being added to working memory (additions are indicated by `=>`, removals by `<=`). In this example, we see the same three production firings that we saw in section 5, only now the distinction between a preference memory change and a working memory change is more evident. The productions create an acceptable preference for a new WME (`S1 ^O10 +`) and issue rejection preferences for the `clear B2` and `ontop O3` augmentations of `S1`. In the working memory phase, these new preferences are assessed, resulting in the addition of (`S1 ^ontop O10`) and the removal of (`S1 ^ontop O3`) and (`S1 ^clear B2`).

```
Soar> init-soar                                   Restart Soar
Soar> run 1 d                                     Run one decision cycle
      0:  ==>S: S1
Initial state has 1, 2, and 3 on the table.
      1:  O: O7 (move-block 3 to 2)
Soar> watch 5
Soar> run 1 e
--- Input Phase ---
--- Preference Phase ---                          Productions fire in
Firing apply*move-block*remove-old-clear          the preference phase
 -->                                              and make changes to
 (S1 ^clear B2 - [O] )                            preference memory
Firing apply*move-block*remove-old-ontop
 -->
 (S1 ^ontop O3 - [O] )
Firing apply*move-block*add-new-ontop
 -->
 (S1 ^ontop O10 + [O] )
 (S1 ^ontop O10 & [O] )
 (O10 ^bottom-thing B2 + [O] )
 (O10 ^top-block B3 + [O] )
--- Working Memory Phase ---                      Working memory changes
=>WM: (56:  O10 ^bottom-thing B2)                 occur only in the
=>WM: (55:  O10 ^top-block B3)                    working memory phase.
=>WM: (54:  S1 ^ontop O10)                        => : addition to WM
<=WM: (28:  O3 ^bottom-thing T1)                  <= : removal from WM
<=WM: (27:  O3 ^top-block B3)
<=WM: (8:  S1 ^ontop O3)
<=WM: (13:  S1 ^clear B2)
--- Output Phase ---
Soar>
```

9.4 Exercise

Now type `init-soar`, set the watch level to 5. Step through the first decision, one elaboration cycle at a time. Pay special attention to the phases and what goes on during the phases. Look at the production firings, the working memory additions, and the selection of operators.

10 Interacting with an external world

In all of the previous sections, Soar was problem solving "in its head". Although our blocks world example could certainly refer to some actual physical environment, the Soar program only manipulated objects that were in working memory. Soar did not perceive an environment outside of its working memory and accomplished no external action. In this section, we will describe one way of connecting and using Soar with an external environment. Our example external environment will not be an actual physical environment, but rather a simulated blocks world implemented in another programming language.

In this new environment, the simulator will supply all of the information about the world that was previously changed by our Soar program. Thus, Soar will no longer need to define the initial (world) state, or add and remove information about the blocks while moving them. These tasks are done by the simulator. The new Soar program will just send a move-block command to the simulator. The simulator will, in turn, execute the command and update the simulator's world, leading finally to changes in Soar's working memory.

In order to get information from the simulator and send commands to the simulator, we will use Soar's *input* and *output links*. These are attached to the state via the `io` attribute, and can be thought of as the sensory and motor control areas of Soar.

The goals of this section are to:

- answer the question, "Why interact with an external world?"
- contrast external interaction with internal problem-solving
- introduce the blocks simulator

10.1 Why interact?

An external simulator can make some things easier for the Soar program. It can provide input about the state of the world so that the Soar program need not update it. It can also implement the operators so that the program doesn't have to calculate all of the changes made by the operators to the environment. However, the two most important reasons to choose external interaction over an internal simulation are:

- An intelligent agent can have some influence on an external environment, but not complete control. In an internal simulation, the agent does have complete control over its world state.
- External interaction allows a cognitive model to take perception and action into account.

Our initial version of the blocks world program represented the environment in Soar's working memory. Although the representation consisted of attributes such as `thing`, `ontop`, and `clear` – relationships possible in a physical world – they were completely under the control of the program. Productions could add and delete these attributes with no consideration of real-world constraints.

However, with an external world (even a simulated one), other forces can effect the state of the world. For example, something could move blocks other than the Soar-controlled robot. A gripper might not respond when told to move. The gripper's suction cup might fail. These potential problems force the underlying Soar program to pay attention to the ways that the world can change unexpectedly. Not just an issue for cognitive modelers, monitoring the external world is necessary for any agent that must behave robustly under potentially uncertain conditions.

Humans perceive an external world and act in it. In order to model such behavior in a psychological model, the model should also perceive what is in the world and take actions in it. These models then also include "submodels" of the operations that are involved in the activities of perception and action as well as cognitive processing.

10.2 The simulator

Soar 7, the latest version of Soar, is now compatible with Tcl/Tk [Ousterhout1994], greatly easing the development of graphical interfaces for Soar models. Figure 14 shows the interface for a blocks world simulator that we'll now use in this tutorial. The simulator contains a simulated gripper which travels around the world moving blocks with its suction cup grip. The blocks and gripper are shown in their initial state. Although this interface is only a simulation (as opposed to the real world), it is important to understand that the simulation is completely outside of Soar's direct control.

As mentioned above, the simulator makes a lot of things easier for our Soar program. First, Soar productions no longer need to initialize the initial world state. This initialization is now done through the simulator program, and it defines what blocks are in the world, and sends this state information to Soar. Similarly, when the Soar program applies an operator, it doesn't have to maintain the state by making the appropriate additions and removals. Instead, an application for the `move-block` operator now simply issues a move command to the simulator. The actual movement of the gripper is made by the simulator. Productions that terminate operators and recognize the goal state remain the same.

Commands are sent sent to the simulator in the *output phase*. Soar's state is updated by the simulator in the *input phase*. The input phase occurs in the elaboration cycle just before each preference phase, and the output phase occurs just after the working memory phase. Thus, each elaboration cycle consists of the following phases (in this order): input, preference, working memory and output.

For the task we've been using throughout the tutorial thus far, only a few changes to the production code are necessary for accomplishing the task through the simulator. We will require two additional productions: one to move information from the input

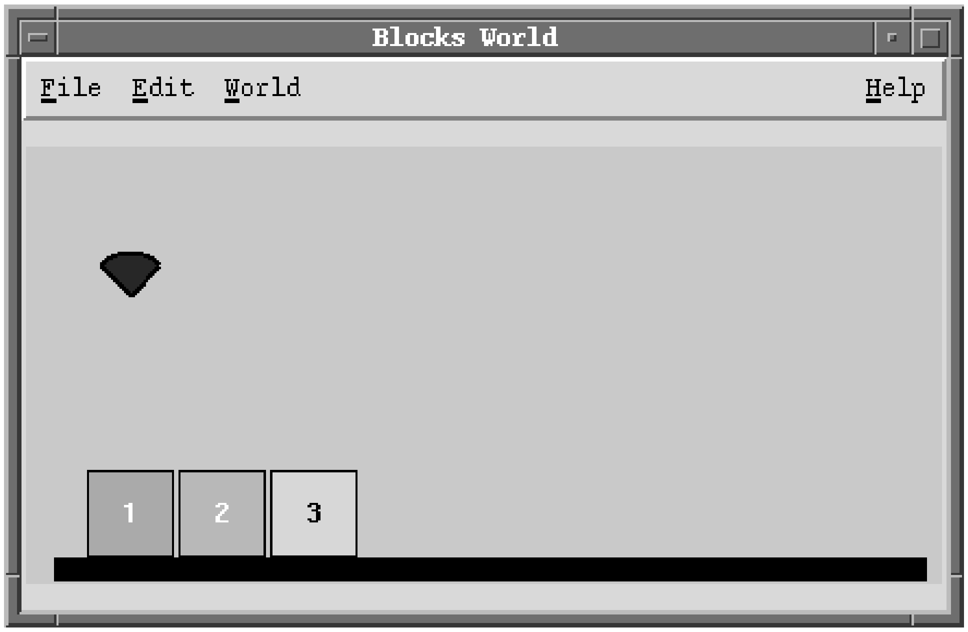


Figure 14: The blocks world simulator in initial state

link to the state, and the other to *create* the `output-link` and we have to change the applications of `move-block` to issue an output command. Because the code is similar to the `blocks.soar` productions, we will not include the actual productions here[15]. Examine the productions and notice:

- There is no initialization production.
- There is a new production which brings in knowledge from the `input-link` and puts it directly on the state. (We'll examine this production in detail below.)
- The operator proposal production is exactly the same.
- There is now only one operator application production. It puts the name of the operator, the `moving-block` and `destination` on the output-link.
- The operator termination production is exactly the same.
- The goal-state recognition production is exactly the same.

The production which copies information from the `input-link` to the state uses several features that we have not yet seen in previous examples. For ease of reference, that production is included here:

```
sp {move-input-to-state
    (state <s> ^io.input-link <il>)
    (<il> ^<att> <val>)
    -->
    (<s> ^<att> <val> + &)}
```

[15]You can can examine the productions in the file: `$soar_library/tutorial/blocks-external.soar`

In addition to using a variable for the value of the attribute (i.e., `<val>`), this production also uses a variable for the name of the attribute (`<att>`). Because this variable is not constrained by anything on LHS of the production other than the `input-link`, the `<att>` variable will match against every attribute on the input link. Each unique attribute and value leads to an instantiation which copies the value on the `input-link` to the state. Collectively, these instantiations copy every attribute on the input-link, and place the copies directly on the state. We could also avoid using this production by changing the rest of our productions to test the input values directly,[16] but we decided to change as few of the previous productions as possible.

Each augmentation copied by *move-input-to-state* `i-support`ed. When the `input-link` value or augmentation disappears (i.e. when the environment changes), the instantiation of *move-input-to-state* that matched against the changed feature will retract, leading to the removal of the copied feature on the state. Thus, the programmer need not worry about removing the copied augmentations when they no longer accurately reflect the state of the world.

This production also gives the copied augmentations both acceptable and parallel preferences. The parallel preferences are necessary because the `clear`, `ontop`, and `thing` augmentations can have multiple values. For instance, block `B1` and `B2` can both be clear simultaneously.

Move-input-to-state also introduces a new element of production syntax: dot notation. The condition:

```
    (state <s> ^io.input-link <il>)
```

could have been written equivalently as two conditions:

```
    (state <s> ^io <io>)
    (<io> ^input-link <il>)
```

Here, the variable `<io>` simply links the ^`io` augmentation to the ^`input-link` augmentation. Dot notation can be used whenever a variable is only created that link two augmentations. It simply makes the production a little simpler and more readable.

10.3 Running Soar: Basic external interaction

The load procedure is changed slightly. Move to Soar's location and start Soar as before. However, now we load the file: `blocks-external.soar`. This file not only contains the productions to build the 1-2-3 blocks tower, it also includes commands to load the simulator (but you not need understand any details of these commands).

Set the `watch` level to 5. This watch level will allow us to see decisions, phases, production firings, additions and deletions from working memory, and preferences. Now we tell Soar to run one phase: `run 1 p`. Previously, we have run Soar a decision and an elaboration at a time. Now we want to see what happens in individual phases.

[16]Most Soar systems test the `input-link`, rather than copy the input to the state.

```
unix% soar                                                  start Soar
Soar> source $soar_library/tutorial/blocks-external.soar
******                                                      Only six productions now!
Soar> watch 5                                               Set the watch level: Everything on
Soar> run 1 p                                               Run 1 Phase
=>WM: (2:  S1 ^superstate nil)                              Architecture creates these WMEs
=>WM: (1:  S1 ^type state)                                  at startup
 0:  ==>S: S1
--- Input Phase ---                                         First phase in elab. cycle. is input
=>WM: (3:  S1 ^io I1)                                       These WMEs are created by the
=>WM: (4:  I1 ^input-link I2)                               simulator and passed into
=>WM: (5:  I2 ^thing I3)                                    Soar via the input function.
=>WM: (6:  I3 ^type table)
=>WM: (7:  I3 ^name table)
=>WM: (8:  I3 ^width 9)
=>WM: (9:  I2 ^clear I3)
=>WM: (10:  I2 ^thing I4)
=>WM: (11:  I4 ^name 1)
=>WM: (12:  I4 ^type block)
=>WM: (13:  I2 ^thing I5)
=>WM: (14:  I5 ^name 2)
=>WM: (15:  I5 ^type block)
=>WM: (16:  I2 ^thing I6)
=>WM: (17:  I6 ^name 3)
=>WM: (18:  I6 ^type block)
=>WM: (19:  I4 ^color orange)
=>WM: (20:  I5 ^color violet)
=>WM: (21:  I6 ^color yellow)
=>WM: (22:  I2 ^clear I4)
=>WM: (23:  I2 ^ontop I7)
=>WM: (24:  I7 ^top-block I4)
=>WM: (25:  I7 ^bottom-thing I3)
=>WM: (26:  I2 ^clear I5)
=>WM: (27:  I2 ^ontop I8)
=>WM: (28:  I8 ^top-block I5)
=>WM: (29:  I8 ^bottom-thing I3)
=>WM: (30:  I2 ^clear I6)
=>WM: (31:  I2 ^ontop I9)
=>WM: (32:  I9 ^top-block I6)
=>WM: (33:  I9 ^bottom-thing I3)
=>WM: (34:  I2 ^holding nothing)
Soar> matches
Assertions:
 top*elaborate*state*io*output-link
 move-input-to-state (12)
Retractions:
```

After the creation of the initial state, Soar enters the input phase. During this phase, Soar talks to simulator program and the simulator sends Soar a description of the state of the world. The description takes the form of working memory elements and they are placed on the `input-link` to distinguish them as input.[17]

At this point, Soar has completed running one phase and stops. We can now use `matches` to see what productions will fire fire in the next phase, the preference phase. We see the production that will create the output-link and 12 instantiations of the *move-input-to-state* production. Closely examine the WMEs that were input. Determine why there are 12 instantiations of this production.[18]

Complete this elaboration cycle by typing `run 1 e`. Print state `S1`. You can see that input augmentations have been copied to the state. If you check the matches at this point, you will see that six instantiations of the *propose*move-block* production are ready to fire, as they did in the prior "in the head" example.

Complete this decision (`run 1 d`). Soar chooses an operator at random (as in in the previous example). The trace shows the choice of the `move-block` operator for moving block 3 to block 2 (`O6`). Note that the identifier has changed (in the old example, the identifier was `O7`). Soar creates a unique symbol for each identifier but these identifiers are arbitrary. Therefore, it's important to look at what the identifier represents, rather than to just look at an identifier itself, which may be different from one run to the next.

Now check the match set. You can see one production ready to fire, *move-block*apply*, rather than the three applications you saw before. `run 1 e` to complete this elaboration cycle. Notice that the result of *move-block*apply* is that a WME is created on the output-link (`O1`). When a WME is created on the output-link, the simulator is called to execute the command placed on the output-link. Although there is no notation in the trace that the simulator has been called, you will see the simulator move a block during this phase.[19]

If you examined Soar's input-link and the simulator state (i.e., the visual representation of the state), you would see a conflict at this point (this step is not shown in the trace). Although the simulator executed the output command, Soar has not yet accepted new input from the simulator so the old world state remains. Now run 1 phase. The simulator now sends Soar the changes to the state. For the move block 3 to block 2 operator, the information changes is as follows: the block 2 `clear` and block 3 `ontop` of table relations are removed from the input-link and the new `ontop`

[17]Soar input WMEs also have no preferences; unlike other WMEs, they are added directly to working memory by the input function. The lack of preferences is another distinguishing characteristic of input WMEs, although it is a distinction you will not need to worry about in the tutorial.

[18]*move-input-to-state* matches against `^io.input-link`. The identifier associated with the `input-link` is `I2`. Thus, to determine the number of instantiations, we need only count the number of WMEs that have `I2` has an identifier. There are 12: four `^things`, four `^clears`, three `^ontops`, and a WME fro the status of the gripper, `^holding nothing`.

[19]In this simple simulation, the Soar program waits for the simulator to completely execute the command before continuing. In most Soar models, the simulator executes a command in parallel while Soar continues running.

relation, block 3 `ontop` of block 2 are added to the input-link. Notice that the simulator has supplied these relations to Soar whereas in the old example, Soar's operator's had to make these changes themselves.

Continue running the example. Make sure you understand when input is updated, when the output is called, and how the productions accomplish the same task as in our previous example.

```
Soar> run 1 e
....                                        Lots of detail you can examine
Soar> matches                               Now check matches
Assertions:
  propose*move-block (6)                    6 instantiations (as before)
Retractions:                                for this production
Soar> run 1 d                               Finish this decision
....
 1:  O: O6 (move-block 3 to 2)              Identifiers have changed!
Soar> matches                               Now check matches
Assertions:
  apply*move-block                          Just one move-block application
Retractions:
Soar> run 1 e
--- Input Phase ---
--- Preference Phase ---
Firing apply*move-block                     Create preferences
 -->                                        for move-block command
 (O1 ^arg2 3 + [O] )
 (O1 ^arg1 2 + [O] )
 (O1 ^command move-block + [O] )
--- Working Memory Phase ---                Add to WM in WM Phase
=>WM: (75:  O1 ^arg2 3)
=>WM: (74:  O1 ^arg1 2)
=>WM: (73:  O1 ^command move-block)
--- Output Phase ---                        Output function (simulator)
                                            is called here because something
                                            was added to output-link (O1)
Soar> run 1 p
--- Input Phase ---                         Now input is updated
<=WM: (27:  I2 ^ontop I9)                   Remove clear block 2
<=WM: (30:  I2 ^clear I5)                   Remove 3 ontop table
=>WM: (76:  I2 ^ontop I10)                  Add 3 ontop 2
=>WM: (77:  I10 ^top-block I6)
=>WM: (78:  I10 ^bottom-thing I5)
Soar>
```

11 Writing a program, multiple-choice style

In this section, you will write your fist Soar program. The exercise will use "multiple-choice programming." We provide a file with a set of productions and you choose only those that are required for your task. With this multiple-choice approach:

- You will not need to type any productions.
- You will only need to recognize correct production syntax — not produce it.
- You can focus on two important parts of a Soar program: selecting and applying operators

The task Imagine that the simulated robot that was used in the external interaction blocks world task of section 10 was replaced by a different robot. This new robot doesn't know how to do `move-block` commands. Instead, it just knows how to pick up a block and how to put the block in its gripper down on some other block or the table. That is, the new robot takes `pick-up` and `put-down` commands. Each command takes one argument, for `pick-up`, the block to be picked up, and for `put-down`, the destination.

Your task is to write a Soar program to implement the blocks world in terms of `pick-up` and `put-down` operators. When designing Soar programs, you should think first in terms of the higher-level concepts described in the beginning of this tutorial: selection and application of operators. Operators are the key object that you should think about. In this new task, there will be two operators, `pick-up` and `put-down`. For each operator, you should consider when it should be *selected*, and how it should be *applied*. Remember the three parts of operator selection: proposal, comparison, and the actual selection by Soar. Because we are currently implementing a random approach to solving the blocks world task, comparison knowledge is not necessary. Additionally, because the Soar architecture selects among proposed operators, you only need a `proposal` production to implement the higher-level selection function.

Application has two parts: 1) productions which actually perform the operator action, and 2) productions which terminate the operator. Because we will be using the external environment, the Soar program need only put a command (pick-up or put-down) on the `output-link` to perform the desired action. The For termination production should check to see if the results of the action one of the operator's initiated (by sending the output command) now exist on the state.

Remember: think in terms of selection and application of operators. Getting a bit more concrete, think of proposal, comparison and selection, and then application and termination of operators. You task is to now recognize these conceptual functions in productions.

How to do it The file `pick-put.soar` in the tutorial directory includes many productions, a subset of which will accomplish the task. Copy the file to your own directory, and comment the productions you think are unnecessary. Refer to other example programs to better understand the productions in `pick-put.soar`. Comment the productions that you don't want using `#` at the beginning of each line of the unwanted production. Now load your program and test. If it doesn't work properly, try again.

A complete solution to this exercise is located in file pick-put-soln.soar, but try not to look at it unless you are really really stuck. Good luck!

12 Adding intelligence: MEA

This is where things start to get really fun. In the previous sections, we've described a program which solved a problem, but it did it in a rather stupid way. It knew that it wanted to get the blocks stacked in a certain way, and it knew how to move a block, but it didn't know anything about when it should move a particular block to a particular location. In Soar-ese, we say that it lacked operator selection knowledge.

In this section, we show how such knowledge can be added to this simple blocks world task. We will use a technique called Means-Ends Analysis (MEA) which attempts do decrease the distance between the current state and the desired state. We'll do this by adding rules which specifically compare operators. The new rules will specify:

- Prefer operators that put a block into its desired position with respect to the destination. Do this by giving appropriate operators a `best` preference.
- If you have one operator proposed that puts block A on block B, and one that puts block B on block C, then prefer the second over the first. Do this with the binary `better` preference.

Another thing that must be added to the program is to explicitly put the goal information on the state. Recall that in the previous version of the program, the goal only existed in a production which recognized the goal, printed a notice, and halted Soar. But in order to be able to explicitly reason about the goal, the goal information must be available. And part of the theory of Soar says that production memory is not accessible. So the first production of the MEA addition must explicitly put the goal on the state. The code (with comments) is in file `$soar_library/tutorial/blocks-mea.soar`, but the productions are included below for your convenience.

The first production is shown below. It puts the goal information on the state by first "accessing" the objects which refer to the blocks 1, 2, and 3, and the table. Then it creates an augmentation on the state called `desired-state`. To this, the production attaches three `ontop` relations, which specify block 1 on top of block 2, block 2 on top of block 3, and block 3 on top of the table.

```
sp {blocks-world*put-goal-on-state
    (state <s> ^superstate nil
               ^thing <block1>
               ^thing <block2>
               ^thing <block3>
               ^thing <table>)
    (<block1> ^type block ^name 1)
    (<block2> ^type block ^name 2)
    (<block3> ^type block ^name 3)
    (<table> ^type table ^name table)
    -->
    (<s> ^desired-state <ds> +)
    (<ds> ^ontop <ontop1> + &
          ^ontop <ontop2> + &
          ^ontop <ontop3> + &)
    (<ontop1> ^top-block <block1>
              ^bottom-thing <block2>)
    (<ontop2> ^top-block <block2>
              ^bottom-thing <block3>)
    (<ontop3> ^top-block <block3>
              ^bottom-thing <table>)}
```

The second production gives selection control information. It specifies that if a proposed operator will move a block on top of another block, and if that `ontop` relation is in the desired state, then it should be given a `best` preference. This (along with the acceptable preference that it already has) means that Soar will select this operator before any other proposed operators which do not have a best preference, unless another preference is specified as better than this one.

```
sp {select*move-block*prefer-into-place
    (state <s> ^operator <o> +
               ^desired-state <d>)
    (<d> ^ontop <ontop>)
    (<ontop> ^top-block <tb>
             ^bottom-thing <bb>)
    (<o> ^name move-block
         ^moving-block <tb>
         ^destination <bb>)
    -->
    (<s> ^operator <o> >)}
```

Important note: There is an interesting syntactic item in this production. The first condition clause checks the operator augmentation of the state and includes the acceptable preference. This is only *allowed for operators, because they are treated differently than other augmentations. Whenever an operator gets an acceptable preference, it is put on the state with an indicator of the preference. When an operator gets* selected *it is added to the state without the preference. Take, for example, the first state we looked at:*

```
(S1 ^type state
    ^thing I3 ^thing I2 ^thing I4 ^thing T1
    ^ontop I7 ^ontop I6 ^ontop I5
    ^operator O4
    ^operator O1 + ^operator O2 + ^operator O3 +
    ^operator O4 + ^operator O5 + ^operator O6 +)
    ...
```

*The line which just says `^operator O4` indicates that `O4` is the selected operator. The proposed operators show up on the state with their acceptable preferences. This is necessary in order for Soar to reason about proposed operators. For any other augmentation that has multiple values and does not specify the **`indifferent`** preference, Soar would just choose one value (depending on the **`user-select`** setting) and put it the state.*

The third production goes one step further than the `best` preference, specifying that one production is better than another. A `better` preference of A over B will cause Soar to select the A before B, even if B has a `best` preference and B doesn't. This allows relative knowledge between two items to take precedence over knowledge about a single item.

The conditions of this production check if there are two move-block operators, and two corresponding `ontop` relations on the desired state which specify that A should be on top of B and B should be on top of C. If so, the operator which moves B to C is preferred over the one which moves A to B.

Note the nature of this bit of control knowledge. The previous production specified that if some operator creates part of the desired state, then it should be preferred. This knowledge is very general and could be applied to almost any task. But the knowledge in the second production is specific to tower-building. It is based on the world knowledge that the base of a tower must be built before higher sections can be added on. The level of specification of operators and the separation of operator selection from proposal, application, and termination allows Soar programs to specify this type of problem-solving knowledge.

```
sp {select*move-block*prefer-lower
    (state <s> ^operator <o1> +
               ^operator { <> <o1> <o2>} +
               ^desired-state <d>)
    (<d> ^ontop <ontop1>
                <ontop2>)
    (<ontop1> ^top-block <mb1>
              ^bottom-thing <mb2>)
    (<ontop2> ^top-block <mb2>
              ^bottom-thing <mb3>)
    (<o1> ^name move-block
          ^moving-block <mb1>
          ^destination <mb2>)
    (<o2> ^name move-block
          ^moving-block <mb2>
          ^destination <mb3>)
    -->
    (<s> ^operator <o2> > <o1>)}
```

12.1 Exercise

For this exercise, combine means-ends analysis with external interaction. Hint: it won't take much.

13 Subgoaling

Up until this point, we have painted a simple, rosy picture of problem solving with Soar. The program provided all of the knowledge about what could be done, and just had to go about doing it until the problem was solved. But in real life, Soar's task is seldom so simple. Consider these three scenarios:

1. You are working with a slightly less powerful simulator in the blocks world task, one that doesn't know commands at the level of `move-block`, but rather at the level of `pick-up`, and `put-down`. You still want to solve the problem in a hierarchical manner, however, invoking a `move-block` operator, which in turn invokes the lower level operators.
2. Your model of human reading behavior generally works at the word level, reasoning about combinations of nouns, verbs, etc. But because of some ambiguity in the input (e.g. a letter is smudged), you need to start problem solving at the letter level in order to disambiguate a word.
3. Your model of navigating from one place to another knows that one mode of transportation (e.g. bus, plane, taxi) will not allow it to get all the way from the current location to its destination. It could start by taking the bus or the taxi, but doesn't know which to choose.

All of these scenarios lead to a condition where Soar does not know what to do next. This is called an *impasse*. Soar always does the same thing when it reaches an impasse — it *subgoals*, that is, it creates a new state which includes the type of impasse and a link to the original state. The goal for processing in this new state is to resolve the impasse. Soar uses subgoaling to shift its focus from solving the original task to solving a particular aspect of that task. This is a key attribute of Soar's problem solving: Whenever it gets stuck, Soar creates a new state which it expects to provide the solution to the impasse.

This section describes:

- the high-level view of subgoaling
- what a problem space is, and how it is used to describe Soar's problem solving
- the definition of impasse and subgoal
- examples of different types of subgoaling
- uses and effects of multiple levels of subgoaling
- the relationships between subgoaling and persistence

13.1 Subgoaling and the PSCM

Because subgoaling is such an important aspect of Soar's operation, we will take this opportunity to introduce some advanced concepts related to the theoretical description of Soar. We normally describe Soar at three different levels, the Problem Space Computational Model (PSCM), the Symbol Level Computational Model (SLCM), and the implementation level. So far in this tutorial, we have been concentrating at the two lower levels, the SLCM and the implementation level. We previously only talked about Soar at the PSCM level — where Soar is described just in terms of states (including initial and desired states) and selection and application of operators — in lesson 2. At the SLCM level, we talk about the proposal, comparison, selection, application, and termination of operators, and the different memories of Soar. At the implementation level, we talk about the actual productions which make up Soar programs. Understanding these different levels is especially important when designing Soar systems, because it allows the designers to concentrate at the appropriate level of description at the appropriate times in the design process. At the beginning of the design process, one should start by thinking of what the states describe and what the operators are — not by thinking of the productions which implement them. In the rest of this section, we will show what the PSCM view of Soar is, and how it relates to subgoaling.

13.1.1 Basic problem spaces

The term "problem space" is used at two levels in Soar. We'll talk about the conceptual level now, and leave the implementation-level discussion of it for later in this lesson. We talk of problem solving occuring in Soar within a problem space. The problem space is the conceptual space in which problem solving is pursued, starting at the initial state, progressing to other states via application of allowable operators, and arriving (hopefully) at a desired state.

Soar people often depict a problem space as a triangle (which represents a tree on its side) as shown in figure 15. In the left side of the triangle is the initial state, the circle with `IS` in it. The arrows are operator applications which transform one state to other states, denoted with `Si`. On the right are desired states, of which there may be one or more (or even none!). The states except for the initial state are drawn with a dashed line to indicate that they are not necessarily known in advance. Application of operators will lead to new states from the initial state, but there is not an enumeration of the possible states. The desired states are described, but not explicitly listed. For example, some specific attribute of a goal state might be specified, but the other attributes might be unspecified. Thus we think of a Soar program navigating through this space by selecting and applying operators, but we don't think of it as searching through this space, because there is no backtracking mechanism.

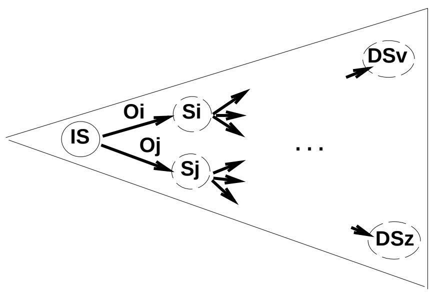


Figure 15: A conceptual problem space, with initial and desired states

13.1.2 Problem spaces and subgoaling

Up to this point in the tutorial, we have only discussed tasks which use one problem space. But it may be more natural for some tasks to use two or more problem spaces within which problem solving will occur. For example, in the reading example, we might have two problem spaces: `word-level` and `letter-level`. At the word level, the state would have descriptions of the current word of interest, and the ones preceding it, and the letters of the current word which have already been read. At the letter level, the state would describe the various features of the current letter. Problem solving at the word level would proceed by adding the next letters in the input to the word, one at a time. If a letter can not be read, the word level would not know how to process it, and the focus would shift to the letter level.

Figure 16 gives a depiction of this two problem space approach to the reading task. Problem solving proceeds in the upper, word-level problem space until it can not progress any further — an impasse[20]. At this point, Soar creates a subgoal and problem solving continues in the new problem space until the impasse is resolved. An impasse is resolved when something changes (usually as a result of processing in the subgoal) which allows Soar to progress at that level.

13.2 Symbol Level (SLCM) view of subgoaling

Now we'll complicate things slightly by shifting to a different view of subgoaling. At the PSCM level, we talked about problem solving in terms of states and operators, and about subgoaling as creating a new space of states in which different operators may apply. At the SLCM level, we talk about problem solving in terms of proposing, selecting, applying and terminating operators, and about the decision cycle. Thus it is more natural here to look at the relationship of subgoaling to the decision cycle ladder (figure 12) that we saw in lesson 9. That view showed the decision cycle as building a ladder down from the initial state to a goal state. Figure 17 shows how subgoaling fits into the decision cycle. As this ladder is build, an impasse is encountered. In order to resolve the impasse, Soar creates a subgoal and commences problem solving to

[20]Impasses, subgoaling, and impasse resolution are defined in symbol-level terms below.

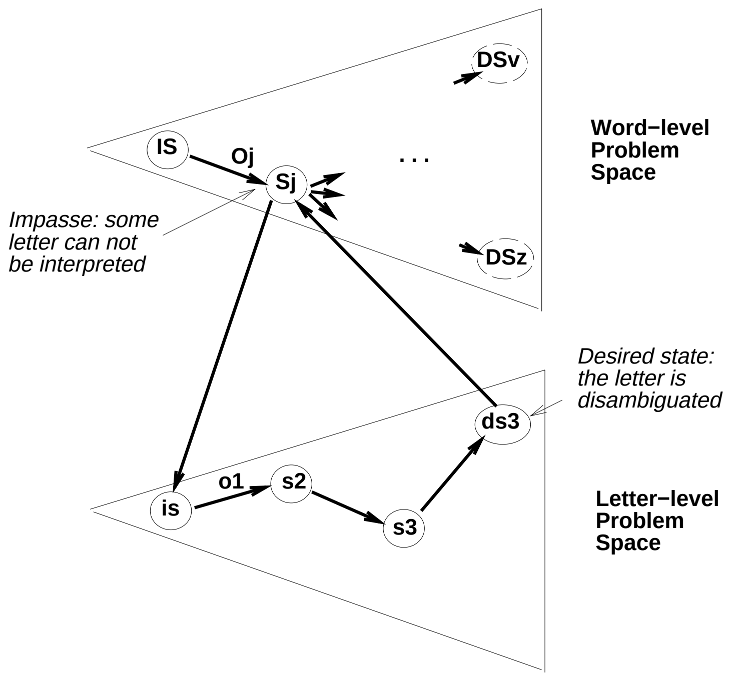

DSv
Word−level Problem Space
IS
Oj
Sj
. . .
Impasse: some letter can not be interpreted
DSz
Desired state: the letter is disambiguated
ds3
o1
s2
is
s3
Letter−level Problem Space

Figure 16: A conceptual problem space with an impasse and subgoal

resolve the impasse, in effect creating a new ladder with the implicit goal of allowing processing to continue on the original ladder.

At this time, it's appropriate to make a more precise definition of some of these terms:

- **Impasse:** A point at which Soar is unable to proceed in its problem solving. This can happen for a variety of reasons, most of which are related to the inability to select a unique operator at the end of a decision cycle.[21] As described in lesson 9, Soar tries to choose an operator in every decision cycle. A number of situations can interfere with that process, including those which come from preference conflicts. If the current operator is not reconsidered, an `operator no-change` impasse is triggered. If multiple operators are proposed, but none is selected, an `operator tie` impasse occurs. If no operator is proposed, then a `state no-change` impasse occurs.

- **Subgoaling:** Whenever an impasse is reached, Soar automatically creates a new state. It contains a description of the impasse plus the state in which the impasse was reached. The goal of problem solving in the new state (which is not necessarily explicit) is to resolve the impasse. In Soar, this process is *automatic* and *universal*. It is automatic in that it is an architectural feature of Soar. The program doesn't deliberately invoke subgoals, the architecture starts a subgoal

[21]There are also impasses caused by difficulties with assigning values to attributes, but these are less important and will not be addressed in the Coloring Book. For a complete list of impasse types, see the Soar Manual.

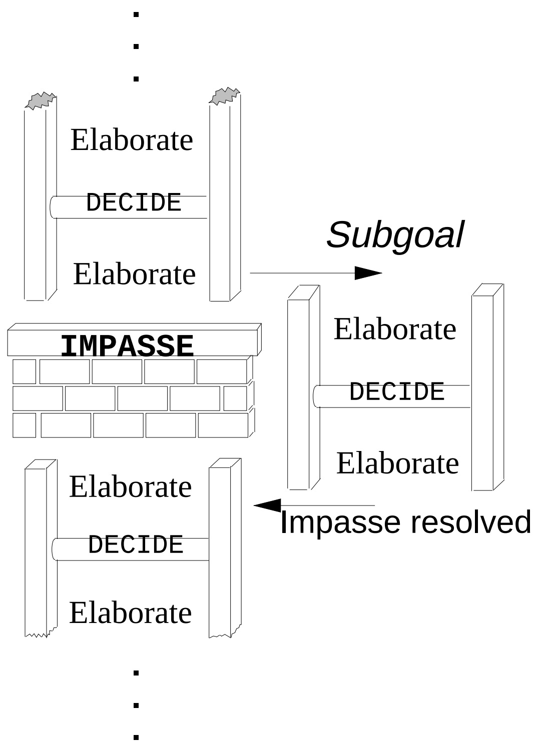


Figure 17: Subgoaling builds sub-ladders

when an impasse occurs. It is universal because it happens whenever any aspect of the problem solving process can not continue.

The conditions described above for creating a subgoal are simple: when Soar can not make progress in its problem solving, that is an impasse, and Soar creates a subgoal. The rule for removing subgoals is equally simple: when an impasse is resolved, all subgoals (there can be more than one, as described later in this lesson) below that impasse are removed. Because the subgoals are often thought of as a stack (built deeper and deeper under the top-state in an upside-down gravity world), the process of removing subgoals is often described as “popping the subgoals” off of the stack.

Figure 18 shows before-and-after scene. On the right, there is a situation with four states, the top-state and three subgoals under it. In the top three, there has been an impasse, so the subgoals underneath have been created. Eventually impasse I2 is resolved (normally as a result of one of the subgoals, but it could also be due to some other change in the environment that has allowed Soar to progress in that problem space).[22] At this point the lower subgoals are removed from the subgoal stack, and problem solving continues.

[22]Note that the act of returning a result is not the same as returning from a subgoal in a traditional programming language. The lower state is not returning control to the upper one, it is just adding information to the upper state.

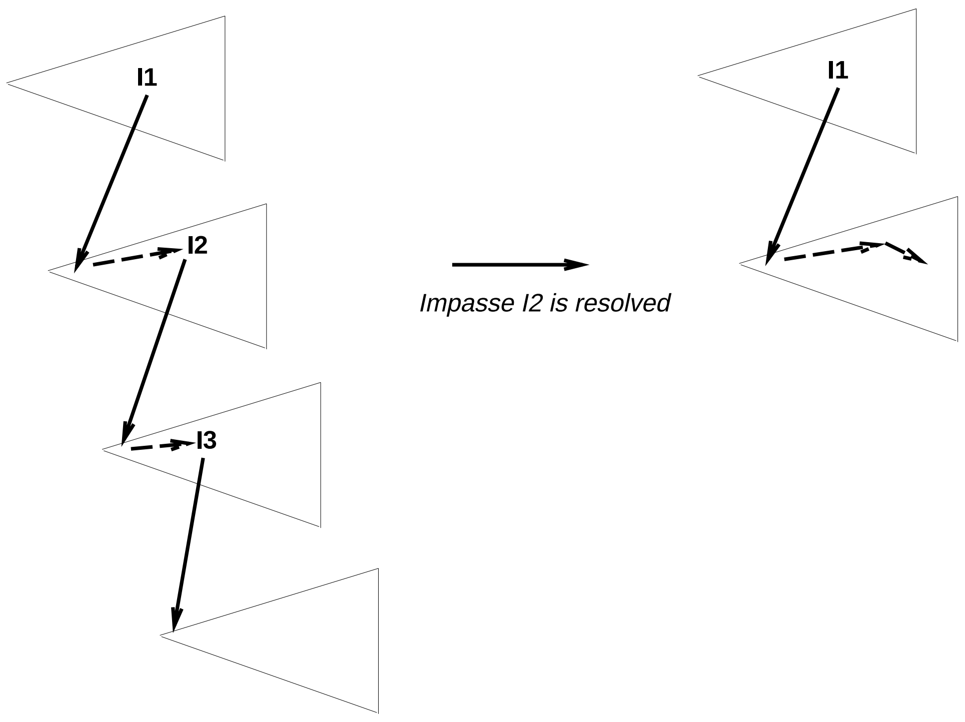


Figure 18: When an impasse is resolved, subgoals beneath it are popped off the stack.

As an example of an SLCM-level description of a task, let's look a bit deeper at the reading example. As mentioned before, at the word level, the state describes the current word which is being built up, and the preceding word, as well as the letter that is currently being "examined". The operators at this level could be something like `add-a-letter` which would add the current letter to the current word, and `complete-word` which would, if it saw a space or some punctuation mark as the current character, signal the current word as complete so that it could be operated on by higher level operators. The proposal conditions for `add-a-letter` would be that there is a current letter which is an alphabetic character. `Complete-word` would be proposed if the current character was punctuation. An obvious reason for an impasse would be if there were no current letter, which could be caused by some sort of sensing failure — Soar can not "make out" the current letter. At this point, there would be no proposed operator, so Soar would reach a `state-no-change` impasse.

As a result of the impasse, Soar would create a subgoal, which would describe the impasse, and have the word-level state as the superstate. In this level, other operators, like `look-for-an-E`, could apply that would try to interpret the features of the current letter as combining to make a certain character. Perhaps there would be an operator for each character that was proposed as indifferent with the others. They could return preferences for interpreting the current letter as "their own" character to the superstate, which would allow processing at the word-level to continue.

13.3 Subgoaling and the implementation level

In this section, we will describe the way that Soar operates at the level of actual productions. For the most part, this will consist of examples of specific Soar programs. We will start, however, with a discussion of the implementation of problem spaces.

13.3.1 Problem spaces at the implementation level

At the implementation level of Soar, we use the term problem space in a very specific way. In the reading example with two conceptual problem spaces, we said that the focus shifts from the word level to the letter level when progress can not be made at the word level. The mechanism for doing this is via the problem-space attribute. In Soar programs, we use a programming convention in which we attach a `problem-space` attribute to the state, and give it a name. Most productions then specify a value for the `problem-space.name` as a condition on their left-hand side. This is a very convenient method of partitioning the production set, i.e. separating the rules based on which problem space they apply in. Thus in the reading example, because we don't want the productions that apply at the word level firing in the letter level, and vice versa, we would add conditions to the productions like:

```
(state <s> ^problem-space.name word-level)
```

or:

```
(state <s> ^problem-space.name letter-level)
```

13.3.2 Implementation subgoaling

The next few sections describe some of the standard ways that subgoaling is used in Soar. We will start with `implementation subgoaling`, in which a subgoal is used to implement a higher level operator. One example mentioned at the beginning of this section was the hierarchical blocks world, where the top-level operator is still `move-block` (as in previous examples), but this is implemented by lower level `pick-up` and `put-down` operators.

The standard way of doing this is to have the same type of proposal and termination productions that we have seen before. For example:

```
### Propose an operator to put thing1 on thing2 if both are clear
### if thing1 is not ontop of thing2.
sp {propose*move-block
    (state <s> ^thing <thing1>
               ^thing { <thing2> <> <thing1> }
               ^clear <thing1>
               ^clear <thing2>
               ^ontop <ontop>)
    (<ontop> ^top-block <thing1>
             ^bottom-thing <> <thing2>)
    -->
    (<s> ^operator <o> + =)
    (<o> ^name move-block
         ^moving-block <thing1>
         ^destination <thing2>)}


### Terminate the move-block operator for thing1 and thing2 if
### thing1 is ontop of thing2.
sp {terminate*move-block
    (state <s> ^operator <o>
               ^ontop <ontop>)
    (<o> ^name move-block
         ^moving-block <thing1>
         ^destination <thing2>)
    (<ontop> ^top-block <thing1>
             ^bottom-thing <thing2>)
    -->
    (<s> ^operator <o> @)}
```

The difference is that there are no rules that apply the move-block operator. Thus, when the `move-block` operator is selected, Soar is not able to make any progress on the operator, and an `operator-no-change` impasse occurs, causing Soar to make a new substate of the current state. Whenever it makes a new state, Soar creates certain standard attributes which identify the type of impasse and link the state to its superstate. For this impasse, the following state would be automatically generated (assuming that the original state was `S1`):

```
(S2 ^attribute operator
    ^choices none
    ^impasse no-change
    ^quiescence t
    ^superstate S1
    ^type state)
```

The `attribute` and and `impasse` augmentations of this state identify it as caused by an `operator-no-change`. The `superstate` augmentation links it to the superstate `S1`. This superstate link is very important. By following the superstate link, we can find out which operator ran into trouble. That is, the Soar program can examine the ^`superstate.operator.name` to determine what action should be taken in the subgoal. In this example, we first use the information on this new state to propose a new problem space. Let's call it `gripper`. The following production creates the problem space and names it, when the operator no-change impasse has occurred and the super-operator is `move-block`:

```
sp {(state <s> ^impasse no-change
               ^attribute operator
               ^superstate.operator.name move-block)
    -->
    (<s> ^problem-space <p>)
    (<p> ^name gripper)}
```

As mentioned above, we then use the problem space name to act as an additional condition for a new set of rules. The following rule proposes the `pick-up` operator. It checks if the problem space name is `gripper`, and if the operator of the superstate is `move-block`, and if the block that is to be moved is not being held (indicated by the absence of the `holding` attribute on the state). If these conditions hold, the `pick-up` operator is proposed.

```
sp {(state <s> ^problem-space.name gripper
               ^superstate.operator <superop>
               -^holding <block1>)
    (<superop> ^name move-block
               ^moving-block <block1>)
    -->
    (<s> ^operator <o> +)
    (<o> ^name pick-up
         ^block <block1>)}
```

The application of the `pick-up` operator adds the fact that the block is being held to the state. The proposal for the `put-down` operator checks that the problem space name is `gripper` and that the super-operator is `move-block` for the specified block. If that block is currently being held, the `put-down` operator is proposed, and the application of it moves the block ontop of the `destination-thing`. At this point, there will be a new `ontop` relation between the `moving-block` and the `destination-thing`. This allows the termination production for the `move-block` super-operator to fire, and consequently to resolve the impasse. At this point, the subgoal is removed, and problem solving can go on in the original problem space.

13.3.3 Exercise: Implementing an implementation subgoal

1. From the description above, describe the changes that would be needed to convert the simple blocks code to perform implementation subgoaling using the `pick-up` and `put-down` operators in the gripper problem space.[23]

[23]The actual process of creating this program will be completed in the next lesson.

2. Implementation subgoaling (or any subgoaling) can work at multiple levels. For example, it is possible given a slightly different problem representation that the `pick-up` and `put-down` operators are not directly implementable. Consider the external interaction world where the gripper must be moved around. If the simulator does not handle commands at the level of `pick-up` (which requires knowledge about moving in relation to objects) but only handles commands like `move-up`, `move-down`, `move-left`, and `move-right`, then an additional level of subgoaling can be used to implement the `pick-up` and `put-down` operators. Design such a program, specifying what descriptions of the world are on the state, what the proposal and termination conditions for each operator are, where the subgoaling occurs, and what the implementations for the operators are.

13.3.4 Resolving an operator tie impasse

Scenario number 3 from the beginning of the chapter described a situation in which a transportation advisor system knew that both bus and a taxi were suitable modes for getting to the desired location, but didn't have the information (in that problem space) to decide which to choose. Let's look a bit closer at this situation.

In the top-level problem space of this task, the planner is examining the distances that should be travelled and suggesting suitable transportation modes. The program could have rules like:

- If the distance from the current location to the desired destination is less than 1 kilometer, then walk.
- If the distance is between 1 km and 10 km, then take a bus.
- If the distance is between 5 km and 15 km, then take a taxi.
- If the distance is between 15 km and 200 km, then take a train.
- If the distance is greater than 200 km, then you should fly.

These rules are implemented by productions which give an acceptable preference to an operator, for example, `take-the-bus`. If the desired distance is 500 km, then only one operator will be proposed, and the program can continue. If the distance is 7 km, then two operators will be proposed by this set of rules, `take-the-bus` and `take-a-taxi`, resulting in an operator-tie impasse, where Soar does not know how to proceed.

In order to resolve this ambiguity, you would probably want your Soar program to take into account other factors than just distance — your money on hand, for example, or the amount of luggage you have to carry. Note that your program could avoid the impasse in a number of ways. You could add more knowledge to the productions which propose the operators, changing the taxi rule, for example, to specify:

- If the distance is between 1 km and 10 km, and you don't have much money, then take a bus.

But this method makes the rule overspecific, so you would have to design multiple mutually exclusive rules like:

- If the distance is between 1 km and 5 km, then take a bus.
- If the distance is between 5 km and 10 km, and you have very little money, and a lot of time, then take a bus.
- If the distance is between 5 km and 10 km, and you have enough money, and very little to carry, and a lot of time, then take the bus.
- and so on

Another way to avoid the impasse would be to include explicit operator comparison rules, using desirability preferences. You could add rules like the following to the top problem space to give preferences to one operator or another, allowing Soar to choose between operators that would otherwise be tied.

- If you have one `suggest-travel(taxi)` operator and another `suggest-travel(bus)` operator, and you have very little money and no luggage, then add a preference that the bus operator is better than the taxi operator.
- If you have one `suggest-travel(taxi)` operator and another `suggest-travel(bus)` operator, and you have plenty of money, prefer the taxi operator over the bus operator.

This method is a reasonably efficient solution to the problem, but has one conceptual difficulty: At one point (when proposing operators), the program is reasoning about distances alone. When comparing the proposed operators, the program is reasoning about amount of money on hand, amount of stuff to carry, and amount of time, as well as the distance to travel. It is conceptually cleaner (and may be actually more efficient if the knowledge is sufficiently complex) to separate the different types of knowledge into separate problem spaces. This is the solution that we will pursue in the rest of this section, with an example program that counts the amount of money on hand in order to resolve an operator tie impasse.

The operator tie impasse Let us start with a Soar program with the original set of distance-based operator proposal rules as in the example file travel.soar. This program has been asked to solve a problem where the distance is 7 km. At this point there will be two operators proposed, `take-a-taxi` and `take-the-bus`. Because there are two operators each of which has an acceptable preference, Soar can not uniquely select an operator, a tie impasse results, and a subgoal is created. Next, the `name-compare-operators-ps` production fires to name the new problem space `compare-operators`. The productions which count the money all check that `compare-operators` is the current problem space, so they will not apply in the top space.

Comparing operators and returning a preference As mentioned above, the productions which bring additional information to bear to resolve an operator tie will only apply in the `compare-operators` problem space. In this space, productions which count the money apply and then use this information to choose between the different transportation modes. To start this process, the `init-money` operator is used to initialize the sum of money to 0.

After the money-sum is initialized, the operators to add the particular bills (ones, fives, etc.) from the money structure on the top-state are proposed and made indifferent. The operators include the new amount that they should put in place of the money-sum, and the old value of the money sum. When the operators are applied, they reject the old value for money-sum, and make an acceptable preference for the new value. The operator applications also place a flag on the state that that amount has been added so that the operators don't get proposed again. When the money-sum is equal to the new sum, then the operators are terminated.

At the same time that the `add-ones`, etc. operators are proposed, the `finished-counting` operator is also proposed, but it is given a worst preference so that it will fire after the others have finished. It puts a `finished-counting` flag on the state so that the actual comparison operators can fire only after all of the money has been counted up.

The productions, `compare-bus-and-taxi*prefer-taxi` and `compare-bus-and-taxi*prefer-bus`, put a desirability preference (`better`, in this case) for one operator over the other on the top state.[24] Because the inability to choose between the two operators was the original cause for the impasse, and because this preference allows Soar to choose one operator, the impasse is solved, and the subgoal is popped off the stack.

13.4 Multiple levels of subgoaling

It is often the case that there are multiple levels of subgoals. This adds very little additional complexity to Soar. Although it is usually the case that because supergoals have encountered an impasse, problem solving can not proceed in them, they are still active in the sense that as soon as the impasse is resolved, problem solving will continue. As mentioned above, as soon as an impasse is resolved, the substates below it are popped of the stack, no matter how many there are, or what their status is. We will leave it as a set of exercises for the reader to see how multiple levels of subgoaling interact. Section 13.3.3 asked for a multi-level operator implementation subgoal;

[24]This is known as "returning a result", and is used for chunking purposes as described in the next section.

there is another multi-level exercise below; and the next lesson includes an example of chunking that involves two levels of subgoaling, in order to implement a lookahead search. *Warning! The exercise below has been deemed by the Surgeon General of the United States to be potentially hazardous to your health. It is very difficult. The exercise in lesson 14 and the example in section 15.3 are closely related, and are a bit more "safe". It may be wise to do that exercise first (after finishing this lesson) and look over that section and then come back to this exercise.*

13.4.1 Exercise: Multiple levels

Imagine a system which goes one step further than the travel example and actually simulates the traveling by changing the location somehow, and decrementing the amount of money. Assume that the rate for the bus is 1 dollar/km, and the rate for the taxi is 3 dollars/km. In order to check if there is enough money in this scheme for a particular mode of transportation, the Soar program should implement the comparison as a lookahead search.[25]

This exercise is a bit complicated, so we'll make it a combined design and running Soar exercise. The idea of the exercise is for the program to create a subgoal of evaluating each of the tied operators. It does this by proposing an operator called evaluate-operator for each tied item with that item itself as an argument. The evaluate-operator operators are made indifferent so that Soar can choose freely among them. The idea is that Soar will try out the different tied possibilities to see which leaves the traveller with a non-negative money-sum.

The second level of subgoaling comes in because we want to try out each operator in its own search space with its own copy of the important information so that trying out one operator doesn't change the money left for trying out the other operator. A second level of subgoaling can be forced by not having operator application productions for the evaluate-operator operators, leading to an operator no-change impasse.

In the resulting subgoal, another production can assign a problem-space name, for example, `evaluate`. Additional productions should copy the important information, money-sum and required-distance from the superstate and the topstate. Then a production can propose performing the operator that was being evaluated in the superstate.

Another requirement of this approach is application and termination rules for the operators. The application productions should reduce the money-sum and put a flag on the state saying that the travel by that transportation mode has been completed. The termination productions can check for that flag.

After the travel has been completed, we will want to evaluate that operator to see if it has left the traveller with any money. This involves a different operator which can be given a worst preference so that it only fires after the transportation lookahead is complete. The application of this operator will be to check the amount of money

[25]Lookahead search is a standard Soar mechanism that will be seen again in lessons 15 and 16.

on hand, and if it is less than zero, put a failure flag on the corresponding evaluate-operator from the superstate. If the money-sum is zero or above, a success flag should be put on the superstate's evaluate operator.

One more pair of productions is needed to pass on the results of the search to the topstate and resolve the operator tie impasse. In the compare-operators problem space, these productions will check to see if there is a success or failure indicator on the current operator, and if so, pass a best or worst preference to the corresponding operator in the top space.

The exercise The preceding description should give you a pretty good idea for what the design of this lookahead program should be, and the productions that will be required. From this description, create a Soar program to implement the lookahead search. Note: the productions from the previous example which do the comparison based on the money-sum should be removed for the lookahead scheme to work. There is an working example of this exercise in the file `travel-lookahead.soar`, but try to avoid referring to it until you actually have tried to write the program.

13.5 Subgoaling, Persistence, and Justifications

Section 6 described the distinction between persistent preferences and elaborations. In short, those preferences (and the WME's that they support) which are created by operator application productions stay in preference memory until they are explicitly rejected by another production. All other preferences are called elaborations and remain only as long as the conditions which created them still hold. This story must be changed slightly when subgoaling is taken into account.

As we have seen, it is possible for a subgoal to create a preference for an attribute for its supergoal ("returning a result"). If that preference is created by operator application in that subgoal, we expect that it will have O-support, and therefore will persist. In this case however, if the subgoal is eliminated (because Soar can proceed in the supergoal), then that O-support is also retracted. (The preference might still be O-supported from a higher-level goal as described below.) *Important point: O-support for a preference is contingent on the existence of the state in which the support was created.*[26]

On the other hand, just because a subgoal goes away, we don't want Soar to eliminate the results of that subgoal, because they generally provide a change to the state that allows Soar to continue past an impasse. So Soar must have a mechanism for remembering that there is support for a result even after the subgoal which created

[26]The reason for this is related to chunking and so will become clearer after the next section. For now, let us just say that a chunk is created when a subgoal returns a result, and often avoids the creation of the subgoal in future processing. The chunk traces the support back to the supergoal. If the result only got O-support from an operator in the subgoal, then the chunk would not give O-support, and therefore Soar would act differently before and after chunking. This probably doesn't make sense now, but if you come back to it after the chunking section, it might.

it goes away. This mechanism is called a *justification*.[27] Justifications record the WME's in the superstate that support the result. Figure 19 shows the superstate, and a WME that is tested in the substate. From the substate, a result is created for the superstate. When the result is created, Soar traces back through the record of support to find the origin of the support in the superstate. You can think of this as building a bridge from the initial WME to the result. When the subgoal is popped off the stack, the supporting justification remains.

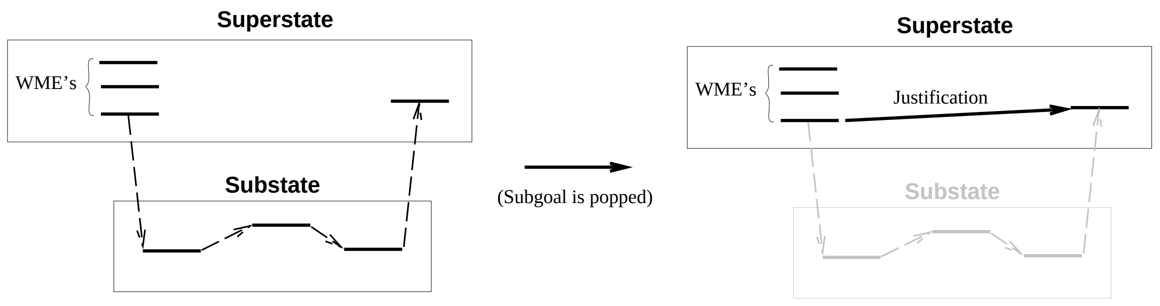


Figure 19: A justification forms a bridge of support

To see a simple demonstration of justifications, turn learning off (`learn off`), type in the following productions, and run two decision cycles with watch 3 on. You will see that the `bar` augmentation still exists on the state after the subgoal is popped off the stack. If you use the preferences command, `preferences s1 bar 1`, it will show that the support for this augmentation comes from the justification. Print the justification to see what it looks like.

```
sp {foo
    (state <s> ^superstate nil)
    -->
    (<s> ^foo foo)}

sp {bar
    (state <s> ^superstate <ss>
               ^impasse no-change)
    (<ss> ^foo <foo>)
    -->
    (<ss> ^bar <foo>)}

sp {baz
    (state <s> ^bar <b>)
    -->
    (<s> ^operator <o>)}
```

[27]A justification is completely equivalent to a chunk, except that a justification goes away as soon as the preference which it supports is rejected. A chunk becomes part of long-term memory, and can fire as a production in the future. Chunks, and the exact mechanism of backtracing to produce them and justifications, are described in the next chapter.

The productions below create an O-supported ^baz augmentation on the top-state. The O-support remains after the subgoal is popped because it comes from the superstate's operator.

```
sp {foo
    (state <s> ^superstate nil)
    -->
    (<s> ^operator <o>)
    (<o> ^name foo)}

sp {bar
    (state <s> ^superstate <ss>)
    (<ss> ^operator.name foo)
    -->
    (<s> ^bar bar)}

sp {baz
    (state <s> ^superstate <ss> ^bar <b>)
    -->
    (<ss> ^baz <b>)}

sp {bob
    (state <s> ^baz <b> ^operator <o>)
    -->
    (<s> ^operator <o> @)}
```

13.5.1 Exercise

Create a demonstration of the “Important point” above by using the techniques of the `foo`, `bar`, `baz`, and `bob` productions to create a result on the top-state that is o-supported from the subgoal, but which loses its o-support when the subgoal goes away.

14 Writing another program: Comments to code

In lesson 11, we created a program which issued `pick-up` and `put-down` commands to the blocks-world simulator to accomplish the simple blocks task. In this exercise, we will modify that approach to be a more natural one for Soar, using what we just covered about subgoaling. This exercise will require a more detailed knowledge of Soar syntax, because you will write the productions which accomplish the task based on comments which describe those productions.

The goals of this exercise are to:

- enforce the fundamentals of subgoaling
- teach the basics of writing productions:
  - matching and linking on the left hand side
  - creating preferences on the right hand side
- create another Soar program
- teach programming-by-copying

The task for this exercise is the same as the previous task except that you will use a hierarchy of operators: At the top level, the `move-block` operator is selected. This is implemented in a subgoal using `pick-up` and `put-down` operators.

To do the exercise, copy the file pick-put-subgoal.soar into your own directory. Refer to the previous pick-put exercise and any other example programs to help you write the productions. Feel free to copy parts of productions, but it is better if you type in the details rather than cutting and pasting. Look at the comments in the file to find out what should be included in the productions. The words which are in all uppercase letters are intended to be literals that should be typed as is (except lowercase) in your program.

When you have written the program, start Soar and source the program file. It will load the simulator. Test out the program. If it doesn't work, try to figure out why. Remember to examine the key parts of the program. When is an operator proposed? Is it applied by sending the command to the simulator? Is the operator terminated?

A complete solution to this exercise is located in file pick-put-subg-soln.soar, but try not to look at it unless you are really really stuck. Good luck!

15 Learning in Soar: Chunking

One of the most important features of Soar is its ability to learn. Clearly humans learn routinely, while performing other cognitive tasks. Soar's learning mechanism is also seamlessly integrated into its processing, and is invoked whenever Soar subgoals during problem solving, as was described in the previous section.

The goals of this section are to describe:

- chunking, Soar's learning mechanism
- the purpose of chunking
- the basic processes of chunking
- the art of getting chunking to work for you
- how chunking can go awry
- the differences between chunking and speed-up learning

15.1 What chunking does, and when

The technique of chunking is closely tied to that of subgoaling, which was described in the previous section. To recap, Soar subgoals when it encounters an impasse, that is, a situation in which it does not know what to do next. Conceptually, this is thought of as Soar not being able to do automatic processing and having to resort to some sort of deliberative decision making for what to do next. This deliberative process is computationally expensive, requiring creation of additional states and switching contexts, so we want to avoid it whenever possible. Chunking is the process of learning what was done to resolve the impasse to enable processing to continue in the original problem space. A chunk is an actual production that is created which should apply in the situation which caused the subgoal (and others like it[28]), and whose actions produce exactly the changes to preference memory that were done while resolving the impasse. The chunk allows Soar to slide past the impasse (an impasse is never reached) as shown in figure 20.

For example, if Soar has a number of operators that are proposed for state `S7`, but has no mechanism for selecting one of the operators in that problem space, an operator tie impasse is reached. At this point, a subgoal (`S11`, we'll say) is created by the Soar architecture which describes the impasse. The impasse is resolved when some production creates one or more preferences which resolve the tie, either by eliminating all but one choice, or by giving a desirability preference (better, best, worse, or worst) to one or more operators to break the tie. These preferences are changes made to the

[28]Generality is achieved by replacing identifiers with variables, as described in the next section.

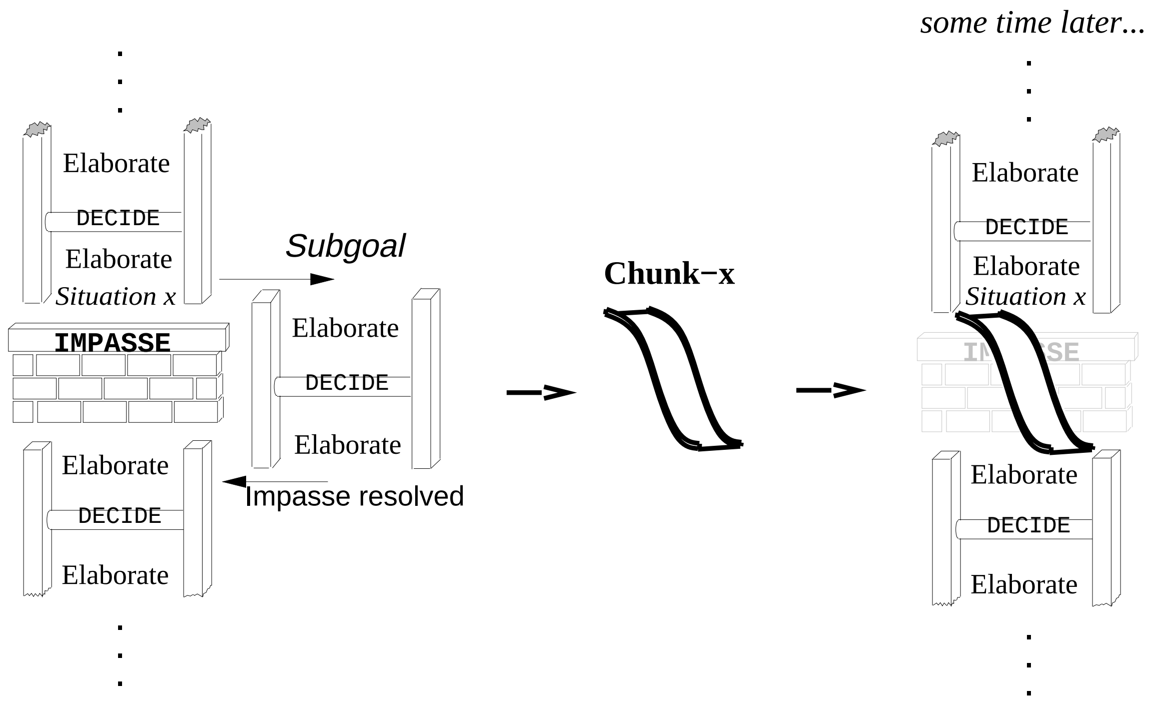


Figure 20: Chunks avoid impasses: At some point in processing, *situation x* is reached, and subgoaling invoked. When the impasse is resolved, `chunk-x` is built. Later, *situation x* is reached again, and chunk-x fires, avoiding the impasse.

superstate[29], to `S7` in this case. In Soar, a *result* is any change made from processing in a substate to preferences in a superstate. In this example, the desirability or reject preferences made in the substate are results. Whenever a result is created (and when learning is not turned off), Soar makes a chunk that describes the conditions under which result should be made, and what the result is. The method for creating chunks is described in the next subsection.

15.2 How chunks are made

As we described in the last section, a chunk is made when a result is passed to the superstate (if learning is turned on). The process which creates a chunk is a bit complicated, so we won't describe it in detail, but we will give you the basic idea.

There are three parts to chunk creation:

- Backtracing
- Variablization

[29]In this section, "superstate" refers to the state in which an impasse is reached, and subgoaling initiated. "Substate" will refer to the state which is architecturally created to handle the impasse, and includes a `superstate` link to the superstate.

- Reordering of conditions

15.2.1 Backtracing

A chunk is a production that produces the effects of a result that occurred during subgoaling. Thus the conditions of a chunk should include exactly those attributes of the superstate which were relevant to returning the result. The actions of a chunk should be exactly the preferences that were created as the result. Obviously, the result is not so difficult to determine, it is merely the preference that is created in the superstate (for example, a best preference for an operator that was previously tied). The conditions are more difficult to compute, and require *backtracing*.

Backtracing starts with the WMEs that matched the condition clauses of the instantiated production which produced a result. These WMEs are attributes that are present on the substate when the result is returned. But remember that what we want for the conditions of the chunk are the relevant attributes from the *superstate* that led to the result. So if these WMEs are on the superstate, then they will become part of the chunk's conditions. If they are not, the backtracing process recursively finds the conditions that created these WMEs, until it find those that are part of the superstate.

Conceptually, this is the reverse of the process that created the WMEs. Productions fire when their conditions match memory, and their actions add preferences to preference memory, from which Soar puts WMEs in working memory. Then these new WMEs can match other productions. During backtracing, Soar checks the WMEs which matched the conditions of a production which returned a result. If those WMEs are not part of the superstate, it checks the preferences which support those WMEs, and the productions that created the preferences. This process continues until the WMEs are traced back to elements of the superstate. Figure 21 illustrates this process.

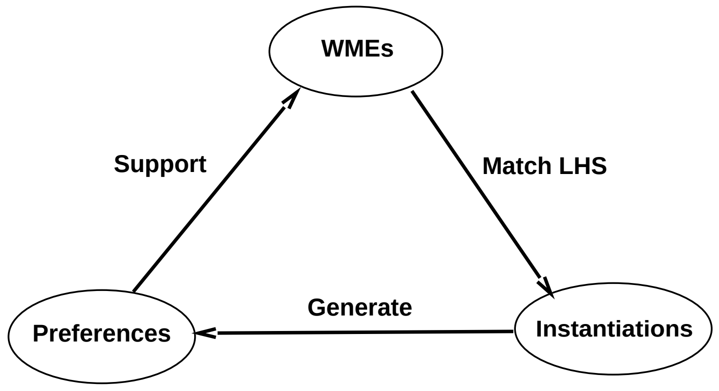


Figure 21: The arrows in forward direction show how things actually happen in time. In the reverse direction, they show how backtracing happens. (Figure by B. Doorenbos.)

Another way of thinking about the subgoaling process is depicted in figure 22. In this diagram, the states are shown as boxes, and the solid dark lines represent WMEs. At time 1, the substate is created when an impasse occurs, and attributes `a`, `b`, and `d` from the superstate are copied down by some production. WME `x` is not linked to the superstate. At time 2,[30] a production matches WMEs `a` and `x`, and creates WMEs `f` and `g` changing the state as depicted in Substate(2). At time 3, another production fires, creating WME `h`. At time 4, a production fires which causes WME `i` to be added to the superstate — a *result*. At this point, the backtracing process is started, following the productions backward to find the WMEs that were matched which are linked to the state.

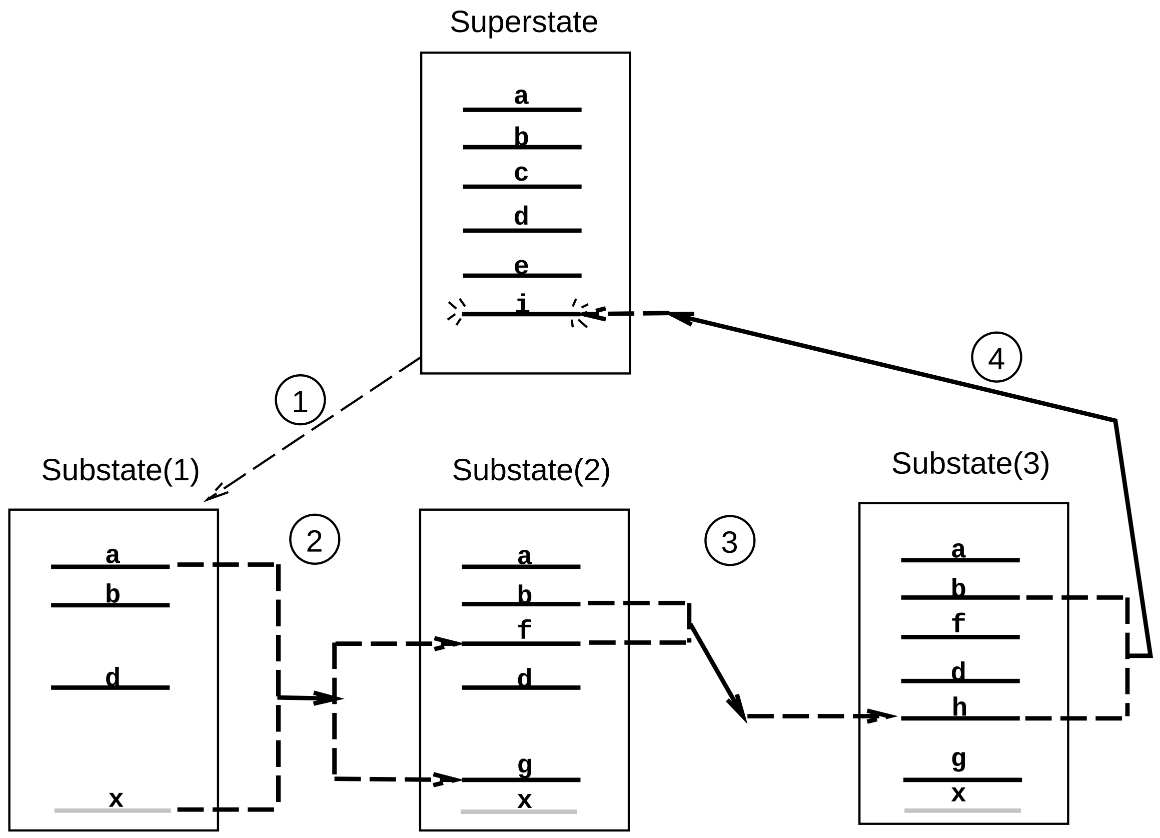


Figure 22: A subgoal in more detail

Figure 23, illustrates the backtracing process. When Soar subgoals, it stores a trace of the productions which fire, the WMEs that they matched, and the WMEs that they produce support for. This allows the backtracing procedure to examine those productions to see where results came from. At step 5 in the diagram, backtrace starts from the result and examines the WMEs that matched the production which created it, `b` and `h`. It knows that `b` is linked to the superstate, so it puts it on its list of grounded WMEs (the list that will become the condition of the newly-formed

[30]We have made the times distinct here for demonstration purposes. Remember that in Soar, productions fire in parallel.

chunk). It does not know about WME `h`, so in step 6, it checks to see where `h` came from. It was supported by WMEs `b` and `f`. Backtrace already knows about `b`, so it doesn't worry any more about that, but it must check `f`. In step 7, backtrace finds that `f` was supported by WME `a`, another grounded WME. It was also supported by `x`, but that is local to the substate, so it is not returned. Backtrace thus returns the two WMEs `a` and `b` as the conditions for the new chunk, which basically says, "If `a` and `b` exist, then create `i`."

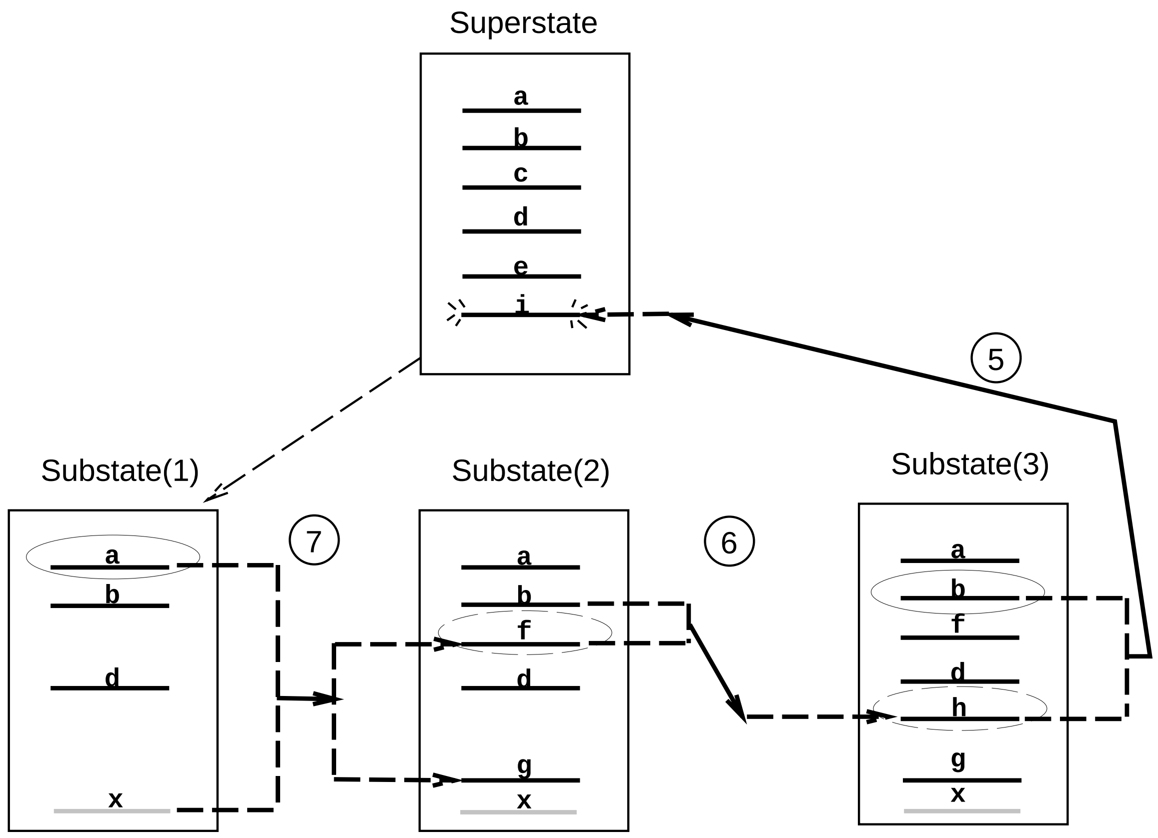


Figure 23: Reverse the process to backtrace

It's about time we introduced an actual example so that you can see how this works. To start with, we use a simple extension of the blocks world with Means Ends Analysis as described in section 12. The difference in this example is that the indifferent preferences from the operators have been removed, causing an operator tie impasse in the top state. A production has been added to name the problem space of the subgoal when it occurs to be the choose-operator problem space. Finally, the MEA productions have been altered to only apply in the choose-operator problem space.

As before, the MEA productions check the desired state to determine what operator should be done at the current time. Now when they provide the better or best preference to an operator, that is a result and triggers the chunking process. Load the program as shown below and run it to see what happens.

Start Soar, and source the file blocks-chunk-mea.soar. Now run the program with the d command, and examine the trace (for your convenience it is shown below). You will see that at step 1, Soar subgoaled, creating state S2, with an operator tie impasse. In resolving the impasse, two chunks were built. (Chunk-2 was ignored because it was the same as chunk-1.) At step 2, processing has reverted to the top state (check the the indentation of the trace), and the move block operator to move 2 to 3 is selected. During the next decision cycle, chunk-1 and chunk-3 fire, and no subgoal is created. At the end of the decision cycle, the operator to move 1 to 2 is selected, and that operator completes the task.

```
soar> source \$soar\_library/tutorial/blocks-chunk-mea.soar
************
soar> d

     0: ==>S: S1
Initial state has 1, 2, and 3 on the table.
     1:    ==>S: S2 (operator tie)
Building chunk-1
Firing chunk-1
Building chunk-2
Ignoring chunk-2 because it is a duplicate of chunk-1
Building chunk-3
     2:    O: O5 (move-block 2 to 3)
Retracting chunk-1
Retracting chunk-3

     3:    O: O8 (move-block 1 to 2)
Achieved 1, 2, 3
System halted.
```

If you look carefully at the chunks that were built (printed below), and the underlying MEA productions, you will notice something striking — the chunks are exactly the same as those productions, with the exception of changed variable names. This situation is addressed in section 15.3.

```
soar> print chunk-1
(sp chunk-1
  :chunk
  (state <s1> ^operator <o1> + ^desired-state <d1>)
  (<o1> ^name move-block ^destination <d2> ^moving-block <m1>)
  (<d1> ^ontop <o2>)
  (<o2> ^bottom-thing <d2> ^top-block <m1>)
-->
  (<s1> ^operator <o1> >))


soar> print chunk-3
(sp chunk-3
  :chunk
  (state <s1> ^operator <o1> + ^operator { <o2> <> <o1> } +
            ^desired-state <d3>)
        (<o1> ^name move-block ^moving-block <d1> ^destination <d2>)
        (<o2> ^name move-block ^destination <d1> ^moving-block <m1>)
        (<d3> ^ontop <o3> ^ontop <o4>)
        (<o3> ^bottom-thing <d2> ^top-block <d1>)
        (<o4> ^bottom-thing <d1> ^top-block <m1>)
      -->
        (<s1> ^operator <o1> > <o2>))
```

In order to see the backtracing as it happens, use the `init-soar` command to initialize Soar, use the `excise -chunks` command to get rid of the chunks that were created, and turn backtracing watch on with the command `watch -backtracing on`. Now run the program again (one decision cycle, or elaboration cycle at a time if you wish) to see Soar print out the results of the backtracing procedure. It lists several types of WMEs. `Grounds` are the WMEs that are definitely linked to the state. `Locals` are those that are definitely local. `Potentials` are the WMEs that are known to not be local, but which are not yet known to be grounded. It is not necessary to immediately understand all of what is printed out in this trace, but the essential parts are the WMEs that get pulled out to become the conditions of the new chunks.

15.2.2 Variablization

Variablization is the process by which the identifiers in the supporting WMEs (which have been returned by backtracing) are replaced by variables. Variablization is not a complicated process, but it is an extremely important one. It is through this process that Soar gets generality in what it learns. Without variablization, a chunk would only apply in exactly the same situation in which it was created — and that is very unlikely to happen.

Examine the trace below (or run it on your own), using the `explain` command to show how the chunk was variablized from the grounded WMEs.

```
soar> explain -on
Explain turned on
soar> source \$soar\_library/tutorial/blocks-chunk-mea.soar
************soar> d 1

     0: ==>S: S1
Initial state has 1, 2, and 3 on the table.
     1:    ==>S: S2 (operator tie)

soar> d 1

Building chunk-1
Firing chunk-1
Building chunk-2
Ignoring chunk-2 because it is a duplicate of chunk-1
Building chunk-3
     2:    O: O5 (move-block 2 to 3)

soar> explain chunk-1
(sp chunk-1
   (state <s1> ^operator <o2> +)
   (<o2> ^name move-block ^destination <i2> ^moving-block <i1>)
   (<s1> ^desired-state <d1>)
   (<d1> ^ontop <o1>)
   (<o1> ^bottom-thing <i2> ^top-block <i1>)
-->
   (<s2> ^operator <o3> >))

1 :  (state <s1> ^operator <o2> +)  Ground : (S1 ^operator O8 +)
2 :  (<o2> ^name move-block)        Ground : (O8 ^name move-block)
3 :  (<s1> ^desired-state <d1>)     Ground : (S1 ^desired-state D1)
4 :  (<o2> ^destination <i2>)       Ground : (O8 ^destination I3)
5 :  (<d1> ^ontop <o1>)             Ground : (D1 ^ontop O1)
6 :  (<o1> ^bottom-thing <i2>)      Ground : (O1 ^bottom-thing I3)
7 :  (<o2> ^moving-block <i1>)      Ground : (O8 ^moving-block I2)
8 :  (<o1> ^top-block <i1>)         Ground : (O1 ^top-block I2)
soar>
```

In the table at the end of the trace above, Soar lists each condition of the newly created chunk, and the grounded WME that it was created from. Notice that the identifiers in the WMEs were replaced with variables in the conditions. For example, `O8` was replaced with `<o2>` in each condition clause.

The important aspects of variablization are that:

- The same identifiers are replaced by the same variable, forcing those variables to match the same identifiers in working memory in the future.
- If there is an explicit inequality check in a production (e.g. `<x> <> <y>`), then

there are two variables in the chunk, with the explicit inequality test between them.

- Overspecific chunks can be created if two different variable coincidentally refer to the same identifier. This could occur if, for example, the block that was being moved was also the last block that moved, and some production trivially tested the last moved block.[31] In this case, a chunk would be built that would fire only when this situation occurred again.

15.2.3 Reordering

Reordering of conditions is a process that Soar does routinely with all productions, not just chunks. Based on an analysis of which condition clauses can most effectively rule out a production from firing, the productions clauses are sorted. Thus if you ever ask Soar to display the match of a certain production, the conditions shown are probably not in the order in which they were written. This is primarily an issue that affects Soar's speed, so we will not deal with it further in this tutorial.

15.3 A more serious chunking example

The previous section showed a very simple example of chunking, in which Soar learns productions which give it Means-Ends Analysis rules for solving the blocks world program. If we compare the chunks which were learned, however, with the underlying MEA rules, we can see that there is no difference between them, except for their variable names.

```
SOAR PRODUCTION                           LEARNED CHUNK
---------------                           -------------
                                          sp {chunk-1
sp {select*move-block*prefer-into-place       :chunk
    (state <s> ^operator <o> +                (state <s1> ^operator <o1> +
               ^desired-state <d>)                            ^desired-state <d1>)
    (<d> ^ontop <ontop>)                      (<d1> ^ontop <o2>)
    (<ontop> ^top-block <tb>                  (<o2> ^top-block <m1>
             ^bottom-thing <bb>)                        ^bottom-thing <d2>)
    (<o> ^name move-block                     (<o1> ^name move-block
         ^moving-block <tb>                             ^moving-block <m1>
         ^destination <bb>)                             ^destination <d2>)
    -->                                       -->
    (<s> ^operator <o> >)}                    (<s1> ^operator <o1> >)}
```

The reason that this happened is because in the subgoal, exactly two productions fired — the MEA productions — and each of them produced a result. The productions

[31] That is, the testing of the last moved block was incidental to the chunk that was being built.

only tested attributes which were linked to the superstate except for the problem-space name, which was added to the productions to make them only apply in the subgoal. When the chunking mechanism examined the trace to create the chunk, it found that it should include exactly the conditions that the productions checked in the new chunk. So you would say that Soar learned something here with the chunking mechanism, but something that it already knew. The reason that there was no difference between what it already knew and what it learned was that Soar did not really have to perform any problem solving to resolve the impasse — it just had to apply the two MEA productions. This section demonstrates another example which does not use MEA to solve the question of which operator it should apply, but instead uses a simple one-step lookahead search. The knowledge-level difference between these is that the MEA goals provided Soar with global information about which operators were better than others. In the current example, the program will not include that information, but will try out an operator and see if it would lead to a state that is on the way to the desired state. Note that this is a general method for achieving progress toward a goal, that can apply whenever intermediate steps toward the goal can be identified, and will be further discussed in section 16.

15.3.1 The overall behavior

The search is initiated in the same way as before: the `move-block` operators are proposed, but not made indifferent, so Soar reaches an operator tie impasse. In the substate, the program proposes to evaluate each of the tied operators. To support this, it copies the important attributes off of the superstate. This allows the lookahead search to modify these values without changing the "official" values which are on the superstate. The `evaluate-operator` operators *are* made indifferent, so Soar randomly chooses one of them. There are no application productions for evaluate-operator, however, so another impasse arises, an operator no-change impasse this time. The purpose of this impasse is to allow the evaluation of each operator to occur "within its own space". Within the third level substate, the original application productions for the `move-block` operators will apply just as if they were in the topstate.

When the `move-block` operator is completed, its the new state is compared to the desired state. If it is on the right path to a solution, a `success` evaluation is put on the corresponding `evaluate-operator` operator in the superstate. If not, a `failure` evaluation is posted. In the intermediate state (with the `evaluate-operator` operators), as soon as a success or failure evaluation is posted on one of the operators, a best or worst preferences is put on the corresponding operator in the topstate. This continues until the original operator tie is resolved: either by one operator getting a best preference, or all but one getting a worst preference. When the tie impasse is resolved, the evalation substate is popped off the stack.

After this round of lookahead search, the program has learned what it should do first, and Soar selects that operator, and the application productions move it. Then new `move-block` operators are proposed, and the lookahead search starts anew to find what to do next.

Remember that whenever a result is passed up, Soar creates a chunk which encodes in a production the conditions that led to the result. Results are passed in two different places in this example. First, when the third level state evaluates the results of a `move-block` operator and passes the `success` or `failure` evaluation to the `evaluate-operator` above it. The second place in which chunks are passed up is when the mid-level state uses the evaluation to place a best or worst preference on the superstate's operator. The corresponding chunks will be described below.

15.3.2 The code

The code (in file `blocks-chunk-lookahead.soar`) includes most of the original simple blocks world, with minor alterations to make it work with subgoaling, and with the MEA production which placed the desired-state information on the top state. Print out a copy of the program and follow along with this description of it. Production `blocks-world*name-choose-problem-space` detects the operator tie impasse, and names the problem space of the resulting state `choose-operator`. Next in the code are four productions which copy attributes from a superstate to its substate. This is necessary for the lookahead search so that the search doesn't change any of the "real" values, i.e. those that describe the state of the world. These four operators are needed to copy attributes which describe the world and the goal: `thing`, `desired-state`, `ontop`, and `clear`.[32]

The next production, `propose*evaluate*move-block`, proposes to evaluate a move-block operator. It checks the `item` attribute of the current state to find the operator which should be evaluated. Remember that when an operator tie impasse is reached, when Soar automatically creates the substate, it gives the substate certain attributes, namely:

```
^impasse tie
^attribute operator
^choices multiple
```

Furthermore, Soar creates an `item` attribute for each of the tied choices, operators in this case. So at this point, the substate has an `item` attribute for each of the `move-block` operators that was proposed in the top-state. For each item, `propose*evaluate*move-block` proposes and `evaluate-operator` operator with that item as the `eval-operator`. For example, if the superstate included a proposed operator to move block A to block B, then this production will propose an operator in the substate to evaluate moving block A to block B. The program makes all of these proposals indifferent so that Soar can just randomly choose between them. The production includes a check to see if a failure attribute has been put on the state for

[32]The default rules supply generalized productions to perform attribute copying. See section 16 for further details.

that operator so that Soar will not re-evaluate the operator. It need not include a check for success, because as soon as an operator is evaluated as successful, the result is passed up, resolving the impasse, and popping the subgoals off of the stack.

The next two productions, `terminate*evaluate*move-block*success` and `terminate*evaluate*move-block*failure`, terminate the `evaluate-operator` operators when success or failure has been recorded for them.

Production `blocks-world*name-eval-problem-space` names the `eval-operator` problem space which is created when there is an `operator-no-change` impasse in the `choose-operator` space. This impasse occurs because there is no information which applies in the `choose-operator` for how to perform an evaluate-operator. The reason for this technique is that we want Soar to be, in effect, applying 2 different operators at the same time. The higher level operator is the `evaluate-operator` operator. In order to implement that operator, Soar must implement the specific `move-block` operator that is linked in as the `eval-operator` of the superoperator. Because a state can only have one selected operator at a time, we must cause Soar to subgoal in order to evaluate an operator.

In the `eval-operator` problem space, we must do two things. First, we must apply the `move-block` operator that is being evaluated. Second, we must check to see if it is the right thing to do. Production `eval*propose-operator` proposes the `move-block` operator as long as it hasn't already been done in that state. Production `eval*propose*perform-evaluation` proposes the `perform-evaluation` operator whenever Soar enters the `eval-operator` problem space, and give it a `worst` preference so that it will only be evaluated *after* the `move-block` has been completed.

Next in the code file are four elaboration productions which recursively evaluate which blocks are stacked correctly according to the desired state and which are stacked incorrectly. The computation of these attributes follows the following rules:

- The table is always stacked right.
- A block on top of another thing is stacked right if it is supposed to be on that thing in the desired state, and if the other thing is stacked right.
- A block is stacked wrong if it is on top of a block that it is not supposed to be on top of in the desired state.
- A block is stacked wrong if it is on top of a block that is stacked wrong.

The next two productions return a result to the choose-operator space. Both productions check if the current operator is `perform-evaluation` to ensure that the move-block operator has been completed. If the block that was moved is stacked right, then a `success` attribute is added to the superstate, with the evaluated operator as its value. If not, a `failure` attribute is added, and given a parallel preference so that there can be multiple failed operators.

The final production, `choose-operator*return-success`, returns a result to the top-state. If the `choose-operator` space has a `success` attribute for some operator, it returns a best preference to that operator in the top-state. Because the original impasse was an operator tie, the impasse is resolved by this result, and the subgoals are removed.

15.3.3 The trace

The trace below shows this example in action. For more detail, run it yourself at different watch levels. We will just use watch 0 here for economy.

```
soar> source \$soar\_library/tutorial/blocks-chunk-lookahead.soar
********#*#********************soar> d

     0: ==>S: S1
Initial state has 1, 2, and 3 on the table.
     1:    ==>S: S2 (operator tie)
     2:       O: O14 (evaluate-operator 1 to 2)
     3:       ==>S: S3 (operator no-change)
     4:          O: O8 (move-block 1 to 2)
     5:          O: O16 (perform-evaluation)
Building chunk-1
     6:       O: O12 (evaluate-operator 3 to 1)
Firing chunk-1

     7:       O: O11 (evaluate-operator 2 to 3)
     8:       ==>S: S4 (operator no-change)
     9:          O: O5 (move-block 2 to 3)
    10:          O: O18 (perform-evaluation)
Building chunk-2
Building chunk-3
    11:    O: O5 (move-block 2 to 3)
Retracting chunk-3

    12:    ==>S: S5 (operator tie)
    13:       O: O23 (evaluate-operator 2 to 1)
Firing chunk-1

    14:       O: O25 (evaluate-operator 1 to 2)
    15:       ==>S: S6 (operator no-change)
    16:          O: O8 (move-block 1 to 2)
    17:          O: O26 (perform-evaluation)
Building chunk-4
Building chunk-5
    18:    O: O8 (move-block 1 to 2)
Achieved 1, 2, 3
System halted.

soar>
```

The processing is described below, decision cycle by decision cycle. The numbers on the lines above correspond to the numbers below, and describe the decision cycle which ends at that point (i.e. line number 2 describes the processing that occurs during the second decision cycle, and ends with the selection of operator `O14`).

1. The operator tie impasse is reached, and a subgoal created with state `S2`.
2. During this decision cycle, an `evaluate-operator` operator is proposed for each move-block operator and they are made indifferent so at the end of step 2, operator `O14` is chosen to evaluate moving block 1 to block 2.

3. The operator no-change impasse occurs and state `S3` is created for the subgoal.

4. The problem space for this state is named `eval-operator`, and then the `move-block` operator is proposed and selected.

5. The `perform-evaluation` operator is selected after the `move-block` is complete.

6. The evaluation produces failure (because moving 1 to 2 is not the next thing that should be done), and this allows the termination of operator `O14`, popping `S3` off of the stack. Operator `O12` is selected to evaluate moving block 3 to block 1.

7. `Chunk-1` fires, giving a failure result to this operator, and subgoaling is avoided. O11 is selected to try to move block 2 to block 3 (hint: this is the right thing to do next).

8. `Chunk-1` does not apply here (print it to see why), so an operator no-change impasse is reached again, and `S4` created.

9. Operator `O5` to move block 2 to 3 is proposed and selected.

10. O18 to perform the evaluation is selected.

11. Chunk-2 is built because a success result is returned to state `S2`. Next, the best preference for operator `O5` is returned to the top-state, and `chunk-3` is built. Both impasses are now resolved, and `S4` and `S2` are popped off of the stack. `O5` is selected as the operator for the top-state.

12. `O5` is applied, `chunk-3` is retracted because it no longer is supported (2 is `ontop` of 3), and new move-block operators are proposed, again with no selection information. An operator tie impasse is reached again, and `S5` is created.

13. `O23` is selected to evaluate moving 2 to 1.

14. It is rejected by `chunk-1`, and `O25` is selected to evaluate moving block 1 to block 2.

15. An operator no-change impasse is reached, and `S6` is created.

16. Operator `O8` is proposed and selected.

17. `Perform-evaluation` is selected to see if that was the right thing to do.

18. Local success is achieved, results are created, chunks are built, and states are popped off of the stack. `O8` is selected in the top state.

19. The desired global state is achieved.

The trace below shows what happens when the example is run again.

```
soar> init-soar
soar> d

     0: ==>S: S1
Initial state has 1, 2, and 3 on the table.
Firing chunk-3

     1:    O: O5 (move-block 2 to 3)
Firing chunk-5
Retracting chunk-3

     2:    O: O8 (move-block 1 to 2)
Achieved 1, 2, 3
System halted.

soar>
```

15.3.4 The chunks

The chunks that were built while running this example are included for your reference, and formatted for your convenience. Read them and understand them, and understand why they were built.

`Chunk-1` performs the evaluation in the `choose-operator` space of operators which fail to perform the next needed move.

```
sp {chunk-1
    :chunk
    (state <s1> ^operator <o1>
                ^problem-space <p1>
                ^desired-state <d1>
                ^ontop <o2>)
    (<o1> ^name evaluate-operator
          ^eval-operator <e1>)
    (<p1> ^name choose-operator)
    (<e1> ^name move-block
          ^moving-block <m1>
          ^destination <d2>)
    (<o2> ^top-block <d2>
          ^bottom-thing <b1>)
    -{ (<d1> ^ontop <o3>)
       (<o3> ^top-block <d2>
             ^bottom-thing <b1>)}
    -->
    (<s1> ^failure <e1> +
          ^failure <e1> &)}
```

Chunk-2 gives a success evaluation to an operator which puts the appropriate block on the block which is on the table and which is supposed to be on the table (in the desired state). This chunk has been reorganized to make it more easy to understand.

```
sp {chunk-2
    :chunk
    (state <s1> ^operator <o1>       O1 is current operator
          ^problem-space.name        current prob space is
          choose-operator            choose-operator
          ^thing <t1>                t1 is a table
          ^ontop <o2>                d1 now on table
          ^desired-state <d2>)
    (<t1> ^name table)
    (<o1> ^name evaluate-operator
          ^eval-operator <e1>)       evaluating move-block
    (<e1> ^name move-block           m1 to d1
          ^moving-block <m1>
          ^destination <d1>)
    (<o2> ^top-block <d1>
          ^bottom-thing <t1>)
    (<d2> ^ontop <o3>
          ^ontop <o4>)
    (<o3> ^top-block <d1>            in desired state, d1 is
          ^bottom-thing <t1>)        on table
    (<o4> ^top-block <m1>            and m1 is on d1
          ^bottom-thing <d1>)
    -->
    (<s1> ^success <e1> +)}          register success
```

Chunk-3 creates the operator selection information (the best preference) for just such an operator, avoiding the evaluation procedure altogether.

```
        sp {chunk-3
            :chunk
            (state <s1> ^operator <o1> +
                        ^thing <t1>
                        ^ontop <o2>
                        ^desired-state <d2>)
            (<o1> ^name move-block
                  ^destination <d1>
                  ^moving-block <m1>)
            (<t1> ^name table)
            (<o2> ^bottom-thing <t1>
                  ^top-block <d1>)
            (<d2> ^ontop <o3>
            ^ontop <o4>)
      (<o3> ^bottom-thing <d1>
            ^top-block <m1>)
      (<o4> ^top-block <d1>
            ^bottom-thing <t1>)
      -->
      (<s1> ^operator <o1> >)}
```

Chunk-4 gives the evaluation information for the move-block operator which will put the top block on a three block stack, if the other two are already in position.

```
  sp {chunk-4
      :chunk
      (state <s1> ^operator <o1>
                  ^problem-space <p1>
                  ^thing <t1>
                  ^ontop <o2>
                  ^ontop <o3>
                  ^desired-state <d2>)
      (<o1> ^name evaluate-operator
            ^eval-operator <e1>)
      (<p1> ^name choose-operator)
      (<e1> ^name move-block
            ^destination <d1>
            ^moving-block <m1>)
      (<t1> ^name table)
      (<o2> ^bottom-thing <t1>
            ^top-block <b1>)
      (<o3> ^bottom-thing <b1>
            ^top-block <d1>)
      (<d2> ^ontop <o4>
            ^ontop <o5>
            ^ontop <o6>)
      (<o4> ^bottom-thing <t1>
            ^top-block <b1>)
      (<o5> ^bottom-thing <b1>
            ^top-block <d1>)
      (<o6> ^bottom-thing <d1>
            ^top-block <m1>)
      -->
      (<s1> ^success <e1> +)}
```

Chunk-5 performs the same function for chunk-4 that chunk-3 did for chunk-2.

```
  sp {chunk-5
      :chunk
      (state <s1> ^operator <o1> +
                ^thing <t1>
                ^ontop <o2>
                ^desired-state <d1>
                ^ontop <o4>)
   (<o1> ^name move-block
         ^destination <d2>
         ^moving-block <m1>)
   (<t1> ^name table)
   (<o2> ^bottom-thing <t1>
         ^top-block <t2>)
   (<d1> ^ontop <o3>
         ^ontop <o5>
         ^ontop <o6>)
   (<o3> ^bottom-thing <t1>
         ^top-block <t2>)
   (<o4> ^bottom-thing <t2>
         ^top-block <d2>)
   (<o5> ^bottom-thing <d2>
         ^top-block <m1>)
   (<o6> ^top-block <d2>
         ^bottom-thing <t2>)
   -->
   (<s1> ^operator <o1> >)}
```

15.4 Data Chunking

There are two basic types of chunking: that which encodes problem-solving knowledge, and that which learns facts. The former has been discussed previously in this lesson. The latter is called data chunking, and will be described here.

When Soar learns, it learns chunks that go into long-term memory. These chunks generally specify procedural knowledge — what to do in certain situations. In some cases, however, it is desirable for Soar to learn information that is more declarative, like facts and associations. This is where "data chunking" comes in. Data chunking is a general term for a technique by which Soar learns chunks which contain declarative information.

The setting for this procedure is that the intelligent agent experiences some target object along with a related cue. Later on, the agent when presented with that cue, is supposed to recall the target object. The basic requirements for Soar to learn this type of information are:

- an impasse which can be resolved in order to create a chunk
- a result that returns something to the superstate which indicates the recall value to be associated with the cue
- a chunk that says if the cue is X, then the target is Y

15.4.1 A data-chunking strawman

The obvious way to try to do data chunking would be to arrange to have some type of impasse (state no-change or operator no-change, depending on the context) occur when the cue and target have been presented. Within the subgoal, productions can examine the target and cue, and place a result on the superstate which is triggered by the cue and which identifies the target. Sounds simple, and it is, as these two productions indicate:

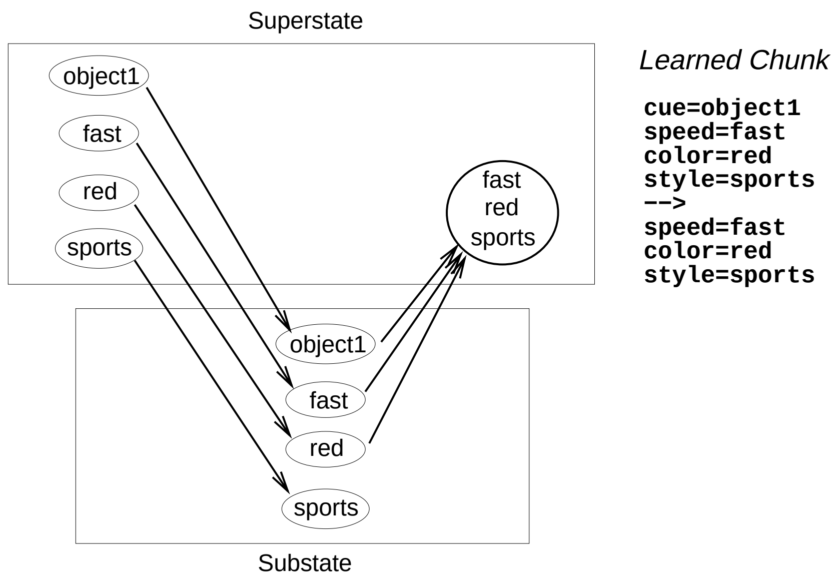
```
sp {initialize-state
    (state <s> ^superstate nil)
    -->
    (<s> ^cue object1
         ^target <targ>)
    (<targ> ^color red
            ^speed fast
            ^type sports)}

sp {recognize
    (state <s> ^superstate <ss>)
    (<ss> ^cue <cue>
          ^target <targ>)
    (<targ> ^color <color>
            ^speed <speed>
            ^type <type>)
    -->
    (<ss> ^recall <recall>)
    (<recall> ^color <color>
              ^speed <speed>
              ^type <type>)}
```

The first production puts the information on the state that the cue is "object1" and the target is a red, fast, sports car (the car is implicit). Because no operators are proposed, a state no-change impasse results, and the second production will fire. It checks the cue and the target, and places a recall object on the superstate. This results in the creation of a chunk.

The problem is that the resulting chunk will look very much like the second production (although it will access the current state instead of the superstate). As shown in figure 24, it will check not only the cue, but the *target* as well. So we do not get a *recall* chunk that would return the target item based on the cue. Instead a *recognition* chunk is created, i.e. one that basically says, "If you show me a cue and an associated target, I will say whether I've seen that pair before."

In order to get a pure recall chunk, it is necessary to eliminate the target information from the conditions of the chunk. But it's not quite obvious how to do that, because



```
cue=object1
speed=fast
color=red
style=sports
-->
speed=fast
color=red
style=sports
```

Figure 24: The simple approach to data chunking creates a recognition chunk. The arrows represent the creation of acceptable preferences.

we certainly have to check that information in order to specify what the target should be. There are two common solutions to this problem. One relies on a subtle feature of the Soar implemention. The other, slightly more complicated solution, uses a two-phase approach to building the chunk.[33]

15.4.2 Taking advantage of desirability preferences

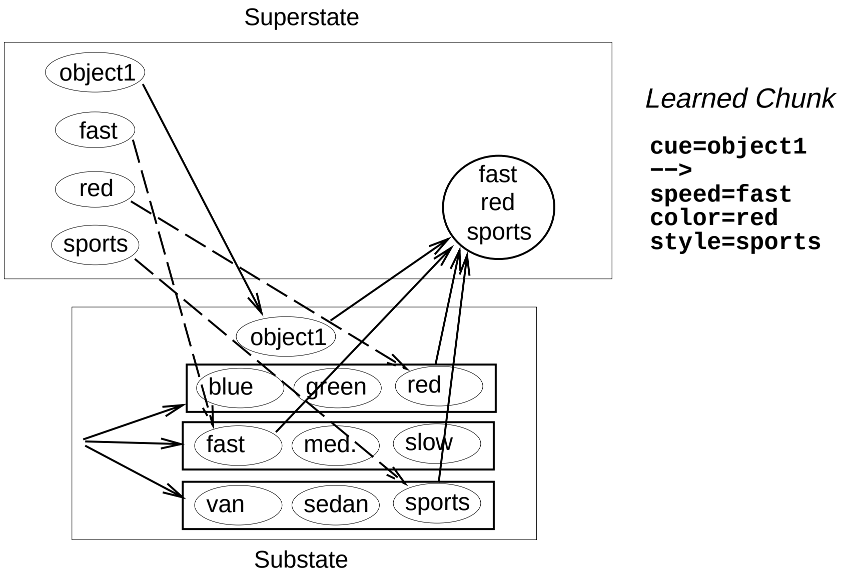
One slightly controversial feature of Soar is that it does not backtrace through desirability preferences. The reason for this is that these preferences are supposed to be used just to provide search control information to the problem solving procedure. This information could make the procedure more efficient, but shouldn't affect its correctness, i.e. it should not make the difference between a program that finds a good solution and one that doesn't. Our first blocks world program, for example, used no search control, and though it might take a long time to eventually "stumble on" the correct final state, it did not risk getting into a dead-end state, from which no solution could be found. The means-ends analysis rules from section 12 added search control information which sped up the task considerably.

Some people claim that not backtracing through desirability preferences is inconsistent (or at least an uncessary special case) because all other preferences are traced (but not negations). And in the future, this feature of Soar might change. Nevertheless, the first approach to data chunking takes advantage of this feature. The idea is

[33]It may seem unfortunate that one must go through so much difficulty just to learn a fact. But it might explain why it's much easier to recognize something ("Oh that's what his name is") rather than to recall it ("What was his name?").

to have the productions give `best` preferences to the features from the target item. Then when backtracing occurs for the creation of a chunk, the target features are not included in the LHS of the chunk.

So far, so good. But the target features in the substate must get acceptable preferences from somewhere. And it would not be good to have a production which conveniently just proposes those features — that would be basically the same as the desired recall rule. The solution to this problem is to have "generation" production(s) which give acceptable preferences to *all* relevant features in the substate. Then the best preferences mentioned above serve to pick out the target features, and to supply them for the result. And they don't have the undesired effect of being backtraced to the superstate. As figure 25 shows, the only direct link back to the superstate is from the cue.



```
cue=object1
-->
speed=fast
color=red
style=sports
```

Figure 25: A viable approach to data chunking. The solid arrows represent the acceptable preferences, and the dashed ones represent best preferences.

The example file `$soar_library/tutorial/data-chunking/simple.soar` contains a minimal example[34] of a program which implements this form of data chunking. The productions initialize the state, just as those shown above do. When an impasse occurs, the `generate-values` production gives acceptable preferences to all values for the appropriate features (three values for each of three features in this example). The `prefer*color`, etc. productions simply give a best preference to the value that matches the target's value. Finally, the `recall` production creates a result: a recall object on the superstate with the features and values from the substate. This results in the creation of a chunk that simply checks for the cue on the LHS, and creates the recall object for the target on the RHS.

[34]This example is minimal in that it doesn't do some things that a working data chunker would do, for example naming the subspace, and creating operators.

A real example In his thesis work, Seth Rogers needed to create data chunks for numbers. Seth's example addresses the problem of what to do when there is a potentially large set of target symbols. In this case, it could be any integer between 0 and some specified upper bound. Instead of generating all of the integers in that range, his approach independently generates each digit for each position (or "place", e.g. ones, tens, hundreds) in the number. Thus, for integers less than 1000, the system must only generate 30 items instead of 1000. His solution to this problem is included in the example files data-chunking/top-ps.soar and data-chunking/number-chunk.soar. We leave the running and understanding of this example as an exercise. Source the two files, and run the program. It will ask for a cue and a target number, and learn a chunk which associates the two.

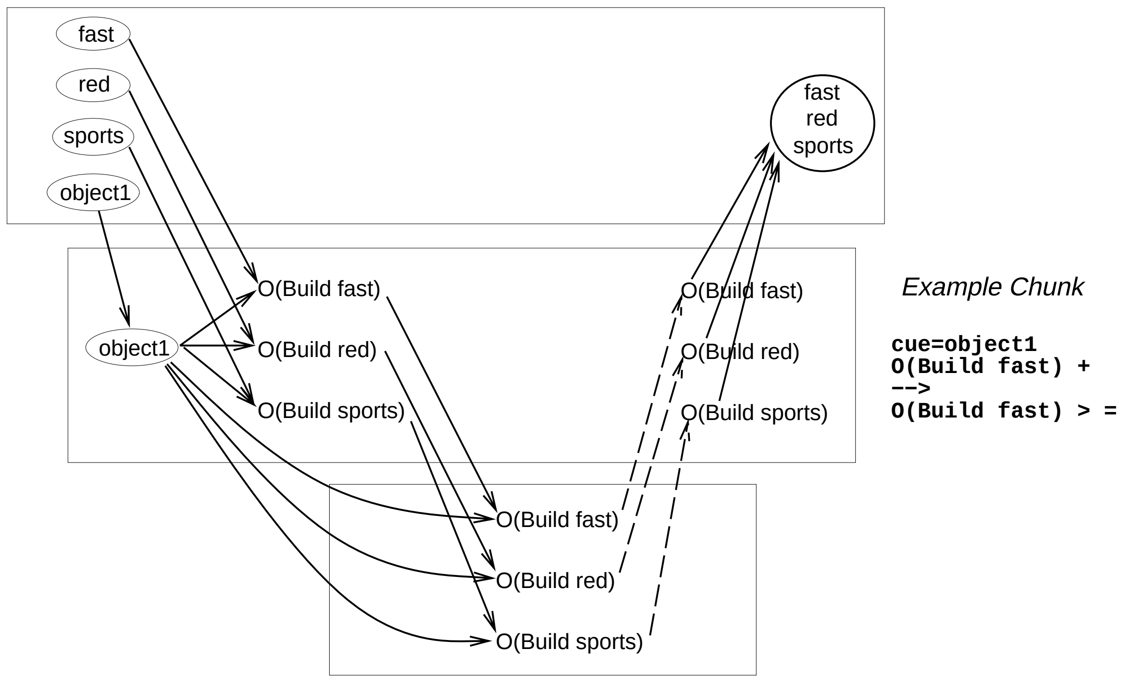
15.4.3 A better approach to data chunking

A couple of years ago, Rick Lewis presented an approach to data chunking that works whether or not there is backtracing through desirability preferences. It uses a two pass process, to first train operators that recognize target values, and then to turn them into a data chunk. Because Doug Pearson did an excellent job explaining and implementing this procedure in the the files in the example directory data-chunking/rick-doug (see the READ.ME file, especially) , I will just give an overall sketch of the procedure here.

Briefly, the idea of this approach is that in the first pass, there are operators proposed to "build" (or create) each of the target features separately. The crucial result of this pass is that Soar learns chunks which give each of those particular operators a best preference. In a sense, it learns to recognize the individual features of the target. In the second phase, build operators are proposed for all appropriate feature/value pairs. Then those previous chunks fire and give preference to the target features. When these build operators are applied, they create a recall object, which, when it's passed to the superstate, results in the creation of the desired recall chunk.

As depicted in figure 26, during the training phase, the `instance` operator is proposed, but not applied in the task problem space, so an operator no-change impasse ensues. In the subgoal, a `build` operator is proposed for each feature/value pair. Because they are tied, another impasse is reached. In this subgoal, productions fire which give each of these operators a best preference and an indifferent preference in the superstate, creating chunks which say, for example, if there is an operator to build the color red, then make that operator best. Because the operators in the middle state are now indifferent, they are chosen at random, and together create a recall object. After they're selected, the `finished` operator passes that object to the topstate's operator. That ends the training phase (and results in the creation of a recognition chunk).

Figure 27 depicts what happens in the recall phase. First, the `recall` operator is proposed in the topstate. It triggers an operator no-change impasse, and the cue is brought down into the generation subspace. This time, a `build` operator is proposed for each value of each attribute. This is the generation step. At this point, the



```
cue=object1
O(Build fast) +
-->
O(Build fast) > =
```

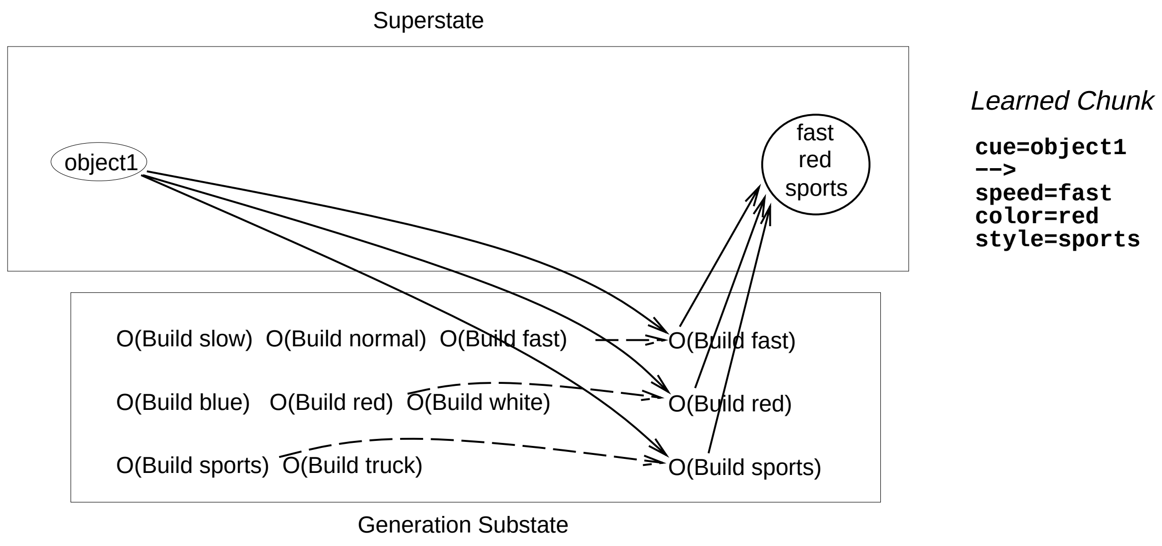
Figure 26: The training phase. Soar learns chunks that look for the cue and one of the target's build operators, and create a best preference for that build operator. (Figure from Doug Pearson)

previously learned chunks fire and give best preferences to the build operators for the target values. (Remember that these chunks depend only on the cue, and the existence of the particular value — not on the entire target description.) Again, each build operator is selected and creates the recall object. Finally, the `finished` operator is selected, and it puts the recall object on the superstate. This time, a different chunk is built — a recall chunk — because neither the production which generates the build operators nor the previously learned chunks which give the best preference to the proper build operators refers to the target features. Run the example following the instructions in the READ.ME file to see how it works.

15.5 The art of chunking

Now that you have seen these examples of chunking in action, you may be thinking that you have a handle on what chunking is, but you are not quite sure how you would get chunking to work for you with your own task. The best way to get this knowledge is by experience — try out some chunking programs and see what happens, and try to figure out what goes wrong.[35] This section gives some guidelines or hints

[35]Soar's `explain-backtraces` command is very useful for this purpose. At the beginning of the run, turn on explain with the command `explain-backtraces -on`. After a chunk has been learned, the command `explain-backtraces` *chunk-name* will print out details of how that chunk was formed. For details, see the manual.



```
cue=object1
-->
speed=fast
color=red
style=sports
```

Figure 27: The recall phase. Build operators are proposed for each value of each feature. Then the previously learned chunks select those appropriate to the cue. (Figure from Doug Pearson)

to help you along that path.[36] Some of the specific difficulties with chunking are also addressed in manual.

The fundamental aspects about chunking to remember are:

- Chunks are a by-product of the subgoaling process.
- Chunks are built when results are created on a superstate from a substate.

Most of the principles described below are derived from these basic facts about chunking, but they may not be obvious at first glance. As you read and understand these hints, you will notice that many of them are interrelated, because they all come from these basic facts of learning in Soar.

15.5.1 Monotonicity

A basic assumption about the way that chunking (and more fundamentally, Soar) works is that the Soar program is incrementally finding out more and more about the problem that it is trying to solve. It is not expected that features will be added and taken away, or that particular values of features will change in the time that it takes for you to resolve the latest impasse. (Of course, your Soar program can do these things, but then learning might not work so well with them.) Suppose, for example, there is a particular value of 5 for the attribute `distance` in some superstate at a particular time, and a subgoal checks that value. If later on that value changes to 7,

[36]Many thanks to Doug Pearson for providing these guidelines.

and is checked again by the same subgoal, then Soar would create a chunk that tests if the distance is 5 *and* 7. This is known as a non-contemporaneous chunk. It's not crucial that you know the name, but if you ever encounter it, you can proudly say, "Oh yes, I've created non-contemporaneous chunks too!"

15.5.2 Copy with tiny productions

As we saw in the previous example, one important thing to do when subgoaling is to copy some features from the superstate to the substate. When doing this, you should use small productions to copy each required feature separately. The reason is that if you have one big production which copies everything that you need, and then if you test just one of those features when returning a result, the backtracing procedure will find that the result depends on all of the copied features, and you will get an over-specific chunk.

15.5.3 Avoid desirability preferences

Soar does not backtrace through desirability preferences (better, best, worst, or worse). The reason is that these search-control preferences are not supposed to affect the *correctness* of the program. They're just supposed to make it run faster. If you use search control preferences in subgoals, then you must be very careful. If the reason for solving a problem in a particular way depends on thes search control, then Soar will learn the wrong chunk as it skips search control during chunking. If the choice of an operator or attribute is critical to the correctness of the program, you should use `require` or `prohibit` preferences which are backtraced by Soar.

15.5.4 Avoid negations in productions

Soar does not backtrace over local negations either. The reason for this is that it is too difficult to figure out why an attribute is *not* there. So if a production tests for the absence of some attribute or attribute/value combination on the substate, and then returns a result, there will be a chunk created, but it won't include anything about the negation test. Backtracing does include negations on superstates, though, so chunks can include negations in their conditions.

15.5.5 Make concepts and relations explicit

In general, it is best to make anything that you are testing explicit on the state. Any production which tests values on the superstate and creates a result, will end up with the explicit values in the chunk. For example, let's say you want to check if two planes that are controlled by Soar are heading in the same direction, and they are both flying east, or 90 degrees. If a production checks if the superstate's direction-one is X and the superstate's direction-two is X, and then creates a result, it will produce a chunk that includes the explicit value of X, i.e. 90.

The right way to do this would be to have a production in the superstate which tests those values, and indicates on the superstate if they are equal or not. Then the substate can simply check if that indicator is on the state, and it will create an appropriate chunk. In general, if you want to test some relation, it is best to "put that relation on the superstate" instead of creating the relation in the substate.

15.5.6 Numbers are pesky

Numbers are difficult for chunking for a number of reasons. One aspect of the problem showed up above: constants which are tested by productions get put directly into chunks. So a production that tests some numerical value on the superstate will produce a chunk with exactly that number in it. It may be that you want to produce chunks with some value range in them, for small, medium, and large values of a feature, for example. In this case, you should put those on the state, by testing (in the superstate) the number and putting the appropriate indicator (small, medium, or large) on the superstate.

Another aspect of the problem with numbers that was described above, is that relations between numbers are lost when they are checked in the substate. If one number is less than or greater than another number, and that relation will be important for chunking, you should put that information on the superstate.

15.5.7 Results which don't resolve impasses can cause problems

Ideally, when a result is passed to the superstate, it will resolve the impasse. If the subgoal stays around after a result, non-contemporaneous chunks can pop up, due to the testing of some attribute at two different times.

Note, however, that this is just a general guideline, and won't necessarily lead to trouble if it is violated. In our lookahead search example, the lower level chunks resolved the operator no-change impasse in the choose-operator space by giving the operator a success or failure evaluation and thereby allowing the operator to terminate. A successful evaluation also led to resolution of the top-level operator tie impasse, because Soar could choose that operator. A failure evaluation, however, did not resolve the operator tie impasse because it just eliminated that one operator from consideration (unless it left only one acceptable operator at the top level).

15.5.8 Sharing is dangerous

As previously mentioned in this section, copying information down from the superstate is a common activity in subgoaling, and must be approached cautiously. Another possible pitfall of copying is that a complex object might be shared and not copied.

Take for example, the object that describes a block, which has color, and type, and name attributes. If you check the block that is held by the gripper in the superstate and copy it to the substate (... `(<s> ^superstate.holding <x>) --> (<s> ^held-block <x>)` ...), and then if you change something about that object (the color, for example, with a painting operator) in the subgoal, then you are in fact, passing a result. Because the block is also on the superstate, and you are changing one if its attributes, you are making a change to the superstate, and a chunk will be built.

To copy the block object, for example, you would build a new `^held-block` object as shown below:

```
...
(<s> ^superstate.holding <x>)
-->
(<s> ^held-block <y>)
(<y> ^copy-of <x>)
```

Next, copy each feature of `<x>` onto `<y>`:

```
...
(<s> ^superstate.holding <x>)
     ^held-block <y>)
(<x> ^color <color>)
(<y> ^copy-of <x>)
-->
(<y> ^color <color>)
```

Of course, you must share some things, for example, the `io.output-link`, where commands are placed. This is often done via the topstate link which is passed to substates. It's a good practice in general to make this the only thing that is shared between states.

15.5.9 Over-general and over-specific chunks

This is a fairly basic statement, but one that should be made. You will often get over-general or over-specific chunks. If you get over-general chunks, that means that the chunks test too few things, underconstraining the situations in which they apply. This happens because the productions which led to the chunks tested too few things on the superstate. Look at those productions to see what else they should test.

If you have chunks which are too specific, and don't apply in enough situations, then you have too many conditions in the productions which led to the chunks. Look in those productions for conditions that can be eliminated.

15.5.10 Keep the superstate/substate distinction in mind

When thinking about your program, try to maintain a mental inventory of which objects are just in the substate, and which are in the superstate. Ensure that the conditions and actions in your productions test and create the objects that you want them too. If they should create chunks, then they should be modifying superstate objects, and be appropriately linked back to the superstate. If they should not create chunks, then they should only be testing and creating preferences for the substate.

15.6 Exercises

To see an example of chunking gone awry, load and run the program in blocks-chunk-op-app.soar. This example learns chunks which apply the move-block operator. Like our first example in this chapter, the appropriate productions are altered to apply only in a subgoal. In this case, the operator application rules have been moved. Soar learns chunks which perform the operator application, but the learning is incomplete. See if you can figure out why.

When you throw an external environment in with a lookahead search, things get rather tricky. In the previous lookahead search example, the Soar program used copies of the state on which to "try out" the moves and then evaluate them. But when the system is working with an external simulator, the moves occur in the simulated world. Thus there are two choices for doing lookahead search: either the evaluated moves can be performed via the simulator, or they can be performed "in Soar's head" with copies of the important state information. Each of these options produces additional complexity. For the former, the simulator must be extended to be able to undo its operations, so that the original state is not altered when an operator is evaluated. For the latter, the program must include operator application rules for performing the actions via the simulator, *and* for performing the actions internally, on its own state. The file blocks-chunk-lookahead-ext.soar contains an example of this latter approach. Run it, and examine its performance, and the chunks it builds. Are the chunks that are learned in internal lookahead search general enough so that they can be used by the simulator?

16 Methods: The Default Rules

In the previous lesson, we saw how chunking allows Soar to create general rules to avoid impasses in various situations. The program used a technique known as a lookahead search, using its knowledge of the states and operators to "imagine" what would happen if it applied a particular operator. In order to do the lookahead search, Soar copied some aspects of the state, and evaluated the effects of various operators on the state. Then it set up special problem spaces for the evaluation of the task operators, performed the search, and assessed the results. Wouldn't it be nice if there were a standard set of productions that would encode this and other basic problem solving strategies? In fact, there is such a set of productions, called the Default Rules.

The goals of this lesson are to:

- introduce the Default Rules
- describe their foundations in AI weak methods
- show how to "interface" your program with the Default Rules

16.1 Weak methods to default rules

The fundamental motivation for the creation of the default rules was, as mentioned above, to provide a set of basic problem solving techniques that could be applied to a wide variety of problems. This came out of the concept of *weak methods*, a variety of general purpose, task independent, heuristic search procedures for solving complex problems. These can be thought of as a form of common sense reasoning for figuring out what to do in the absence of domain-specific problem-solving knowledge. Two specific weak methods which are implemented in the default rules are depth-first search and credit and blame assignment. Guidelines for including these techniques in your programs are given below.

16.2 Integrating the default rules

In the best of all possible worlds, the default rules would automatically perform whichever techniques were appropriate for a given situation. But when dealing with computers, things are never so simple. The default rules contain many attribute and value names which must coincide with those that you use in your program in order for them to cooperate. These attribute and value names provide a canonical organization for working memory and consequently influence what your Soar programs should look like. This canonical organization is implicit in the augmentations which the default rules create, and those which they check for in their conditions. The manual describes in some detail what each group of default rule productions does, so that a program can use them. In the Coloring Book, we will provide the reverse: a set of situations in which the default rules could come in handy, and a description of how to use them to tackle those situations.

The default rules are located in the file `$soar_library/default.soar`.[37] The default rules provide methods for:

- setting up useful general attributes on the top-state and other states
- default choices for impasses
- proposing a wait operator to be selected if no other operators are proposed
- performing a lookahead search, including copying the state
- operator subgoaling (i.e. subgoaling to change the state in order to apply the current operator)
- halting Soar when success or failure has been indicated

In this lesson, however, we will just concentrate on some of the most common things that can be done with the default rules: wait operators, lookahead search

16.3 The Wait operator

When you think about it, waiting is one of the most common things that we do. In this respect, many cognitive models are a bit strange — too hyperactive, always trying to *do* something. This becomes especially obvious when one is interacting with an external environment. Just as we have to wait for the computer to complete whatever processing it is doing, a cognitive model of some process would reasonably be expected to wait until some aspect of the external environment has been changed. For example, if the blocks world simulator were changed so that it took several steps[38] for a `move-block` operator to complete, then the Soar program would have to wait for that operator to be terminated. A very simple method for doing this is the `wait` operator.

The `wait` operator is proposed by the default rules in the topstate, and give a worst preference. This ensures that if there are any other operators proposed in the topstate, `wait` will not be selected. If there are no other operators, however, wait will be selected. There are no application operators for `wait`. As soon as it is selected, the terminate production fires. The wait operator is quite simple to introduce into your program. It needs no special data structures on the topstate. You should just be aware that it is there. You can simply see the behavior of the wait operator by

[37]The default rules have numbers which correspond to the sections in the old Soar manual in which they were described. In the Soar7 manual, this information is in an appendix.

[38]Because the same process on the computer is running both the simulator and the Soar program, a multiple-step action is used to be an approximation of an action that has a duration longer than the cycle time of the cognitive process.

starting soar and just loading the default rules and no task-specific productions. Run some decision cycles, and you will see that wait is proposed, and then selected and terminated over and over and over again.

16.4 Lookahead Search

At the beginning of this lesson, we referred to the previous lesson's lookahead search example. The file blocks-lookahead-def.soar in the tutorial directory contains the Soar program which uses the default rules instead of custom-made productions to perform the lookahead search. Eleven of the original 22 productions (that were added to the basic blocks world productions) were removed because the default rules perform exactly those functions. One production was added to specify which elements of the state should be copied. And the names of some of the attributes were changed to accommodate the coding conventions of the default rules. This section describes how to create a program that uses the default rules to do lookahead search using this blocks world program as an example.

16.4.1 The desired state

In the sample blocks world program, the production blocks-world*put-goal-on-state puts a description of the desired final state onto the topstate. When evaluating an operator, the program compares the new state that has been generated by the application of that operator, and checks it against the desired state (using the stacked-right elaborations). Although this description is used in the lookahead search, the desired state description is not required by the default rules. The default rules do rely on the existence of a ^`desired` augmentation on the topstate. (Fortunately, the default rules also create this augmentation if you neglect to.) The default rules attach information about the evaluations of operators to the ^`desired` augmentation.

You can also modify the behavior of the lookahead search by attaching augmentations to the `desired` object. As described in the manual, you can specify whether evaluations of numerical values should be based on bigger numbers being better or smaller numbers being better. You can also specify that equal numerical values should not be treated as indifferent.

Despite the fact that the description of the final goal state is not required by the default rules, we should stress that it is a very important feature for a lookahead search. The towers-of-hanoi, waterjug, and eight-puzzle demo programs each use an explicit description of the final state to guide their lookahead search. The farmer and missionaries demos use specific productions which test the states generated by lookahead search for failure or success conditions which are implicit in the productions.

16.4.2 State copying

A key aspect of the lookahead search process is the notion of copying states in order to "protect" the original state from modification during the search for which operator to perform next. After all, in the missionaries and cannibals problem (or the farmer task[39]), one could hardly go back and try another solution, after arriving in a state in which one of the missionaries is eaten. The solution is to copy the state description, in essence creating missionary clones, for example, to ensure the safety of the originals during the search. The default rules provide mechanisms for copying state information. The task programmer need only specify which aspects of the state should be copied.

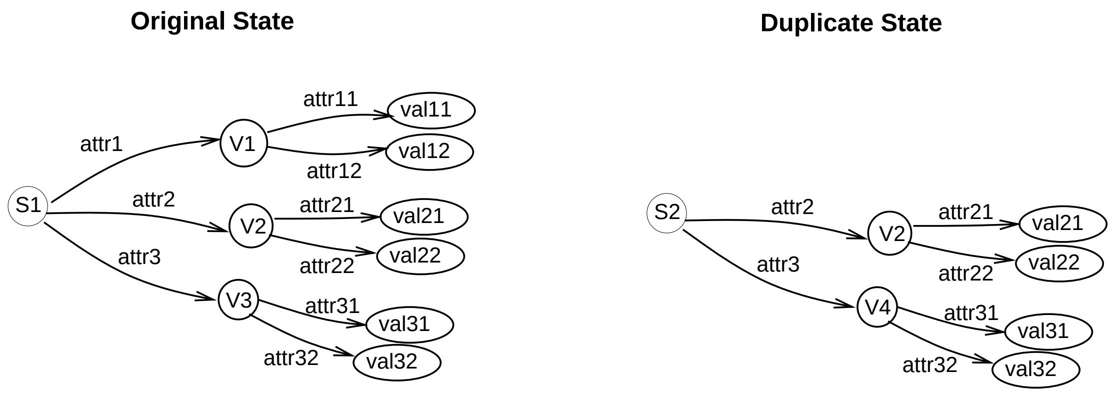


Figure 28: Original and new states with different levels of copying

The state copy mechanism supports three different levels of copying, as shown in figure 28. The first level is to not copy an attribute at all. In the figure, this is the case for attribute `attr1`. The second level (termed "copying" in the manual) is to give the new state the ssame attribute and value pairs as the original state. This amounts to the creation of a link from the substate to the value on the superstate, and you could imagine an attribute link named `attr2` from `S2` to `V2` in the figure. The important point to remember here is that any modification to `V2` will affect both the superstate and the substate.

The third level of copying is called duplication in the manual. As shown in the figure, the `attr3` attribute is copied to the substate. In addition, a new identifier, `V4`, is created, and the attributes of `V3` are copied to it. Now `V4` is a duplicate of `V3`. It has the same attributes and values. But changes to the top level of the `V4` object will not affect `V3`.[40] As an example of where this might be necessary, imagine that a block object on the state stored its position with `xpos` and `ypos` attributes (e.g. `^thing.xpos 2, ^thing.ypos 1`). During the course of the lookahead search, the

[39]For those not familiar with these tasks, they consist of a set of predators (e.g. cannibals), and a set of prey (e.g. missionaries) which must be ferried from one side of a river to another via canoe, while avoiding situations in which the predator(s) could eat the prey.

[40]The default rules do not contain a mechanism for copying objects that are more complex (i.e. that have more levels of attributes) than these.

position of a block would naturally change. But you wouldn't want to change the position indicators of the blocks on the top state, so in this case, duplication would be necessary.

In order to "request" that the default rules copy states when performing lookahead search, you must set attributes on the problem-space attribute of the top-state. These particular attributes are described in detail in the manual, but we will briefly define the basic ones here:

- ^`default-state-copy yes`
  Enable state copying.
- ^`one-level-attributes` attributes
  Copy the listed attributes.
- ^`two-level-attributes` attributes
  Duplicate the listed attributes.
- ^`all-attributes-at-level one` or `two`
  Copy every attribute on the state at level one (copying) or two (duplicating).
- ^`dont-copy` attributes
  Do not copy the listed attributes.
- ^`dont-copy-anything true`
  None of the attributes on the state should be copied.

In order to perform the copying shown in figure 28, the following production would be used to specify these attributes:

```
sp {copy-control
    (state <s> ^superstate nil)
    -->
    (<s> ^problem-space <p>)
    (<p> ^name superspace
         ^default-state-copy yes
         ^one-level-attributes attr2
         ^two-level-attributes attr3)}
```

In the example program blocks-chunk-lookahead-def.soar, we want to copy all of the `things` from the state, as well as the `clear` and `ontop` relations. Although these relations may change during the lookahead search, the operator application rules for `move-block` will add and delete them as necessary. None of these objects has complex attributes that must be duplicated, so we can just specify the attributes to copy at level one, as shown below:

```
sp {copy-control
    (state <s> ^superstate nil)
    -->
    (<s> ^problem-space <p>)
    (<p> ^name blocks
         ^default-state-copy yes
         ^one-level-attributes clear + &, thing + &, ontop + &)}
```

16.4.3 Credit and blame assignment

The third aspect of the lookahead search process that is supported by the default rules is credit and blame assignment. After a potential operator has been applied during the search, evaluation rules compare the resulting state to the desired state. Your task rules will perform the initial evaluation, and the default rules convert the evaluation into desirability or necessity preferences on the original operators (the ones that caused the original operator tie impasse).

There are two types of evaluations: numeric and symbolic. Numeric evaluations compare the values of attributes. Symbolic evaluations use a qualitative comparison of the state to the desired state. The result of the evaluation is placed on the superstate operator's (it should be an evaluate-operator operator) ^`evaluation` object. The evaluation object is created by the default rules along with the evaluate-operator operator. The attribute name depends on the type of evaluation.

Numeric evaluations If numeric results are computed to gauge the results of the lookahead search, then the values of the evaluations should be placed as the ^`numeric-value` attribute of the evaluation object. The default rules will compare these evaluations and produce `better`, `worst`, or `indifferent` preferences for the corresponding tied operators. As mentioned above, you can specify whether smaller or larger numbers are better by placing a ^`better higher` or ^`better lower` attribute on the problem-space object of the state at which the operator tie was originally encountered.

Symbolic evaluations When the result of the evaluated operator is qualitatively compared to the desired state, there are several different possibilities:

- If the resulting state matches the desired state, then a `success` evaluation should be given as the value of the evaluation object's ^`symbolic-evaluation` attribute. The default rules will then give the corresponding tied operator a best preference.
- If there is a partial match of the desired state (indicating one possible step on the way to a solution), those rules can supply a `partial-success` evaluation

to the superstate's `evaluate-operator` operator. The default rules then create a `best` preference for the corresponding tied operator.

- If the resulting state is a required step on the way to any solution, the evaluation can produce a `require-success` evaluation which the default rules turn into a `require` preference for the tied operator.
- If that state can be on no path on the way to a solution, the evaluation rules should return a `prohibit-failure` evaluation which will cause a `prohibit` preference to be passed to the tied operator.
- If the state is not a good step toward a solution, but is not a dead-end, a `partial-failure` evaluation can be returned, which gives the tied operator a `worst` preference.
- If the state is an obvious failure state, then a `failure` evaluation can be returned, which gives the tied operator a `worst` preference.

As previously mentioned, the evaluation rules are task specific, and you must provide them. The production below returns `partial-success` evaluations for our blocks-world example lookahead search. It checks if the block that is being moved by the currently evaluated `move-block` operator is stacked correctly. (This is computed by elaboration rules which compare the current state to the desired state.) If so, then the ^`symbolic-value partial-success` attribute is added to the evaluation object of the superstate's operator.

```
sp {eval*apply*perform-evaluation*success
    (state <s> ^operator.name perform-evaluation
               ^superstate.operator <so>
               ^stacked-right <block>
               ^ontop <ontop>)
    (<so> ^name evaluate-object
          ^object <eval-op>
          ^evaluation <eval>)
    (<eval-op> ^moving-block <block>
               ^destination <dest>)
    (<ontop> ^top-block <block>
             ^bottom-thing <dest>)
    -->
    (<eval> ^symbolic-value partial-success)}
```

17 Soar Input and Output

This section outlines how to do I/O (input/output) in Soar. The importance of I/O was partially covered in section 10 on external interaction. There are three basic types of I/O in Soar:

- text I/O
- tcl I/O
- C I/O

Text I/O allows for simple interaction with the user via reading and writing of text. It is appropriate for input if the user must make some simple decisions to influence the way that the program runs. For example, if some program has a training phase and a testing phase, text I/O could be used to prompt the user to specify which phase should be pursued. Text output is often used to indicate the progress of the program. For example, our simple blocks program printed a message when the state was initialized and when the goal was reached.

If the task requires a large amount of user interaction, or if some sort of graphical display is appropriate, then `tcl I/O` should be used. Tcl, the task command language [Ousterhout1994], extends the range of output possibilities in Soar. For example, Tk, the Tcl "took kit," provides a relatively simple way to create buttons, and menus, and graphical displays. The external interaction interface described in section 10 used tcl I/O.

Tcl is an interpreted (rather than compiled) language and can be rather slow, especially when many changes have to be made to a graphical display. In order to speed graphical display, interface routines can be programmed in the C programming language, and then compiled with Soar. C I/O requires both technical knowledge of C as well as expertise in linking different programs together. Additionally, C I/O does not significantly extend the functional capabilities of tcl I/O. Therefore, C I/O will not be covered in this tutorial.

The goals of this section are to describe:

- reading and writing text with the `accept` and `write` RHS functions
- how to do input from and output to a tcl program
- the basics of making a simple simulator with tcl

17.1 Accept and Write

As described in the Soar Manual, the `write` function (along with the `crlf` function) can be used on the right-hand side of productions for displaying textual output to the user, and the `accept` function can be used to get input from the user. This subsection describes the use of these functions.[41]

The `write` function takes one or more arguments and writes them to the window in which Soar is being run. The arguments must all be symbols; i.e., numbers or atoms. If you want to display a string of words, you must enclose them within vertical bars (e.g. `(write |an extended atom|)`) to create an atom. If you are using a `watch` command, the output of the `write` function may be mixed in with the watch output, so it is often useful to use the `crlf` function (short for carriage return/line feed) to separate the written output from the trace output. A simple production which prints a message when there is a success attribute on the state is shown below.

```
sp {print-success-message
    (state <s> ^success *yes*)
    -->
    (write (crlf) (crlf) |The goal has been reached!|)}
```

The `accept` function gets input from the user. `Accept` stops Soar processing and waits for the user to enter a number or atom and press the return key. Currently, the atoms must be simple atoms, and not surrounded by vertical bars. It is usually informative to print out a prompt message before an `accept` function so that the user knows that an action is required. As previously mentioned, user input can signal a particular phase of processing that the Soar program should pursue. The productions below show how the user can be prompted for input, and how that input can be used to affect further processing.

```
sp {get-user-input
    (state <s> ^choose-phase *yes*)
    -->
    (write (crlf) (crlf) |Enter desired phase (train or test): |)
    (<s> ^phase (accept))}

sp {start-test-phase
    (state <s> ^phase test)
     ...
```

17.2 tcl I/O

Using tcl for I/O processing allows a much wider range of interaction possibilities. There are three different levels of increasing functionality that can be achieved with tcl I/O:

[41] Note that `accept` is potentially a poor choice for some psychologically-motivated problems. `accept` causes the whole system to halt while waiting for the user to enter some information. Thus, `Accept` may not be appropriate for any model which includes perceptual components, or at least ones in which the specifics of input could affect the model's results and predictions.

- more powerful basic I/O
- windows, buttons, menus, etc
- simulators

This section describes what can be done at these different levels. It does not, however, describe tcl/TK. For information on this programming language, we refer you to [Ousterhout1994] and/or [Welch1995].

17.2.1 Basic I/O

The main advantage of doing I/O with tcl is that you have much greater control over the formatting of the output. Instead of using Soar's built-in `write` and `crlf` functions, you can use the `tcl` RHS function which passes its arguments as a string to tcl. Tcl then interprets the string and performs any functions that are specified in it. The functions can be built-in tcl commands or user-defined procedures, like `showspeed` below .

This example uses a production and a tcl function to print a "bar chart" of the speed that an a Soar agent is going (e.g., if Soar was controlling a simulated car). The production `write-speed` fires every time the speed changes. Whenever this production fires, it calls a tcl function called "showspeed" with an argument equal to the value of `<x>`. (Note the space after the showspeed procedure name. It is required in order to separate the function name from the argument.)

```
sp {write-speed
    (state <s> ^superstate nil
               ^speed <x>)
    -->
    (tcl |showspeed | <x>)}
```

Showspeed prints the speed in a special way (included below; you can ignore the specifics of tcl syntax). It prints a newline, and then prints one asterisk for each unit of the speed. In conjunction with a program that changes the speed augmentation on the top-state, this function will print out a graph of the speed.

```
proc showspeed speed {
   puts ""
   set counter $speed
   while {$counter >= 0} {
      puts -nonewline "*"
      incr counter -1
   }
}
```

Using Soar's `tcl` command and tcl procedures, you can tailor the output of your program to your specific needs.

17.2.2 Tk: Adding windows

Tk provides graphics primitives for Tcl. For example, Tk includes constructs for basic windows, menus, buttons and canvases for graphics. Even if you don't plan to build a Tcl/Tk simulation, these items can be quite useful for displaying the status of a running Soar program.

The code fragment below creates a top-level window, names it Speedometer, and creates a Tk scale object in it. The scale object shows the value of the global variable `speed`. The `display-speed` production sets the value of that variable. This example demonstrates a very convenient method for graphically displaying the value of a Soar WME.

```
toplevel .top
wm title .top "Speedometer"
set speed 0
scale .top.speed -label Speed -from 0 -to 120 -length 10c -orient \
   horizontal -variable speed -command newSpeed
pack .top.speed

proc newSpeed new {
   global speed
   set speed $new
}

sp {display-speed
    (state <s> ^superstate nil
               ^speed <x>)
    -->
    (tcl |set speed | <x>)}
```

In this example, the "scale" widget not only displays the value of `speed`, it can also be used to set the value of the variable, just by sliding the scale. We now outline how to move this new value into Soar's working memory as input.

17.2.3 Making a Simulator

Interacting with an external simulation demands a clearly defined interface between the intelligent agent and the world. Creating an external simulator with Tcl is fairly easy: Tcl is a simple scripting language and Tk provides a wide variety of graphical displays. We do not have space in the tutorial to cover the specifics of these languages, although you can examine the Tcl/Tk code in the $soar_library/tutorial/simulator/ directory to see how simulators are built. However, in this section, we introduce examples of the functions that define the interface between Soar and the external world: the tcl output function and the tcl input function. These two functions completely specify the communication between Soar and the external world.

Below is an example of a an output function. This example is from the `pick-up` and `put-down` exercise in section 11. The output function has to process the two commands and their arguments. `Pick-up` takes one argument, `block`, and `put-down` takes one argument, `on`.

```
proc my-other-output-procedure {mode outputs} {
    global io_header output_link_id

    set argument ""
    switch $mode {
        added-output-command {
            set output_link_id \
                    [get-output-value $outputs $io_header "output-link"]
        }

        modified-output-command {
            if [string match $output_link_id ""] {return}
            set command_name \
                    [get-output-value $outputs $output_link_id command]
            if [string match $command_name "pick-up"] {
                set argument \
                 [get-output-value $outputs $output_link_id block]
            } elseif [string match $command_name "put-down"] {
                set argument \
                 [get-output-value $outputs $output_link_id on]
            }
            if [string match $command_name ""] {return}
            $command_name $argument
        }

        removed-output-command  {
            puts "removed output command"
        }
    }
}
```

Soar passes the output function a “mode,” which can take one of the three values. The output function executes different code segments (“switches”) based on this mode value. When the output-link is first created (`added-output-command` mode), the output function saves the identifier number of the output_link_id to a global variable. The value is saved in order to easily reference the output-link. Soar sets the mode to modified-output-command whenever a WME linked from the output link is created or deleted. For example, adding the pick-up command to the output link will cause Soar to call the output function in this mode because the output-link has been modified with the new augmentation ^command pick-up. For the modified-output-command mode, the Tcl code uses the output_link_id to determine what items are linked from the output-link. It then parses these augmentations for strings matching either pick-up or put-down. If it finds these values, it also records the argument as well.

The last line inside the switch actually executes the command. For example, state that looked like this:

```
(S1 ^io <io>
    ^thing <block1>)
(<io> ^output-link <out>)
(<out> ^command pick-up
       ^block <block1>)
```

would result in the tcl command (assuming 2 is the block id):

```
pick-up 2
```

The final mode value, removed-output-command, occurs whenenever the output-link augmentation is removed from working memory. Normally, this removal occurs only when we initialize soar and thus there is no special processing associated with this command. If we had built a simulation that executed commands in parallel with Soar, instead of just executing them within the output function as we do here, we might want to use this mode to recognize that any executing commands should be terminated.

The input function for the blocks world simulator can be found in the file $soar_library/tutorial/simulator/soar7-tutorial.tcl. The structure of this function is similar to the output function. Again, Soar provides the input function with a mode value which is used to switch among different activities within the function. The mode top-state-just-created is called once, when Soar creates the state s1. In this mode, the primary activity is to save the input_link_id in a global variable. We will then use this variable to place additional input in the proper place in Soar's working memory.

For normal-input-cycle mode, the code checks the various objects in the simulated world. If it finds that an object or relation no longer exists, it removes the WME from Soar's input-link with the remove-wme command. Similarly, when an object or relation is found that has not yet been placed on the input-link, a new WME is created via add-wme command. Much of this function is commented out because it was originally created for a much more detailed simulator. Although this code is not used in the simple blocks world, you can examine it to see how other relations would be computed.

17.3 C I/O

If you really need speed in your input and output, then you should use C functions. Input and output in C is considered an advanced topic for Soar and is beyond the scope of this manual. You can read about it in the Advanced Soar Manual.

A Descriptions of example files

This appendix contains descriptions of the example Soar programs that were prepared for this tutorial. This description is also located in the file $soar_library/tutorial/README.

This directory contains a number of different examples that were created in conjunction with the Soar Coloring Book. Most of them use a simple blocks world with a gripper, and three blocks, labelled 1, 2, and 3, on a table.

Note: These files are expected to be in the tutorial subdirectory of the Soar library directory (to which your SOAR_LIBRARY environment variable should point). If you install them in a different place, you must change some of the pathnames in these files.

The files in this directory are:

blocks.soar:

The basic code which randomly chooses a move-block operator, and does problem solving "in its head".

blocks-mea-rules.soar:

This file contains three productions which add Means-Ends Analysis capabilities to the basic blocks world. These rules should be loaded in addition to the basic rules. One rule adds the desired-state information to the top-state. The second rule gives a best preference to an operator which moves its block into the desired ontop relation with another block. The third rule gives a better preference to an operator that moves its block into place over another that moves its block into place if that operator (the preferred one) moves a block lower in the stack than the other operator.

blocks-external.soar:

The blocks world code which uses the external blocks simulator. The original four operator application productions are replaced by one which places a move-block command on the output-link.

simulator:

This directory contains the tcl code which defines the external blocks world simulator.

blocks-chunk-mea.soar:

This file contains a simple example of chunking behavior in Soar. The chunks which are learned are equivalent to the Means-Ends Analysis productions in mea-rules.soar.

blocks-chunk-lookahead.soar

This file learns which operators to prefer in the blocks world task by performing a lookahead search. By removing the indifferent preferences, an operator tie impasse is encountered. To resolve it, the program randomly selects an operator to evaluate. It evaluates the operators by comparing their results to an explicit description of the desired state. When an operator is found that makes progress toward the state, a best preference is placed on that operator in the superstate. A chunk is learned to codify the results of the lookahead search.

blocks-chunk-lookahead-ext.soar

This takes the same approach as the previous one, but incorporates it with the external simulator. This is a bit tricky because unless there is knowledge about how to "undo" an operator, the lookahead search is incompatible with using an external simulator. If the simulator were to perform the potential operators for the lookahead search, then the program could make a wrong move and not be able to reverse it. So this program does lookahead search internally, or "in its head". The move-block command is issued to the simulator only after the search is completed, and the desired operator is found.

blocks-chunk-lookahead-def.soar

The same as blocks-chunk-lookahead.soar except that it uses the default rules instead of custom lookahead search. This required the changing of the desired-state attribute to just desired, and one production that specifies which parts of the state should be copied for the search.

pick-put.soar

This file is used for the first substantial program-creation exercise. The task is to perform the basic blocks task by using pick-up and put-down operators instead of move-block operators. This is accomplished by "multiple-choice programming". This file contains multiple different versions of each production that is required to perform the task. The student must comment out the improper productions to create a set of productions which will actually work.

pick-put-solution.soar

This is a properly commented-out version of the above file.

pick-put-subgoal.soar

This is the second major programming exercise. The task is similar to that in pick-put.soar, but instead of doing the pick-up and put-down operators at the top level, move-block operators are proposed at the top level, just as in the earlier versions of the blocks world task. There are no application productions for the move-block operators, though. They are implemented in a subgoal via pick-up and put-down commands. The exercise requires the user to write the required Soar productions by reading the comments in this file, and then producing the appropriate syntax for the production.

pick-put-subg-soln.soar

This is a solution to the exercise above.

data-chunking

This subdirectory contains the example files which implement data chunking.

travel.soar

This example demonstrates the use of subgoaling in a simple task in which the Soar program proposes an operator for a method of transportation that is suitable for the distance that must be travelled and the amount of money on hand. The amount of money and required-distance are stored onthe topstate. Various rules propose travel operators based on the distance. If there is a tie, productions count the amount of money in the subgoal, and break the tie.

travel-lookahead.soar

This is the same task as the previous example, but it implements the solution via a simple lookahead which checks different operators to see if there is any money left after applying them.

B Answers to exercises

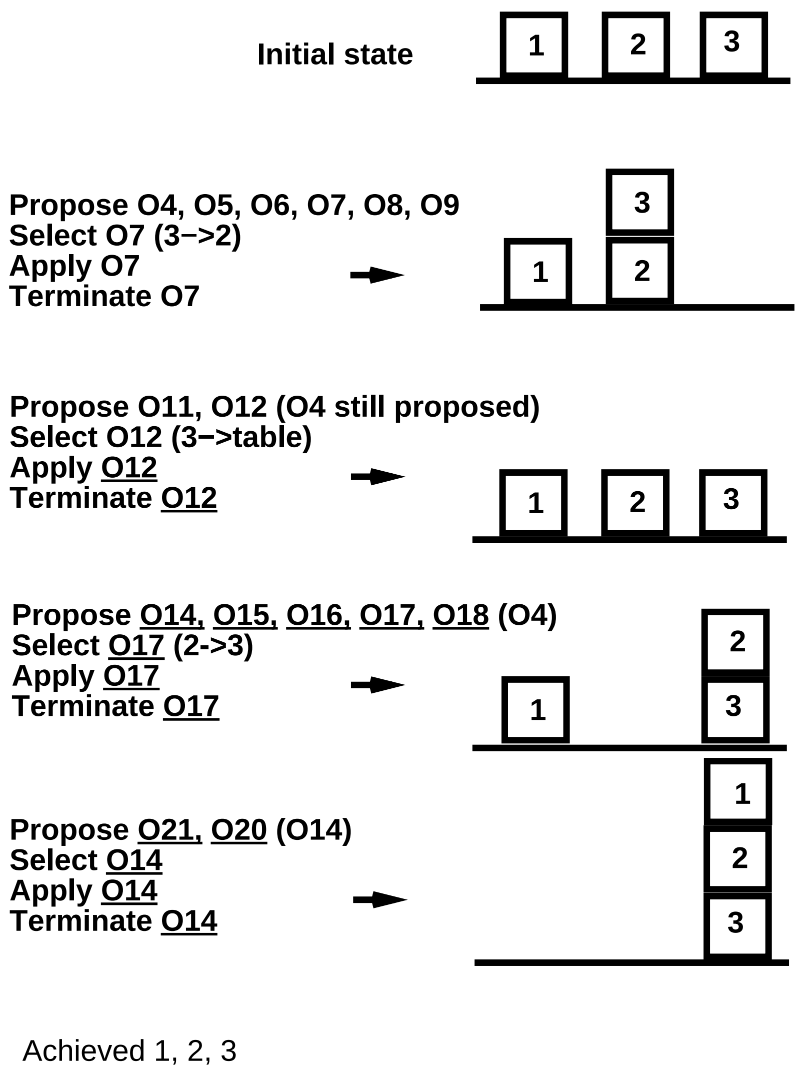
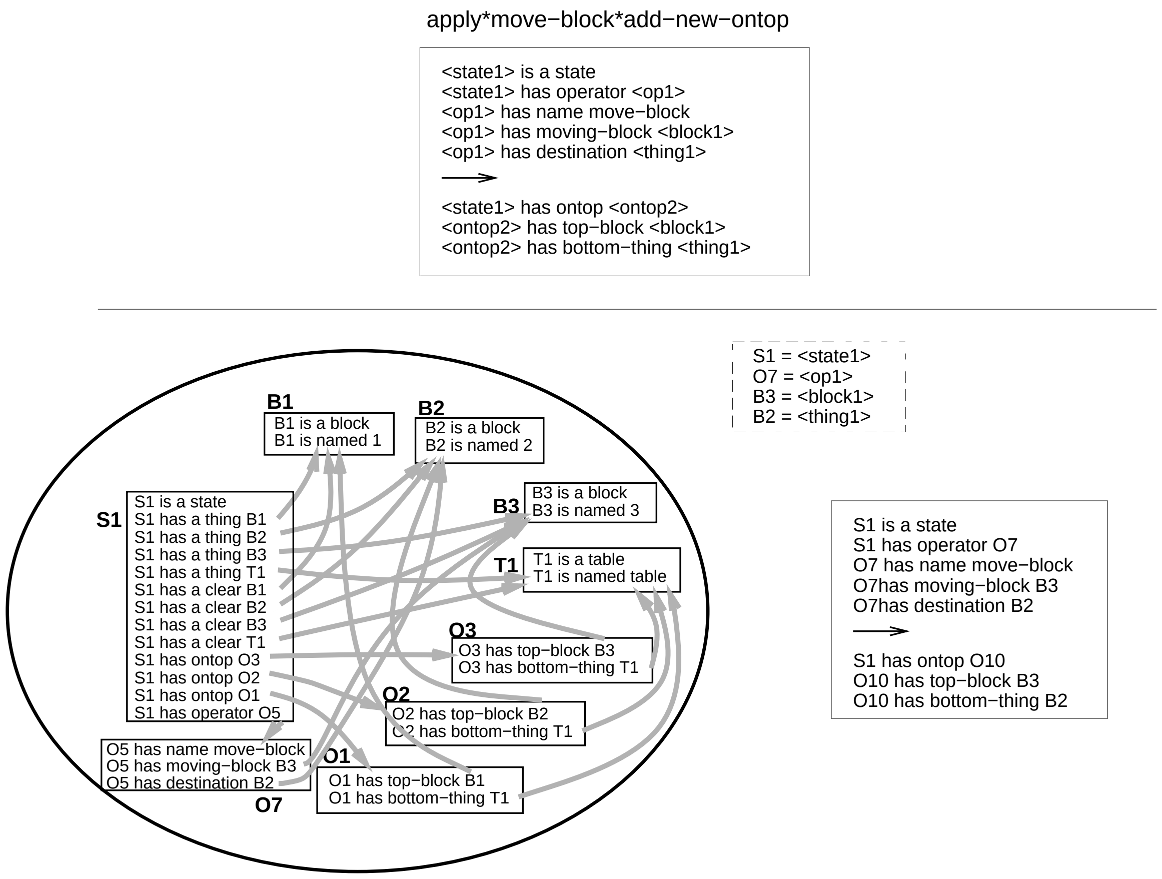


Figure 29: Answer to exercise from section 3. Please note that because Soar automatically generates the identifiers for the operators, it could generate different ones (e.g. instead of `O14`).



apply*move−block*add−new−ontop

```
<state1> is a state
<state1> has operator <op1>
<op1> has name move−block
<op1> has moving−block <block1>
<op1> has destination <thing1>
⟶
<state1> has ontop <ontop2>
<ontop2> has top−block <block1>
<ontop2> has bottom−thing <thing1>
```

```
S1 = <state1>
O7 = <op1>
B3 = <block1>
B2 = <thing1>
```

```
S1 is a state
S1 has operator O7
O7 has name move−block
O7has moving−block B3
O7has destination B2
⟶
S1 has ontop O10
O10 has top−block B3
O10 has bottom−thing B2
```

Figure 30: Answer to exercise from section 4. Please note that because Soar automatically generates identifiers, the new `ontop` relation could have a different identifier than `O10`.

Psuedo−production
apply*move−block*add−new−on

```
<state1> is a state
<state1> has operator <op1>
<op1> has name move−block
<op1> has moving−block <block1>
<op1> has destination <thing2>
⟶
<state1> has ontop <ontop> with
  acceptable and parallel preferences
<ontop> has top−block <block1>
<ontop> has bottom−thing <thing2>
```

Preferences

```
S1 ^ontop O3 + &
O3 ^top−block B2 +
O3 ^bottom−thing T1 +
```

Figure 31: Answer to exercise from section 5

Answers for section 6:

1. Persistent
   Preferences created by operator applications are, be Soar's definition, persistent.

2. Elaborations
   You checked the support of these relations in the example in this section. Remember, for a production to create a persistent result, it must test the current operator.

3. They are elaborations because they only test the state.
   If a block is no longer clear, an operator proposal to move that block should automatically be retracted.

4. O-supported WMEs are removed when another operator application explicitly removes them (by issuing a reject preference).

5. I-supported WMEs are removed when the instantiating production is retracted. A production is retracted when the conditions of the instantiation are no longer satisfied.

Translation

<state1> is a state
<state1> has an operator <op1>
<op1> has name move−block
<op1> has moving−block <block1>
<op1> has destination <thing2>
$\longrightarrow$
<state1> has ontop <ontop> with
    accept and parallel preferences
<ontop> has ^top−block <block1>
<ontop> has ^bottom−thing <thing2>

Soar Production

```
sp {apply*move-block*add-new-on
    (state <state1> ^operator <op1>)
    (<op1> ^name move-block
           ^moving-block <block1>
           ^destination <thing2>)
    -->
    (<state1> ^ontop <ontop> + &)
    (<ontop> ^top-block <block1>
             ^bottom-block <thing2> )}
```

Figure 32: Answer to exercise from section 7

Answer to exercise in section 13.5.1

This little bit of Soar code creates a `state no-change` (by doing nothing), and then an operator in the subgoal. Then it returns a result, `bar`. When `bar` is on the superstate, an operator is proposed, popping the subgoal. At this point, the O-support for `bar` disappears, and so the operator proposal production `baz` is retracted. But the operator selection persists. Note that because the subgoal never checks any attributes of the superstate, there are no grounds for a justification, and so one is not created.

```
sp {foo
    (state <s> ^superstate <ss>
       ^impasse no-change)
    -->
    (<s> ^operator <o>)
    (<o> ^name foo)}

sp {bar
    (state <s> ^superstate <ss>
               ^operator.name foo)
    -->
    (<ss> ^bar bar)}

sp {baz
    (state <s> ^superstate nil
               ^bar <b>)
    -->
    (<s> ^operator <o>)}
```

References

[Congdon1996] Clare Congdon. Soar user's manual. TR, August 1996.

[Laird *et al.*1987] John E. Laird, Allen Newell, and Paul S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33:1–64, 1987.

[Newell1990] Allen Newell. *Unified Theories of Cognition.* Harvard University Press, 1990.

[Ousterhout1994] J. K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[Welch1995] B. Welch. *Practical Programming in Tcl Tk.* Prentice Hall, 1995.

Index