Expert-System Development in Soar: A Tutorial

Erik Altmann and Gregg R. Yost

June 1992
CMU-CS-92-151

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

* Digital Equipment Corporation
111 Locke Drive (LM02/K11)
Marlboro, MA 01752

Abstract

This is a tutorial for building an expert system in Soar using the TAQL programming language. It provides a self-contained reference for the end-to-end development of a Soar system that accomplishes a particular task. It presents a natural-language task description, a system design, and a sample implementation, including a documented code listing. It also discusses chunking (Soar's learning mechanism) in the context of the sample implementation.

This document has been approved for public release and its distribution is unlimited.

This research was sponsored in part by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597. The research was also supported in part by Digital Equipment Corporation and the Natural Sciences and Engineering Research Council of Canada. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, DEC, the Government of Canada, or the US Government.
# Table of Contents

1. Introduction 1
2. The Shipment Scheduling Assistant: Task Description 3
   2.1. Description 3
      2.1.1. Drivers, trucks, cities, and highways 3
      2.1.2. Constraints 4
   2.2. Test case 5
3. Designing a Soar Solution 7
   3.1. Introduction 7
   3.2. Notes on the Test Case 8
   3.3. Problem Space Design 9
      3.3.1. Task analysis 9
      3.3.2. Dynamic behavior 10
      3.3.3. Static structure 12
      3.3.4. A detailed design 16
   3.4. Notes on Chunking 20
      3.4.1. Correctness, generality, and backtracing 20
      3.4.2. Preventing incorrect chunks 21
      3.4.3. Expensive chunks 25
4. Sample Implementation in Soar 27
   4.1. Introduction 27
   4.2. Program Listing 27
   4.3. Execution Trace 43
   4.4. Chunk Listing 49
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Procedure that solves the shipment scheduling task</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>Trace of lookahead search</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>Static problem-space structure</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>Pseudo-trace corresponding to Figures 2 and 3</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>Simple backtracing example</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>Novalue: backtracking without chunking</td>
<td>21</td>
</tr>
<tr>
<td>7</td>
<td>Conditions that bind constants</td>
<td>22</td>
</tr>
<tr>
<td>8</td>
<td>Lookahead leading to search-control chunk p946</td>
<td>23</td>
</tr>
<tr>
<td>9</td>
<td>The search-control chunk p946</td>
<td>23</td>
</tr>
<tr>
<td>10</td>
<td>Backtracing to the trip3 condition</td>
<td>24</td>
</tr>
<tr>
<td>11</td>
<td>Conditions that prevent overgeneral search-control chunks</td>
<td>25</td>
</tr>
<tr>
<td>12</td>
<td>An expensive chunk</td>
<td>26</td>
</tr>
</tbody>
</table>
1. Introduction

This is a tutorial for building a learning expert system in the Soar problem solving architecture (Laird et al., 1990), using the TAQL programming language (Yost and Altmann, 1991; Yost, 1992). It provides a self-contained reference for end-to-end development of a Soar system, beginning with a natural-language task description and ending with a sample implementation of a system to accomplish the task. In between it illustrates a number of design and programming techniques (without claims to complete coverage of such techniques, or independence from personal style). It also discusses chunking (Soar's learning mechanism) in the context of the sample implementation.

Chapter 2 describes the task of the shipment scheduling assistant, which is to generate schedules that coordinate trucks, truck drivers, and shipments. Chapter 3 presents a design that outlines both the dynamic behavior of a system and the static relationship of its components. The last section of this chapter presents a detailed view of chunks learned by a sample implementation of this design. Chapter 4 presents the details of this implementation, including a documented code listing, a trace of the running system, and a listing of the chunks learned during the traced run.

For information about obtaining the code and other listings on-line, or to obtain this or other documents concerning Soar and TAQL, send electronic mail to soar-requests@cs.cmu.edu, or physical mail to The Soar Project, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3890.
2. The Shipment Scheduling Assistant: Task Description

This description was derived from a formal description of a larger set of problems in the shipment scheduling domain (Filman, 1988a). The formal description is for the problem used as an example in a recent expert systems paper (Filman, 1988b).

2.1. Description

The Big Giant Trucking Company ships materials among cities in the Midwest. Customers contact the company and request that goods be transported from one city to another on a specified day. Big Giant's dispatchers collect the orders and create suitable delivery schedules.

A schedule consists of some number of trips, where each trip has an itinerary, a truck, and a driver. An itinerary is a list of cities and the highways that the driver should take from one city to the next. The itinerary also states what shipments, if any, the driver should pick up and deliver at each city (sometimes the driver will just pass through with no pickups or deliveries).

Producing a schedule is difficult because of the many constraints on the trips and the schedule as a whole. For example, each driver can drive only one trip, and union drivers can only drive trips that take less than eleven hours. The shipment scheduling assistant takes the set of itineraries on the schedule (which we assume were put together by a dispatcher), and tries to find an assignment of trucks and drivers to the itineraries that does not violate any constraints. If no such assignment is possible, the assistant informs the dispatcher, who must then revise the itineraries and try again.

The remainder of this description provides the information the assistant needs to try to find valid truck and driver assignments.

2.1.1. Drivers, trucks, cities, and highways

Tables 1, 2, and 5 define the driver, trucks, and highways used by Big Giant.

Big Giant serves the following cities:

- **In Illinois**: La Harpe, Oregon, Thayer, Utica, Viola, Yale, and Zion.


<table>
<thead>
<tr>
<th>Name</th>
<th>Union</th>
<th>License Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brown</td>
<td>yes</td>
<td>3</td>
</tr>
<tr>
<td>Gray</td>
<td>no</td>
<td>1</td>
</tr>
<tr>
<td>Green</td>
<td>yes</td>
<td>3</td>
</tr>
<tr>
<td>White</td>
<td>no</td>
<td>2</td>
</tr>
</tbody>
</table>

**Table 1: Drivers**
2.1.2. Constraints
The constraints on schedules and trips are:

- Each driver can drive only one trip, and each trip has only one driver.

- Each truck can be used on only one trip, and each trip has only one truck.

- The maximum weight of a truck’s load at any point during a trip cannot exceed the truck’s rated weight limit. Big trucks can hold 32000 pounds, medium trucks 10000 pounds, and small trucks 5000 pounds.

- The maximum volume of a truck’s load at any point during a trip cannot exceed the truck’s rated volume limit. Big trucks can hold 1280 cubic feet, medium trucks 640 cubic feet, and small trucks 400 cubic feet.

- The driver and truck assigned to a trip must be in the trip’s origin city to begin with.

- The license class of a driver must be at least as great as the license class required by the truck he or she is assigned. Big trucks require class 3 licenses, medium trucks require at least class 2 licenses, and small trucks require at least class 1 licenses.

- A driver can only drive trips whose duration is less than his or her maximum allowable driving time. The duration of a trip is the sum of the driving times for each segment on the itinerary, plus the time needed for loading and unloading shipments during the trip. The driving time for a segment is the length of the road used for that segment divided by the estimated speed for that road (as determined by the weather and road grade, see Table 3). Union drivers can be on a trip for at most 11 hours, while non-union drivers can be on a trip for at most 12.5 hours.

- White cannot drive on any trip that passes through a city in Illinois (he is wanted for a crime there).

<table>
<thead>
<tr>
<th>Name</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cannonball</td>
<td>big</td>
</tr>
<tr>
<td>Piper</td>
<td>small</td>
</tr>
<tr>
<td>Queen Bee</td>
<td>medium</td>
</tr>
<tr>
<td>Traveler</td>
<td>medium</td>
</tr>
</tbody>
</table>

Table 2: Trucks

<table>
<thead>
<tr>
<th>Road grade →</th>
<th>Primary</th>
<th>Secondary</th>
<th>Tertiary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weather ↓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fair</td>
<td>60</td>
<td>55</td>
<td>50</td>
</tr>
<tr>
<td>Rain</td>
<td>55</td>
<td>50</td>
<td>35</td>
</tr>
<tr>
<td>Snow</td>
<td>45</td>
<td>40</td>
<td>30</td>
</tr>
</tbody>
</table>

Table 3: Estimated travel speed, given road grade and weather
2.2. Test case

This section presents a simple test case that you can use to partially test your scheduling assistant.

The weather throughout Big Giant's area of operations is rainy. Drivers Brown and Gray are in Gary, and drivers Green and White are in Indianapolis. Trucks Piper and Traveler are in Gary, and trucks Cannonball and Queen Bee are in Indianapolis.

The dispatcher's schedule has three trips. The shipments referred to in the trips are listed in Table 4.

- **Trip 1:** Starting in Gary, pick up the typewriter shipment and take highway U30 to Warsaw, followed by U31a to Kokomo, U31b to Indianapolis, and I74 to Attica. Deliver the typewriter shipment in Attica.

- **Trip 2:** Starting in Gary, take U41a to Cook, I80b to Utica, and I80a to Viola. Pick up the carpet shipment in Viola. Then, take I80a back to Utica, and I80b to Cook. In Cook, deliver the carpet shipment and pick up the newsprint shipment.

- **Trip 3:** Starting in Indianapolis, take I70b to Yale, then take U41e to Attica.

One valid solution for this test case is to assign Gray/Piper to trip 1, Brown/Traveler to trip 2, and Green/Cannonball to trip 3. Piper is the only truck in Gary that Gray is licensed to drive, leaving Traveler (the only other truck in Gary) for Brown. White cannot drive trip 3, because it passes through Illinois, so Green must do it.

<table>
<thead>
<tr>
<th>Material</th>
<th>Origin</th>
<th>Destination</th>
<th>Weight</th>
<th>Volume</th>
<th>Loading Time</th>
<th>Unloading Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bicycles</td>
<td>Roselawn</td>
<td>Bloomington</td>
<td>500.0</td>
<td>100.0</td>
<td>0.2</td>
<td>0.25</td>
</tr>
<tr>
<td>Books</td>
<td>Oregon</td>
<td>Mitchell</td>
<td>1000.0</td>
<td>50.0</td>
<td>0.2</td>
<td>0.25</td>
</tr>
<tr>
<td>Carpet</td>
<td>Viola</td>
<td>Cook</td>
<td>500.0</td>
<td>100.0</td>
<td>0.2</td>
<td>0.25</td>
</tr>
<tr>
<td>Computers</td>
<td>Seymour</td>
<td>Thayer</td>
<td>1000.0</td>
<td>150.0</td>
<td>0.2</td>
<td>0.25</td>
</tr>
<tr>
<td>Newsprint</td>
<td>Cook</td>
<td>Indianapolis</td>
<td>6000.0</td>
<td>400.0</td>
<td>0.2</td>
<td>0.25</td>
</tr>
<tr>
<td>Refrigerators</td>
<td>Kokomo</td>
<td>Warsaw</td>
<td>9000.0</td>
<td>600.0</td>
<td>0.2</td>
<td>0.25</td>
</tr>
<tr>
<td>Toys</td>
<td>La Harpe</td>
<td>Oregon</td>
<td>1000.0</td>
<td>100.0</td>
<td>0.2</td>
<td>0.25</td>
</tr>
<tr>
<td>Typewriters</td>
<td>Gary</td>
<td>Attica</td>
<td>1000.0</td>
<td>200.0</td>
<td>0.2</td>
<td>0.25</td>
</tr>
</tbody>
</table>

**Table 4:** Shipments
<table>
<thead>
<tr>
<th>Name</th>
<th>Connects</th>
<th>Grade</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>I64a</td>
<td>New Harmony, Jasper</td>
<td>primary</td>
<td>60.0</td>
</tr>
<tr>
<td>I64b</td>
<td>English, Jasper</td>
<td>tertiary</td>
<td>30.0</td>
</tr>
<tr>
<td>I65</td>
<td>Seymour, Indianapolis</td>
<td>primary</td>
<td>60.0</td>
</tr>
<tr>
<td>I70a</td>
<td>Thayer, Yale</td>
<td>primary</td>
<td>150.0</td>
</tr>
<tr>
<td>I70b</td>
<td>Indianapolis, Yale</td>
<td>primary</td>
<td>90.0</td>
</tr>
<tr>
<td>I74</td>
<td>Indianapolis, Attica</td>
<td>primary</td>
<td>60.0</td>
</tr>
<tr>
<td>I80a</td>
<td>Viola, Utica</td>
<td>primary</td>
<td>100.0</td>
</tr>
<tr>
<td>I80b</td>
<td>Cook, Utica</td>
<td>primary</td>
<td>90.0</td>
</tr>
<tr>
<td>I90</td>
<td>Oregon, Gary</td>
<td>secondary</td>
<td>100.0</td>
</tr>
<tr>
<td>I94</td>
<td>Zion, Gary</td>
<td>primary</td>
<td>60.0</td>
</tr>
<tr>
<td>S125</td>
<td>La Harpe, Thayer</td>
<td>tertiary</td>
<td>80.0</td>
</tr>
<tr>
<td>S25</td>
<td>Delphi, Attica</td>
<td>tertiary</td>
<td>40.0</td>
</tr>
<tr>
<td>S26</td>
<td>Delphi, Kokomo</td>
<td>tertiary</td>
<td>30.0</td>
</tr>
<tr>
<td>S37a</td>
<td>Bloomington, Indianapolis</td>
<td>secondary</td>
<td>50.0</td>
</tr>
<tr>
<td>S37b</td>
<td>Bloomington, Mitchell</td>
<td>tertiary</td>
<td>30.0</td>
</tr>
<tr>
<td>S37c</td>
<td>Paoli, Mitchell</td>
<td>tertiary</td>
<td>10.0</td>
</tr>
<tr>
<td>S37d</td>
<td>Paoli, English</td>
<td>tertiary</td>
<td>10.0</td>
</tr>
<tr>
<td>U231</td>
<td>Cook, Hebron</td>
<td>tertiary</td>
<td>20.0</td>
</tr>
<tr>
<td>U24</td>
<td>La Harpe, Fowler</td>
<td>secondary</td>
<td>180.0</td>
</tr>
<tr>
<td>U30</td>
<td>Gary, Warsaw</td>
<td>secondary</td>
<td>70.0</td>
</tr>
<tr>
<td>U31a</td>
<td>Kokomo, Warsaw</td>
<td>secondary</td>
<td>70.0</td>
</tr>
<tr>
<td>U31b</td>
<td>Kokomo, Indianapolis</td>
<td>primary</td>
<td>40.0</td>
</tr>
<tr>
<td>U41a</td>
<td>Cook, Gary</td>
<td>secondary</td>
<td>20.0</td>
</tr>
<tr>
<td>U41b</td>
<td>Cook, Roselawn</td>
<td>secondary</td>
<td>20.0</td>
</tr>
<tr>
<td>U41c</td>
<td>Fowler, Roselawn</td>
<td>secondary</td>
<td>30.0</td>
</tr>
<tr>
<td>U41d</td>
<td>Fowler, Attica</td>
<td>secondary</td>
<td>30.0</td>
</tr>
<tr>
<td>U41e</td>
<td>Yale, Attica</td>
<td>secondary</td>
<td>90.0</td>
</tr>
<tr>
<td>U41f</td>
<td>Yale, New Harmony</td>
<td>secondary</td>
<td>70.0</td>
</tr>
<tr>
<td>U50</td>
<td>Mitchell, Seymour</td>
<td>tertiary</td>
<td>30.0</td>
</tr>
<tr>
<td>U51</td>
<td>Utica, Oregon</td>
<td>secondary</td>
<td>40.0</td>
</tr>
<tr>
<td>U67</td>
<td>Viola, La Harpe</td>
<td>secondary</td>
<td>50.0</td>
</tr>
</tbody>
</table>

**Table 5: Highways**
3. Designing a Soar Solution

3.1. Introduction

This chapter discusses how to design a Soar system that solves the shipment scheduling task. The guidance toward a particular implementation increases as the chapter progresses, allowing novice Soar users to choose a stepping-off point, or to stick with the chapter to the end to gain exposure to one particular set of design and programming methods. The particular implementation, together with a trace and a chunk listing, is given in Chapter 4.

Section 3.2 presents a one-page analysis of the test case from Chapter 2, including a summary of the various constraints on the solution.

Section 3.3 begins with an analysis of these constraints and a procedure for solving the task. It then outlines the dynamic behavior and static structure of a Soar system. The dynamic and static designs are depicted graphically, and then tied together in a Soar pseudo-trace. The last subsection presents a detailed design, in textual form.

Section 3.4 introduces chunking, backtracing, and the problem of learning from exhaustion. It also discusses methods of manipulating data to prevent overgeneral chunks, including a case-study of how certain conditions come to be included in a particular chunk.
3.2. Notes on the Test Case

The description of the three trips in the test case (Section 2.2, page 5) is reproduced below. Relevant notes, compiled from the various tables, are made under each trip. A summary of constraints, and a specification for the solution, are given afterwards.

- **Trip 1**: Starting in Gary, pick up the typewriter shipment and take highway U30 to Warsaw, followed by U31a to Kokomo, U31b to Indianapolis, and I74 to Attica. Deliver the typewriter shipment in Attica.

  *Notes*: There are four segments. The *time* needed to drive them, and to load and unload the single shipment, adds up to 3.7 hours, so any driver will do. Similarly, the *size* of the carpet shipment (volume 200, weight 1000) provides no constraint: any truck will do.

  *Drivers* available are Gray and Brown, and *trucks* available are Piper (small) and Traveler (medium). Possible assignments: Gray/Piper, Brown/Traveler, Brown/Piper. (Gray is licensed only for Piper.)

  Brown/Piper will cause the schedule to fail, because Trip 2 also starts in Gary and Gray/Traveler is illegal.

- **Trip 2**: Starting in Gary, take U41a to Cook, I80b to Utica, and I80a to Viola. Pick up the carpet shipment in Viola. Then, take I80a back to Utica, and I80b to Cook. In Cook, deliver the carpet shipment and pick up the newsprint shipment.

  *Notes*: There are six segments, counting loading the newsprint as the beginning of a segment that is otherwise null. Drive *time* adds up to 6.5 hours, so any driver will do. The *size* of the shipments provides one constraint: the *weight* of the newsprint shipment (6000) requires a medium truck (Traveler).

  *Drivers* available are Gray and Brown, and *trucks* available are Piper (small) and Traveler (medium). Possible assignments: Gray/Piper, Brown/Traveler, Brown/Piper. (Gray is licensed only for Piper.)

  Any assignment involving Piper will fail on the last segments because Piper’s weight limit will not accommodate the newspaper shipment.

- **Trip 3**: Starting in Indianapolis, take I70b to Yale, then take U41e to Attica.

  *Notes*: There are two segments. The *time* needed to drive them is 3.4 hours, so any driver will do. There are no shipments, so *size* provides no constraint.

  *Drivers* available are Green and White, and *trucks* available are Queen Bee and Cannonball. Possible assignments: all four pairs.

  Any assignment involving White will fail on the first segment because Yale is in Illinois.

Global constraints: &lt;Brown, Piper, Trip 1&gt; means Trip 2 cannot be covered.

Local constraints: &lt;X, Piper, Trip 2&gt; will fail because Piper is too small.
&lt;White, X, Trip 3&gt; will fail because White will be arrested in Yale.

Solution: &lt;Gray, Piper, Trip 1&gt;, &lt;Brown, Traveler, Trip 2&gt;, &lt;Green, X, Trip 3&gt;
3.3. Problem Space Design

3.3.1. Task analysis
A first step in developing a solution is to look at the source of difficulty in the task, which in this case is the set of constraints listed on page 4.

The constraints fall into three categories:

1. **Immediate** constraints — These can be checked immediately, based on the information in the tables given with the task description. For instance, a series of table lookups can tell us that a particular driver is in the city in which a trip originates, or is licensed to drive a particular class of truck.

2. **Local** constraints — These must be met for a particular assignment of a truck and a driver to a trip to be successful (the constraints are local to a particular assignment).

Checking local constraints requires some computation beyond looking up information in the tables. For instance, the load in the truck at any one time is a function of how the shipments are spread out across the trip, and checking that the truck's capacity is not exceeded requires simulating the effects of loading, unloading, and driving.

Violations of local constraints surface as the inability to complete a particular trip.

3. **Global** constraints — These must hold across assignments for the entire schedule to be successful (they are global with respect to particular assignments).

For instance, consider the possibility of assigning Brown to Piper for Trip 1. This assignment is successful, but leaves no remaining assignments for Trip 2, because Gray is not licensed for Traveler. Thus the schedule as a whole will fail if it includes the first assignment.

Global constraint violations surface as a situation in which there are trips left over but no assignments for them.

These categories help to outline a procedure for solving the task, which is shown in Figure 1. Step 1 generates assignments that satisfy the immediate constraints. Steps 2 and 3 check the local and global constraints. Step 4 detects when the task is solved, and Step 5 recursively invokes the procedure when it is not.

The procedure in Figure 1 can be mapped onto problem spaces as follows. If we associate each assignment with an operator, each such operator presents an opportunity to check the local constraints associated with that assignment. The effect of such an operator would be to simulate a trip with a particular truck and driver, and generate a state in which either the schedule is updated with that assignment, or in which a failure of the assignment (due to local constraints being violated) is indicated. We can call this the *simulate-trip* operator. The operator itself requires a sequence of steps, namely simulating the loading and unloading of shipments and the driving of segments of the trip. Simulate-trip can be applied by a separate problem space that executes these steps.

The immediate constraints can be incorporated in the proposal of simulate-trip operators. This
1. Generate a set of possible assignments for unassigned trips that satisfy all immediate constraints, and choose one assignment.

2. Check the local constraints on that assignment, by simulating the trip. If a constraint fails, backtrack to some previous instance of Step 1 where alternatives remain, and choose one. If there are no previous instances where alternatives remain, halt with failure.

3. If no local constraints are violated, check whether any global constraints have been violated by checking if there are trips left over but no assignments for them. If a global constraint has been violated, backtrack to some previous instance of Step 1 or halt with failure (as in Step 2).

4. If no local or global constraints are violated, check whether we have a successful schedule, by checking whether all the trips have assignments. If so, halt with success.

5. Mark the trip as having been assigned, update the pool of available trucks and drivers, and recurse to Step 1.

Figure 1: Procedure that solves the shipment scheduling task

will reduce the number of operators generated (and therefore the search involved in solving the task) Further reduction in search comes if we base successive operator proposals on the updated set of trips and the updated pool of available trucks and drivers (all of which shrink with each successful assignment). These updates can be carried out by simulate-trip itself, upon successful completion of a simulation. This updating also allows testing for success by detecting when no more unassigned trips remain, and testing for violations of global constraints by detecting when trips are left over but no assignments are left.

The states in this scheme will have to represent several kinds of information. First is that from the tables presented in Chapter 2. This information implicitly represents many of the immediate constraints, and therefore must be available when proposing simulate-trip operators. Second is the dynamic information that changes from trip to trip. This includes the sets of unassigned trips, available drivers, and available trucks; the schedule, as it grows with each successful assignment; and indications of local and global constraint violations. Third is the dynamic information that changes only from segment to segment within a trip. This includes resources such as available drive time and truck capacity.

Finally, we need a mechanism for backtracking when constraints fail. One simple mechanism is depth-first search, in which the selection at the most recent instance of Step 1 is changed, unless there are no more alternatives, at which point the procedure backtracks to the next most recent instance of Step 1.

3.3.2. Dynamic behavior

Figure 2 shows an example trace of the procedure above as mapped onto problem spaces, with an emphasis on how depth-first search works in Soar. Spaces are depicted as triangles, states as circles, operators as lines emanating from states, impasses as down arrows, and subgoal results are up arrows. The trace is annotated with examples of how each step of the procedure is realized as a problem-space operation. (Figure 3 shows a corresponding pseudo-trace that resembles actual Soar output.) The problem spaces shown are the task space, which contains the
simulate-trip operator, and the selection space, which implements depth-first search (usually referred to as lookahead search in the context of Soar).

In the top instance of the task space (T1), a set of simulate-trip operators is proposed (per Step 1). These tie, resulting in an impasse. The selection space (S1) is selected automatically for the subgoal (by Soar's default productions). In the selection space, the default productions create an evaluate-object operator for each tied alternative. The evaluate-object operators are given indifferent preferences, so one is selected (at random or in a prescribed pattern; see the Soar user-select command). Evaluate-object itself impasses, and in the resulting subgoal the default productions cause the task space to be selected (T2). This subgoal is referred to as the evaluation subgoal. The operator being evaluated is then selected automatically, under the assumption that it might generate a state that either succeeds or fails (or might otherwise yield information that allows the operator to be evaluated). This automatically-selected operator is referred to as the lookahead operator.

In the left-most evaluation subgoal (T2), the lookahead operator causes a local constraint violation. (The space that applies simulate-trip is omitted, but see Figure 3.) The violation is detected (per Step 2), and results in a failure evaluation for evaluate-object. T2 exits, and a second evaluate-object operator is selected.

![Figure 2: Trace of lookahead search](image-url)
The lookahead operator in T3 yields neither constraint violations nor success. A new set of simulate-trip operators is proposed (based on the updated sets of unassigned trips and available trucks and drivers). The resulting tie leads to a new instance of the selection space (S2). Below S2, in T4, the simulation leads to a global constraint violation (per Step 3). As with a local constraint violation, this generates a failure evaluation, causing T4 to exit and another alternative to be evaluated.

In T5, the lookahead operator violates no local constraints, and leads to a state in which only one new simulate-trip operator is proposed. That operator is selected, and results in a state in which no trips are left. The space exits with a success evaluation (per Step 4). The default productions propagate success up to T1 (and would propagate it further if there were more levels). This breaks the tie in T1, with the lookahead operator from T3 being selected. If chunking was turned on, the success of the lookahead operator in T5 will have been learned as a chunk that selects that operator after the first selection in T1, leading directly to a successful state in T1.

For further details on lookahead search, consult the Soar manual. The depth-first behavior described here is only one of many weak methods that can arise from the universal weak method (Laird, 1984), depending on how much and what kinds of evaluation knowledge are available. For example, hill-climbing arises if any state can be evaluated, and not just those that represent failures or successes.

3.3.3. Static structure

The previous section outlined a procedure for solving the shipment scheduling task, and described at a high level how that procedure could be cast in terms of states and operators. This section describes in more detail a set of Soar problem spaces for performing the task. The first part of the description is a diagram (Figure 3) that shows the static problem space structure, and the second part (beginning on page 16) is a textual specification. Both the diagram and the specification correspond directly to the code presented in Chapter 4, so ambiguities and other confusions can be resolved by looking there.

Figure 3 shows the operators in each space, and the information that flows between spaces. The name of each space appears at its top right-hand corner, and the operators appear in the interior of the triangle. The lines connecting spaces denote impasses.

The remainder of this section traces through the figure, showing how the spaces work together. Note that the diagram represents the static structure of the system. This explains how the task space can be connected to two superspaces: it responds to impasses in both, at different times. All the impasses represented in the diagram are operator no-changes; the diagram omits the operator tie impasse that occurs in the task space (on simulate-trip operators), which leads to the selection space. This interaction is described in Figure 2.

The do-task operator in the top space is applied in the task space. To carry out the task, operators in the task space need access to the information from the tables, and initial values for the trip, truck, and driver sets. When the task space exits, it returns the complete schedule to the top space as the solution to the task.
The simulate-trip operator in the task space is applied in the simulate-trip space. Simulate-trip takes the particular assignment and simulates the trip. As part of its result, the space updates the trip, truck, and driver sets by removing those involved in the trip it just simulated. It also returns an indication of whether any local constraints were violated during the simulation. If not, it returns the schedule as updated with the addition of the trip just simulated.

If a simulation generates a local constraint violation, the task space passes the violation signal directly up to the selection space above, if the task space occurs in an evaluation subgoal. The
default productions in the selection space act on the failure by terminating the evaluate-object operator and selecting a new one. However, if the simulation is successful, a new simulate-trip operator is selected in the task space.

If after a simulation the task space finds that all trips have been assigned to, the schedule-succeeds operator is proposed. This returns a success signal to the selection space if this is an evaluation subgoal, or the complete schedule to the top space. If, on the other hand, there comes a point where there are still trips to be assigned to but no simulate-trip operators proposed, then the global-constraint-failed operator is selected. This returns a failure signal to the selection space.

In the simulate-trip space, the simulate-segment operator is selected once for each segment of the trip. If after any segment a resource (such as truck capacity) is found to have run out, the local-constraint-failed operator is selected. This operator returns a failure signal to the task space, causing the simulate-trip operator to terminate (thereby terminating the simulate-trip subgoal). The task space then terminates as well, as described above. If, on the other hand, all the segments are simulated with no violations, the trip-succeeds operator is selected. This operator updates the trip, truck, and driver sets in the task space, and returns the updated schedule.

The simulate segment operator, when selected, impasses, because simulating a segment involves several operations. These occur in the simulate-segment space. The operations for each segment consist of loading any shipments at the beginning of the segment, followed by the drive to the next location, followed by unloading shipments to be delivered at that location. Loading and unloading affect the volume and weight capacities remaining for that trip, and driving affects the time the driver has been on the road. When the segment is complete, the compute-resources operator computes the remaining resources and checks if any have run out, and if so signals a local constraint violation on the superstate. If not, the update-resources operator updates the resources on the superstate, in preparation for the next segment.
The Soar pseudo-trace in Figure 4 brings together both static and dynamic aspects of the system. The labels for task and selection spaces from Figure 2 appear along the right margin. The space and operator names are carried over from Figure 3. The impasse types are also given.

G: g1
P: p1 (top-space)
S: s1
O: o1 (do-task)

=>G: g2 (operator no-change)
P: p2 (task)
S: s2

=>G: g3 (operator tie)
P: p3 (selection)
S: s3
O: o3 (evaluate-object (simulate-trip ((gray) (piper) (trip2))))

=>G: g4 (operator no-change)
P: p2 (task)
S: d1 (duplicates s3)
O: o1 (simulate-trip (gray) (piper) (trip2))

=>G: g5 (operator no-change)
P: p4 (simulate-trip)
S: s4
O: o4 (simulate-segment segment1)

...intermediate segments

O: o20 (simulate-segment segment6)

=>G: g21 (operator no-change)
P: p21 (simulate-segment)
S: s21
O: o21 (load (newsprint 6000.0))
O: o22 (drive)
O: o23 (compute-resources)
O: o24 (update-resources)

O: o25 (local-constraint-failure weight-limit-exceeded)
Evaluation of operator c1 (simulate-trip) is failure
O: o26 (evaluate-object (simulate-trip ((green) (cannonball) (trip3))))

=>G: g22 (operator no-change)
P: p2 (task)
S: d2 (duplicates s3)
O: o2 (simulate-trip (green) (cannonball) (trip3))

...simulation is successful

O: o27 (trip-succeeds)

=>G: g23 (operator tie)
P: p23 (selection)
S: s22
O: o28 (evaluate-object (simulate-trip ((brown) (piper) (trip1))))

=>G: g24 (operator no-change)
P: p2 (task)
S: d3 (duplicates d2)
O: c3 (simulate-trip (brown) (piper) (trip1))

...intermediate trips

O: o40 (global-constraint-violation trips-left-over)
Evaluation of operator c3 (simulate-trip) is failure
O: o41 (evaluate-object (simulate-trip ((gray) (piper) (trip1))))

=>G: g25 (operator no-change)
P: p2 (task)
S: d4 (duplicates d2)
O: c4 (simulate-trip (gray) (piper) (trip1))

...intermediate trips

O: o50 (simulate-trip (brown) (traveler) (trip2))

O: o70 (schedule-succeeds)
Evaluation of operator c4 (simulate-trip) is success

...success propagates up to T1

Figure 4: Pseudo-trace corresponding to Figures 2 and 3
3.3.4. A detailed design

This section specifies the operators in more detail, giving their arguments (in parentheses) and their semantics in terms of conditions, effects, and termination conditions. Where appropriate, information about state contents and operator search control is also given.

- **top-space**

  State (also called the *top state*): Contains the invariant information from the tables, which is used to initialize dynamic values like the trip, truck, and driver sets.

  Operators:
  1. **do-task**

     Conditions: Proposed without conditions.

     Also called the *task operator*. Applied in the task space.

     Effects: Adds the completed schedule to the state.

     Termination: Automatic, when a successful or failed state is reached in the subgoal that applies it.

---

1. The space itself is provided by TAQL by default.

2. The effects of an operator applied in a subspace are carried out by the subspace.

3. This termination is provided by TAQL’s runtime support.
- **task**

  State: Contains trip, truck, and driver sets, either initialized from the top state (when below the top space), or duplicated from another instance of the task space (when below the selection space).

  Operators:

  1. **simulate-trip** (trip, truck, driver)

     Conditions: Reads the current state’s trip, truck, and driver sets. Checks that the assignment satisfies the immediate constraints that the driver and the truck be in the city where the trip begins, and that the driver is licensed for the truck.

     Applied in the simulate-trip space, which checks local constraints by simulating the trip with that truck and driver.

     Effects: Updates the trip, truck, and driver sets by removing its arguments from them. Updates the current schedule if no local constraints are violated.

     Termination: Terminates (1) if the subspace updates the current schedule with the simulated trip, or (2) if the subspace signals a local constraint violation.

  2. **schedule-succeeds** (complete-schedule)

     Conditions: Proposed when the trip set is empty.

     Effects: Signals success in the superspace, either via evaluation knowledge (during lookahead search) or the goal test (when the superspace is the top space). If the superspace is the top space, returns complete-schedule.

     Termination: By higher decision.

  3. **global-constraint-failed**

     Conditions: Proposed without conditions, but made worst. Consequently, selected only when there are trips left over (inhibiting schedule-succeeds) but no simulate-trip operators are proposed. This condition implies a global constraint violation.

     Effects: Signals failure in the superspace, either through evaluation knowledge (during lookahead search) or the goal test (when the superspace is the top space).

     Termination: By higher decision.

---

4When information is returned that resolves an impasse in the supercontext, Soar terminates all lower selections.
• **simulate-trip**

State: Contains resources (initially the time for which the driver can be on the road, and the weight limit and volume of the truck), and the current segment of the trip being simulated. Initialized from the superoperator, and updated by simulate-segment.

Operators:

1. **simulate-segment** (segment, resources)
   
   Conditions: Reads the segment and resources from the state.
   
   Applied in the simulate-segment subspace, which loads, unloads, and drives as necessary for that segment. The subspace signals if a resource went negative, or any other local constraint was violated (such as White being arrested because the segment ends in Illinois).
   
   Effects: Changes the current segment to be the next segment, and updates the resources on the current state. Signals failure on the current state if there was a constraint violation (see local-constraint-failed).
   
   Termination: Terminates (1) when the current segment becomes the next segment, if there is no constraint violation; or (2) when a constraint violation occurs.

2. **trip-succeeds** (assignment)
   
   Conditions: Proposed without conditions, but made worst. Consequently, selected only when there are no more simulate-segment operators and no constraint violations.
   
   Effects: Adds assignment to the current schedule on the superstate, allowing the superoperator to terminate. Also updates the trip, truck, and driver sets on the superstate.
   
   Termination: By higher decision.

3. **local-constraint-failed** (failed-assignment)
   
   Conditions: Proposed if simulate-segment signals failure, and selected immediately when proposed.
   
   Effects: Signals failure in the supercontext, allowing the assign operator to terminate. Returns failed-assignment to the superstate, so we can figure out what happened.
   
   Termination: By higher decision.
• simulate-segment

State: Contains resources (weight, volume, time), initialized from the superoperator.

Search control: A segment begins with a load (if any), and ends with an unload (if any). Therefore, for correctness with respect to the truck’s weight limit and volume, load must be selected before unload.

Operators:

1. load (volume, load-time, weight)
   Conditions: Proposed if there is a shipment to load in the beginning location.
   Effects: Updates weight, volume, and time, on the current state.
   Termination: By direct application.  

2. unload (volume, unload-time, weight)
   Conditions: Proposed if there is a shipment to unload at the end location.
   Effects: Updates weight, volume, time.
   Termination: By direct application.

3. drive (time)
   Conditions: Proposed without conditions (segments always involve driving).
   Effects: Updates time.
   Termination: By direct application.

4. compute-resources
   Conditions: Proposed without conditions, but made worst. Consequently, selected when all other operations have been carried out.
   Effects: Augments the state with a record of the remaining resources, indicates in the record whether resources went negative, or whether any other local constraints were violated (such as White being arrested in Illinois).
   Termination: By direct application.

5. update-resources (resource-record)
   Conditions: Proposed when the resource record has been computed.
   Effects: Returns resource-record to the superstate. Changes the current segment to be the next segment on the superstate, allowing the superoperator to terminate.
   Termination: By higher decision.

---

5 When an operator is applied entirely by an apply-operator TC, edits in the TC terminate the operator.
3.4. Notes on Chunking

Section 3.4.1 discusses some general aspects of chunking in Soar. Section 3.4.2 examines some of the code from Chapter 4 in light of how it affects chunking. Novice Soar users are advised to skim this section for future reference, rather than read it for understanding immediately. Section 3.4.3 briefly discusses a sample expensive chunk.

3.4.1. Correctness, generality, and backtracing

There are two aspects to good chunks: correctness and generality. These aspects trade off: incorrect chunks are often that way because they are overgeneral. There is usually an identifiably optimal degree of generality, in which the chunk conditions are the weakest that guarantee that the knowledge contained in the action is correct. This applies to both forms of chunks learned in the sample implementation (operator application chunks, which modify the state, and search control chunks, which create preferences for operators).

Chunk conditions are created by a dependency analysis on subgoal results, called backtracing. Backtracing is a form of operator regression (Mitchell et al., 1986). In Soar terms, backtracing involves tracing through working memory elements that were added and deleted from working memory during processing in the subgoal, beginning with the subgoal result and working back to the working memory elements that it depends on.

Figure 5 shows a simple example of subgoal-result generation and backtracing. The initial state is created from supercontext augmentations augl and aug2. Augmentations aug3 and aug4 are copied down directly from the supercontext. Operator 1 tests only the state, and generates augmentation aug5. Operator 2 generates aug6 from aug3 and aug4. Operator 3 generates aug7 from aug5 and aug3. Finally, Operator 4 generates aug8, the subgoal result, from aug7. Backtracing follows this dependency path backwards, starting with aug8. Aug6, and before that aug4, are irrelevant to the subgoal result, and therefore are not included in the chunk conditions. Note that every operator must test the state, either to edit the state directly or to reach the objects that will be edited, and therefore backtracing will always reach the augmentations used to generate the initial state (aug1 and aug2, in the figure).
Productions that create desirability preferences (other than the require and prohibit preferences, which are special cases) are not backtraced through, even if they bring about a particular operator selection. The underlying premise is that search control affects only the efficiency and not the correctness of problem-solving (Rosenbloom et al., 1987). Consequently, encoding various forms of knowledge in terms of search control rather than conditions on operator proposal is a powerful mechanism for modulating the generality of chunks.

3.4.2. Preventing incorrect chunks

One of the main causes of incorrect chunks is the exhaustion problem. One way this can arise is if an operator is proposed with very weak conditions, and made worst so that it will be selected only when the set of other operators is exhausted. This is a convenient programming technique, as illustrated by several operators in the sample implementation (global-constraint-failed, selected when there are neither simulate-trip operators nor a schedule-succeeds operator proposed; trip-succeeds, selected when all simulate-segment operators are exhausted; and compute-resources, selected when the operations for a particular segment are exhausted). The drawback relates to subgoal results based on effects of that operator: the weak proposal conditions present little for backtrace to work with, so the conditions of the resulting chunk will be wildly overgeneral.

The sample implementation shows two responses to the exhaustion problem. The first response, which is to learn nothing, is illustrated in the TAQL constructs (TCs) in Figure 6 (they appear in the code on pages 33 and 34). The global-constraint-failed augmentation is added by task*ao*global-constraint-failed and detected by eo*task, which assigns the novalue evaluation for that case. This triggers a Soar feature (use of the quiescence t goal augmentation) that allows a production to return a result with no chunk being built. A novalue evaluation is interpreted as knowledge that this path is fruitless, with no corresponding knowledge (or desire to generate it) about how to assign credit.

```
(apply-operator task*ao*global-constraint-failed
 :space task
 :op (global-constraint-failed ^reason <reason>)
 (edit :what state
      :new (global-constraint-failed <reason>)))

(evaluate-object eo*task
 :space task
 :what lookahead-state
 ... :symbolic-value (novalue :when ((state ^global-constraint-failed)))
 ...)
```

Figure 6: Novalue: backtracking without chunking

A second response is to make up for the lack of constraint in the proposal productions by adding conditions, in either the proposal or the application TC for an operator. This is illustrated in the TC in Figure 7 (which appears in the code on page 36). Sometimes these conditions arise naturally, as when it is necessary to bind information that will be used later. For example, Condition 1 binds the trip, truck, and driver elements that are removed by the edit clause of the TC.
Figure 7: Conditions that bind constants

Frequently, however, building hazard-free chunks requires a comprehensive approach, combining an appropriate representation of the data with correct manipulations at run-time. The Soar manual (beginning on page 119) discusses data representation issues, among them techniques for achieving maximally general (but still correct) chunks. The remainder of this section presents methods of manipulating data to affect chunk generality.

Returning to Figure 7, Condition 2 uses constant binding to ensure that chunks built from replacing the head of a list do not cause problems. Constant binding exploits the fact that Soar replaces object identifiers in chunks with variables but leaves constants in place. Thus, binding a constant makes the chunk specific to the name of the element pushed on the list. If this were not done, the chunk built from severing the pointer to the first element of the list would fire whenever that pointer were set.

Condition 2 also has a less obvious effect. Consider the two simulate-trip operators in the trace excerpt in Figure 8 (the excerpt begins on page 44). As the second simulation (o380) is being carried out, a constraint violation occurs. The default productions convert the violation into a failure evaluation (for the first simulate-trip operator, since it is the one being evaluated). The failure evaluation results in a search-control chunk (p946) that in the future will reject operators similar to c374 under similar circumstances.

Figure 9 shows p946 (with some of the more ghastly conditions excised; the full chunk appears in the chunk listing on page 52). The effect of Condition 2 is to cause the brown/traveler/trip1 assignment to be rejected specifically when the previous assignment involved trip3. There is no
34 0: o344 ((brown) (traveler) (trip1) simulate-trip) evaluate-object
38 ... 0: c374 ((brown) (traveler) (trip1) simulate-trip)
68 ... 0: o419 (trip-succeeds)
69 ... 0: o380 ((gray) (piper) (trip2) simulate-trip)
112 ... 0: o938 (weight-limit-exceeded local-constraint-failed)

Evaluation of operator c374 (simulate-trip) is failure

Figure 8: Lookahead leading to search-control chunk p946

clear purpose for this condition, in terms of knowledge usefully captured by the chunk. In fact, this condition makes the chunk overspecific, since the relevant information is only that trip2 cannot be completed with a small truck (such as piper).

Questions of usefulness aside, the trip3 test illustrates how chunking can be affected by

(sp p946
(goal <g2> ^desired <d4> ^problem-space <pl> ^state <d2>
 ^operator <ol> )
(problem-space <pl> [...] ^name task)
(state <d2> ^dummy-att true ^driver <d3> ^truck <t4> ^trip <t3>
 ^current-schedule <ll>)
(driver <d3> ^drive-time 12.5 ^name gray)
(truck <t4> ^volume 400 ^weight-limit 5000 ^type small ^name piper)
(trip <t3> ^name trip2 ^first-segment <s1>)
(segment <s1> ^name segment1 ^trip trip2 ^source gray ^next-segment <s2>)
(segment <s2> ^name segment2 ^trip trip2 ^next-segment <s3>)
(segment <s3> ^name segment3 ^trip trip2 ^next-segment <s7>)
(segment <s7> ^load-shipment carpet ^unload-shipment NIL ^name segment4
 ^trip trip2 ^next-segment <s8>)
(segment <s8> ^unload-shipment carpet ^unload-shipment NIL
 ^name segment5 ^trip trip2 ^next-segment <s9>)
(segment <s9> ^load-shipment newsprint ^load-shipment NIL ^name segment6
 ^trip trip2)
(list <ll> ^car <ol> )
(operator <ol> ^trip <t2> )
(trip <t2> ^name trip3)
(operator <ol> [...] ^name simulate-trip
 ^trip <t5> ^truck <t1> ^driver <dl> )
(trip <t5> ^name trip1)
(truck <t1> ^volume 640 ^weight-limit 10000)
(driver <dl> ^drive-time 11)
(goal <g1> ^object NIL ^state <s6> )
(state <s6> ^shipment <s5> <s4> ^license <12> ^city <c2> )
(shipment <s5> ^name newsprint ^weight 6000.0 ^volume 400.0
 ^load-time 0.2)
(shipment <s4> ^name carpet ^weight 500.0 ^volume 100.0 ^unload-time 0.25
 ^load-time 0.2)
(license <12> ^truck-type small ^holder gray)
(city <c2> ^name gray ^truck piper ^driver gray)

Figure 9: The search-control chunk p946
manipulating the data. Figure 10 shows how backtracing comes to include trip3 in the chunk conditions. The simulate-trip*ao*trip-succeeds TC (of Figure 7) replaces the current-schedule augmentation while testing trip3. In Figure 10, the new current-schedule augmentation is lxxx (l because we know it is a list object, and xxx because unlike with operators we can not determine its identifier from the trace). This creates a dependency of lxxx on trip3. When lxxx is tested during the application of o938 (the simulate-trip*ao*local-constraint-failed TC is on page 37), the dependency is extended to the subgoal result. When the chunk is built, o938 acts as a bridge from the result to lxxx, and o419 acts as a bridge from lxxx to trip3. Trip3 was copied directly from the higher task context, so backtracing includes it in the chunk conditions. The general technique for ensuring that a particular condition (based on a supercontext augmentation) is included in a chunk is to ensure that each operator in the subgoal extends the dependency, by creating an augmentation while testing a previous augmentation that depends on the one from the supercontext.

Note that the augmentation created by one operator to be tested by the next need not be meaningful beyond the function of extending the dependency. In a variant known locally as beadining, the augmentations are simply new identifiers, each replaced by the next. Backtracing then strings up the identifiers like beads, backtracing through whatever augmentations were tested in creating them.\(^6\)

![Diagram](image)

**Figure 10:** Backtracing to the trip3 condition

A second instance of constant binding, shown in Figure 11, does provide a useful constraint on p945. During the application of o380 (between decision cycles 69 and 112 in Figure 8), Condition 3 causes trip2 to be included in the backtrace and eventually in the chunk conditions (Figure 9). This test is semantically relevant to the constraint that small trucks are not able to complete this trip.

For completeness, we also account for the appearance of trip1 (the third and last trip name) in the conditions of p946. The trip1 test occurs because of the simulate-trip*ao*local-constraint-failed TC (page 37), which applies o938. The relevant condition (corresponding to Condition 2)

---

\(^6\)The bead metapnor is due to Rick Lewis.
(apply-operator simulate-segment*ao*compute-resources
   :space simulate-segment
   :op compute-resources
   ...
   ;; add the updated resources to the object:
   (edit :what (:none resources <resources>)
   ...
   ;; add the trip, to make sure that the segment and trip stay
   ;; tied together in the chunks that update the resources.
   ;; if we don’t do this, the chunks could transfer
   ;; incorrectly to the same segment of another trip.
   :new (trip <trip-name>
      :when ((segment <segment> ^trip <trip-name>))
      ...
   )

Figure 11: Conditions that prevent overgeneral search-control chunks

tests the name of the last trip pushed onto the current schedule (trip1, by 0419). Note that 0938 accesses trip1 through lxxx, extending the dependency of the subgoal result on trip3 as described at the top of the previous page.

3.4.3. Expensive chunks

The problem of expensive chunks (Tambe et al., 1990) is a manifestation in Soar of the utility problem in explanation-based learning (Minton, 1985). It can arise whenever a representation uses sets (multi-attributes), such as that of the top state of the sample implementation.

The chunk in Figure 12 implements the task operator (the chunk also appears on page 56). It reads the top-state and creates the solution (the actions are elided). It is expensive to match because of the combinatorics in the <s4> condition (second from the top). The ^license test can match 3^3 ways, the ^city test 2^2 ways, the ^driver test 3^3 ways, etc. The match cost is proportional to the product of these terms.

For this task, a set-based representation is a great convenience. However, the general solution to expensive chunks is to use structured representations, such as lists.
Figure 12: An expensive chunk
4. Sample Implementation in Soar

4.1. Introduction

This chapter presents a sample implementation for the shipment scheduling assistant. This implementation serves as a reference for the design given in Chapter 3.

Section 4.2 is a program listing, organized by problem space. Section 4.3 is an execution trace, followed by statistics on the run and the source code. Section 4.4 is a listing of the chunks generated during the run, useful for cross-referencing with the trace, and as a reference for the discussion of learning in the previous chapter. For information on obtaining on-line versions of these files, send mail to soar-requests@cs.cmu.edu.

4.2. Program Listing

```taql
- Mode: TAQL -

File: /afs/cs.cmu.edu/usr/altmann/taql/truck/truck.taql
Author: Erik Altmann
Created On: Sat Sep 28 20:56:38 1991
Last Modified By: Erik Altmann
Update Count: 113
TAQL Version: 3.1.4

PURPOSE
Soar implementation of the Shipment Scheduling Assistant,
a.k.a. the Trucking Task. Runs with chunking on. See trace.txt
for a trace, and chunks.soar for the corresponding chunks.

TABLE OF CONTENTS
In terms of spaces: top-space, task, simulate-trip, and
simulate-segment. To reach the code for SPACE, search for
"ps*SPACE", which will locate the propose-space TAQL construct
for that space.

(C) Copyright 1991, Carnegie Mellon University, all rights reserved.

object/attribute pairs for Soar trace. the value of each attribute
will be printed.

(trace-attributes '(% attributes of the simulate-trip operator (in the task space):
(operator driver)
(operator truck)
(operator trip)

% attributes of operators in the simulate-trip space:
(operator segment) ; simulate-segment
(operator reason) ; trip-succeeds

% attributes of operators in the simulate-segment space:
(operator shipment-name)
(operator volume)
(operator load-time)
(operator unload-time)
(operator weight)
(operator time) }
```
the top context contains the task state and the task operator. the task-state contains all the static information for the task. the task operator is bare.

there is no ps*top-space to.

(proposal-task-state pts*task

:new (weather rain)

:new (driver
    ((driver "name green" "union yes" "drive-time 11")
    (driver "name white" "union no" "drive-time 12.5")
    (driver "name brown" "union yes" "drive-time 11")
    (driver "name gray" "union no" "drive-time 12.5"))

the hierarchy of licenses is implicit in the "holders:

everyone holds type 1, fewer hold type 2, fewest hold type 3:

:new (license
    ((license "name class3" "truck-type big" "holder green + &, brown + &")
    (license "name class2" "truck-type medium" "holder green + &, brown + &, white + &")
    (license "name class1" "truck-type small" "holder green + &, brown + &, white + &, gray + &"))

:new (truck
    ((truck "name cannonball" "type big" "weight-limit 32000" "volume 1280")
    (truck "name piper" "type small" "weight-limit 5000" "volume 400")
    (truck "name traveler" "type medium" "weight-limit 10000" "volume 640")
    (truck "name queen-bee" "type medium" "weight-limit 10000" "volume 640"))

:new (city
    ((city "name gary" "state indiana" "driver brown + &, gray + &" "truck piper + &, traveler + &")
    (city "name indy" "state indiana" "driver green + &, white + &" "truck cannonball + &, queen-bee + &")
    (city "name utica" "driver nil" "truck nil" "state illinois")
    (city "name viola" "driver nil" "truck nil" "state illinois")
    (city "name warsaw" "driver nil" "truck nil" "state indiana")
    (city "name kokomo" "driver nil" "truck nil" "state indiana")
    (city "name attica" "driver nil" "truck nil" "state indiana")
    (city "name cook" "driver nil" "truck nil" "state indiana")

:new (shipment
    (shipment "name typewriters" "weight 1000.0" "volume 200.0" "load-time .2" "unload-time .25")
    (shipment "name carpet" "weight 500.0" "volume 100.0" "load-time .2" "unload-time .25")
    (shipment "name newsprint" "weight 6000.0" "volume 400.0" "load-time .2" "unload-time .25"))
;; trip 1:
:new (trip
 (trip ^name tripl ^first-segment <s1>)
 (segment <s1> ^name segment1 ^road u30 ^source gary ^dest warsaw
  ^load-shipment typewriters
  ^unload-shipment nil
  ^trip tripl
  ^next-segment <s2>)
 (segment <s2> ^name segment2 ^road u31a ^source warsaw ^dest kokomo
  ^load-shipment nil
  ^unload-shipment nil
  ^trip tripl
  ^next-segment <s3>)
 (segment <s3> ^name segment3 ^road u31b ^source kokomo ^dest indy
  ^load-shipment nil
  ^unload-shipment nil
  ^trip tripl
  ^next-segment <s4>)
 (segment <s4> ^name segment4 ^road 174 ^source indianapolis ^dest attica
  ^load-shipment nil
  ^unload-shipment nil
  ^trip tripl))

;; trip 2:
:new (trip
 (trip ^name tripl ^first-segment <s1>)
 (segment <s1> ^name segment1 ^road u41a ^source gary ^dest cook
  ^load-shipment nil
  ^unload-shipment nil
  ^trip tripl
  ^next-segment <s2>)
 (segment <s2> ^name segment2 ^road 180b ^source cook ^dest utica
  ^load-shipment nil
  ^unload-shipment nil
  ^trip tripl
  ^next-segment <s3>)
 (segment <s3> ^name segment3 ^road 180a ^source utica ^dest viola
  ^load-shipment nil
  ^unload-shipment nil
  ^trip tripl
  ^next-segment <s4>)
 (segment <s4> ^name segment4 ^road 180a ^source viola ^dest utica
  ^load-shipment carpet
  ^unload-shipment nil
  ^trip tripl
  ^next-segment <s5>)
 (segment <s5> ^name segment5 ^road 180b ^source utica ^dest cook
  ^load-shipment nil
  ^unload-shipment carpet
  ^trip tripl
  ^next-segment <s6>)

;; note the several nil fields. trip2 ends oddly, with doing a load:
 (segment <s6> ^name segment6 ^road nil ^source cook ^dest nil
  ^load-shipment newsprint
  ^unload-shipment nil
  ^trip tripl)))

;; trip 3:
:new (trip
 (trip ^name trip3 ^first-segment <s1>)
 (segment <s1> ^name segment1 ^road 170b ^source indy ^dest yale
  ^load-shipment nil
  ^unload-shipment nil
  ^trip trip3
  ^next-segment <s2>)
 (segment <s2> ^name segment2 ^road u41e ^source yale ^dest attica
  ^load-shipment nil
  ^unload-shipment nil
  ^trip trip3)))
(augment a*top-space*speed-when-rainy-on-primary
  :space top-space
  :when ((top-state ~weather rain ~road <r>))
  :what (:none road <r>)
  :new (speed 55)
  :when ((road <r> ~grade primary)))

(augment a*top-space*speed-when-rainy-on-secondary
  :space top-space
  :when ((top-state ~weather rain ~road <r>))
  :what (:none road <r>)
  :new (speed 50)
  :when ((road <r> ~grade secondary)))

(augment a*top-space*speed-when-rainy-on-tertiary
  :space top-space
  :when ((top-state ~weather rain ~road <r>))
  :what (:none road <r>)
  :new (speed 35)
  :when ((road <r> ~grade tertiary)))

; once the speeds are computed, the time to traverse each road can be
; computed, given the length. the times also augment the road
; object.

(augment a*top-space*time-from-length-and-speed
  :space top-space
  :when ((state ~road <r>)
         (road <r> ~length <l> ~speed <s>))
  :what (:none road <r>)
  :new (time (compute <l> / <s>)))

; propose doing the scheduling task:

(propose-task-operator pto*task
  :op do-task)
; apply the schedule operator from the top-space, by trying various assignments of trucks and drivers to trips.

(provide-space ps*task
  :function (apply operator do-task)
  :space task)

(provide-initial-state pis*task
  :space task)

; create a set each of the available drivers and trucks and the trips that have to be covered. we'll update this set as assignments are made (simulate-trip*ao*trip-succeeds).

:copy (driver truck trip)

; initialize the current-schedule. use a dummy simulate-trip operator, so that the various conditions that test the name of the last assignment on the current-schedule for chunking purposes will work without a special case for the empty list.

; note: below, "list" is a data macro. this clause creates a list whose car is the dummy operator. the syntax used is the third form of ACTION-SPEC (see the TAQL manual, p. 52).

:new (("current-schedule (list <dummy>)")
  (operator <dummy> ^name simulate-trip ^trip <dummy-trip>)
  (trip <dummy-trip> ^name dummy)))
generate an assignment (a <driver, truck, trip> tuple), and propose
simulating the trip.

the conditions of the operator proposal generate the assignments
that embody the immediate constraints, like the truck and the
driver being in the first city of the trip. these constraints
are easy to test, given the structure of the information on
the top state.

(propose-operator task*po*simulate-trip
  :space task
  :when (bind a driver, truck, and city:
    (state "<driver> <driver> <truck> <truck> <trip> <trip>)
  ;; constrain the driver and truck to be in the same city:
    (top-state "city <city>)
    (city <city> "name <city-name> "truck <truck-name> "driver <driver-name>)
    (truck <truck> "name <truck-name>)
    (trip <trip> "name <trip-name> "first-segment <segment>)
    (segment <segment> "source <city-name>)
  ;; constrain the driver to be licensed to drive the truck:
    (top-state "license <license>)
    (driver <driver> "name <driver-name>)
    (truck <truck> "type <truck-type>)
    (license <license> "truck-type <truck-type> "holder <driver-name>))
  :op (simulate-trip "driver <driver> "truck <truck> "trip <trip>))

simulate-trip is applied in a subgoal, by the simulate-trip space.
it succeeds if no local constraints are violated, such as resources
going negative. the termination conditions are in the next two TCs.

note: the simulate-trip operator itself is as an easy way to
represent an assignment, so we build up the schedule by adding
simulate-trip operators to a list, called "current-schedule.

simulate-trip succeeds when the simulate-trip operator itself has
been added to the current schedule (by the subgoal). terminate the
operator when this happens:

(apply-operator task*ao*simulate-trip*trip-succeeded
  :space task
  :op simulate-trip
  :terminate-when ((state "current-schedule <schedule>))
    bind "cons cell"
    (operator (car <schedule>) "trip <trip>)
    bind its "car"
    (operator "trip <trip>))

terminate simulate-trip if the subgoal signals a local constraint
violation. local-constraint-failed is detected as a failure
condition by eo*task (below).

note: multiple :terminate-when clauses in one TC are conjunctive,
so we need two apply-operator TCs to represent the two conditions.

(apply-operator task*ao*simulate-trip*constraint-violation
  :space task
  :op simulate-trip
  :terminate-when ((state "local-constraint-failed)))
when there are no trips left, we're done:

(propose-operator task*po*schedule-succeeds
  :space task
  :when ((state ^current-schedule <last-assignment> - ^trip))
  :op (schedule-succeeds ^complete-schedule <last-assignment> ^reason no-trips-left))

; bind the name of the previous trip. this causes the search
; control chunks that arise from detecting the success of this trip
; to include the previous trip name in their conditions:

:when ((operator (car <last-assignment>) ^trip <last-trip>))

; we know that the current schedule is complete, so put that
; information on the state. this signals success (via eo*task, below).

(apply-operator task*ao*schedule-succeeds
  :space task
  :op (schedule-succeeds ^complete-schedule <schedule>)
  (edit :what state
        :new (complete-schedule <schedule>)))

; be prepared for a global constraint violation. propose an operator
; that represents such a violation occurring, then make it worst. if
; at some point there are trips left but no assignments for them,
; then neither simulate-trip nor schedule-succeeds will be proposed,
; so global-constraint-failed will be selected by default.

; note: this illustrates a general programming technique, which is
; to use the selection of an operator with a worst preferences to
; implicitly represent the knowledge that each of a set of options
; has been exhausted. however, because such a selection is not
; based on any explicit knowledge that can be backtraced through,
; chunks learned from results of such operators can be vastly
; overgeneral. one solution is to not learn from such results (see
; eo*task).

(propose-operator task*po*global-constraint-failed
  :space task
  :when ((state ^trip))
  :op (global-constraint-failed ^reason trips-level-over))

(prefer task*p*global-constraint-failed*worst
  :space task
  :op global-constraint-failed
  :value worst)

(apply-operator task*ao*global-constraint-failed
  :space task
  :op (global-constraint-failed ^reason <reason>)
  (edit :what state
         :new (global-constraint-failed <reason>)))
:: signal task success and local and global failures. this TC causes
:: a lookahead instance of the task space to exit, causing another
:: simulate-trip alternative to be tried out.

(evaluate-object eo*task
  :space task
  :what lookahead-state

  :: succeed when there’s a complete schedule (generated by
  :: task*ao*schedule-succeeds). Soar’s default productions
  :: propagate success to the top of the lookahead-search stack,
  :: causing Soar to learn search-control chunks that remember the
  :: sequence of simulate-trip operators.

  :symbolic-value (success
                  :when ((state ^complete-schedule)))

  :: backtrack if there’s a global-constraint-failed. the novalue
  :: evaluation tells Soar not to learn anything. we don’t know
  :: which assignment caused the problem or why, so there’s nothing
  :: useful to learn (see note at task*po*global-constraint-failed).

  :symbolic-value (novalue
                   :when ((state ^global-constraint-failed)))

  :: backtrack (and learn that the last assignment was a bad choice)
  :: if there’s a local-constraint-failed (created by
  :: task*ao*simulate-trip*constraint-violation).

  :symbolic-value (failure
                   :when ((state ^local-constraint-failed)))

  :: goal-test knowledge, to detect success and failure when not in
  :: lookahead search. replicates the knowledge in eo*task.

(goal-test-group gtg*task*success
  :space task
  :group-type success
  :when ((superspace ^name top-space)
         (state ^complete-schedule)))

(goal-test-group gtg*task*failure
  :space task
  :group-type failure
  :when ((superspace ^name top-space))

  :: disjunctive failure test.
  :test (failure
         :when ((state ^global-constraint-failed)))
  :test (failure
         :when ((state ^local-constraint-failed)))

  :: return the complete schedule to the top state.

  :: note: with this TC, Soar builds a chunk that applies the task
  :: operator (proposed by pto*task) after learning. without this TC,
  :: no chunk will be learned for the task operator (TACL’s runtime
  :: support automatically terminates the task operator when a
  :: successful or failed state is reached in the space that applies
  :: it); after learning, the task operator will impasse, and chunks
  :: learned during lookahead search will guide Soar directly to a
  :: solution in the subspace without further impasses.

(result-superstate rs*task*success
  :space task
  :group-type success
  (edit :what superstate
         :when ((state ^complete-schedule <schedule>))
         :new (complete-schedule <schedule>)))
::; --------------------------
::; the simulate-trip space
::; --------------------------

::; applies the simulate-trip operator from the schedule space.
::; if the simulation succeeds, trip-succeeds pushes the superoperator
::; onto the current-schedule list on the superstate. if it fails (due
::; to a local constraint failure), local-constraint-failed pushes this
::; assignment onto the current schedule, and returns that as the value
::; of *local-constraint-failed.

(propose-space ps*simulate-trip
  :function (apply operator simulate-trip)
  :space simulate-trip)

::; copy the trip/truck/driver set down from the superstate, for
::; convenience. establish the volume and weight-limit of the truck.
::; and the maximum time the driver can drive, as initial values of
::; those resources.

(propose-initial-state pis*simulate-trip
  :space simulate-trip
  :when ((superoperator ^truck <truck> ^trip <trip> ^driver <driver>)
    (driver <driver> ^drive-time <drive-time>)
    (truck <truck> ^volume <volume> ^weight-limit <weight-limit>))
    :new (driver <driver>)
    :new (truck <truck>)
    :new (trip <trip>)
    :new (initial-resources ((resources ^current-time <drive-time>
      ^current-volume <volume>
      ^current-weight <weight-limit>))))

::; simulate the driver driving the truck over each segment of the trip
::; in turn. after each segment, the simulate-segment operator updates
::; the resources.

(propose-operator simulate-trip*po*simulate-segment
  :select-once-only ; so first segment is only done once
  ::; propose doing the first segment, using the initial resources:
  :op (simulate-segment ^segment <segment> ^resources (list <res>))
    :when ((state ^trip <trip> ^initial-resources <res>)
      (trip <trip> ^first-segment <first>)))

::; at each segment, propose doing the next one, using the current
::; resources:
  :op (simulate-segment ^segment <next> ^resources <res>)
    :when ((operator ^segment <current>)
      (segment <current> ^next-segment <next>)
      (state ^current-resources <res>))))

::; operator simulate-segment is applied in subgoal, in the
::; simulate-segment space. the space returns a resource object,
::; pushing it onto the list *current-resources (creating the list if
::; it has to). the resource object specifies the segment it was
::; constructed in, allowing us to test we can terminate the operator.
::; the resource object also specifies "failed true if any resource
::; (volume, weight-limit, time) ran out during the segment, or if
::; other constraints were violated (see
::; simulate-segment*so*compute-resources).

(apply-operator simulate-trip*so*simulate-segment
  :space simulate-trip
  :op (simulate-segment ^segment <segment>))
  :terminate-when ((state ^current-resources <res>)
    (resources (car <res>) ^segment <segment-name>)
    (segment <segment> ^name <segment-name>)))
;; be prepared to declare the simulation successful. propose
;; trip-succeeds and make it worst, so that it's selected only
;; when there are no segments left to drive over.
;;
;; in the proposal, create a new list whose car is the current
;; assignment, and whose cdr is the existing schedule from the
;; super-space. make this new list an argument to the operator,
;; which can return it directly to the superstate (becoming the
;; updated schedule).

(propose-operator simulate-trip*po*trip-succeeds
 :space simulate-trip

 ;; bind the superoperator, and the current schedule:
 :when ((superstate ^current-schedule <last-assignment>)
      (supergoal ^operator <so>))

 ;; put the updated schedule on the operator, so that the operator
 ;; application TC can return the updated schedule to the
 ;; super-state. (updating and returning requires two steps, so
 ;; doing the update here allows the apply-operator TC to return
 ;; the result.)
 :op (trip-succeeds ^assignment (cons <so> <last-assignment>))

(prefer simulate-trip*p*trip-succeeds*worst
 :space simulate-trip
 :op trip-succeeds
 :value worst)

 ;; if the assignment succeeds, return the new schedule to the
 ;; super-context, and update the superstate's trip/truck/driver sets.
 (apply-operator simulate-trip*a*trip-succeeds
 :space simulate-trip
 :op (trip-succeeds ^assignment <new>)

 ;; bind the information we need to update the trip/truck/driver sets:
 :when ((operator (car <new>)
            ^trip <trip> ^driver <driver> ^truck <truck>))

 ;; the chunks that update the schedule in the super-context will
 ;; loop unless we're careful, because one chunk rejects the
 ;; current head of the list, and a second adds the new head.
 ;; we can prevent looping by making the chunks specific to the
 ;; current head of the list. <current-trip-name> matches a
 ;; constant, and that constant will appear in the chunk:
 :when ((superstate ^current-schedule <current>)
        (operator (car <current>) ^trip <current-trip>)
        (trip <current-trip> ^name <current-trip-name>))

(edit :what superstate

 ;; push the new trip onto the head of the schedule list:
 :replace (current-schedule :by <new>)

 ;; trip is taken care of:
 :remove (trip <trip>)

 ;; driver and truck are used up:
 :remove (driver <driver>)
 :remove (truck <truck>)))
the simulation fails if a resource went negative, the value of 'reason is printed in the trace (see trace-attributes).

```
(propose-operator simulate-trip*po*local-constraint-failed
 :space simulate-trip
 ;; as in simulate-task*po*trip-succeeds:
 :when ((supergoal "operator <so>"
          (superstate "current-schedule <last-assignment>")
          (local-constraint-failed "failed-assignment" (cons <so> <last-assignment>
          ^reason <reason>))

 ;; failure condition:
 :when ((state "current-resources <res>
          (resources (car <res>) "failed true ^reason <reason>)

 ;; if local-constraint-failed is proposed, select it immediately.
 (prefer simulate-trip*p*local-constraint-failed*require
 :space simulate-trip
 :op local-constraint-failed
 :value require)

 ;; if the simulation fails, return local-constraint-failed to the supercontext. don't bother updating the trip/truck/driver sets, because the supercontext will signal a failure condition and exit

 (apply-operator simulate-trip*ao*local-constraint-failed
 :space simulate-trip
 :op (local-constraint-failed "failed-assignment <schedule>")

 ;; to prevent chunks that will loop in other cases (as in simulate-trip*ao*trip-succeeds):
 :when ((superstate "current-schedule <current>
          (operator (car <current>) "trip <current-trip>
          " (trip <current-trip> "name <current-trip-name>")

 ;; failure termination condition on the superoperator:
 (edit :what superstate
       :new (local-constraint-failed <schedule>))
```
the simulate-segment space

applies the simulate-segment operator from the simulate-trip space.

the space succeeds if the segment can be completed with the volume
and weight currently available in the truck, and if the time left
the driver is enough to cover loading, driving, and unloading. it
fails otherwise. the return value is a resources object, pushed
onto the *current-resources list of the superstate. the list is
created if it isn't there. if the segment fails, the resources
object is flagged with "failed true and a "reason.

the compute-resources operator represents the oddball constraint
that white can't drive through illinois.

function (apply operator simulate-segment)

the arguments to the superoperator are the segment to do and the
resources to do it with. unpack the resources so we can modify
them locally. unpack the segment for convenience.

(propose-space ps simulate-segment

:space simulate-segment

:when ((superoperator *segment <seg> *resources <res>)
   (resources (car <res>)
     *current-volume <volume> *current-time <time> *current-weight <weight>))
   :new (segment <seg>)
   :new (current-volume <volume>)
   :new (current-time <time>)
   :new (current-weight <weight>))

:when the *shipment-name of a segment is non-nil, then propose loading
that shipment:

(propose-operator simulate-segment*po*load

:space simulate-segment

:select-one-only ; load only once per segment
:op (load *shipment-name <shipment-name> *volume <volume> *load-time <load-time>
  *weight <weight>)
:when ((top-state *shipment <sh>)
   (shipment <sh> *name <shipment-name> *weight <weight>
    *volume <volume> *load-time <load-time>))
:when ((state *segment <seg>)
   (segment <seg> - *load-shipment nil *load-shipment <shipment-name>))))

load the shipment by adjusting the resources:

(apply-operator simulate-segment*ao*load

:space simulate-segment

:op (load *weight <weight> *volume <volume> *load-time <load-time>)
:when ((state *current-weight <current-weight> *current-volume <current-volume>
  *current-time <current-time>))

(edit :what-state
  :replace (current-weight :by (compute <current-weight> - <weight>))
  :replace (current-volume :by (compute <current-volume> - <volume>))
  :replace (current-time :by (compute <current-time> - <load-time>))))
; similarly for unloading. for correctness, loading has to be done before
; unloading (see the explanation above simulate-segment*load*unload*better).

(provide-operator simulate-segment*po*unload
:space simulate-segment
:select-once-only ; unload only once per segment
:op (unload *shipment-name <shipment-name> *volume <volume>
  *unload-time <unload-time> *weight <weight>)
:when ((top-state *shipment <sb>)
  (segment <seg> *name <segment-name> *weight <weight>
  *volume <volume> *unload-time <unload-time>))
:when ((state *segment <seg>)
  (segment <seg> - *unload-shipment nil *unload-shipment <shipment-name>)))

(apply-operator simulate-segment*ao*unload
:space simulate-segment
:op (unload *weight <weight> *volume <volume> *unload-time <unload-time>)
:when ((state *current-weight <current-weight> *current-volume <current-volume>
  *current-time <current-time>))
(edit :what state
  :replace (current-weight :by (compute <current-weight> + <weight>))
  :replace (current-volume :by (compute <current-volume> + <volume>))
  :replace (current-time :by (compute <current-time> - <unload-time>)))))

;; adjust the time resource by the length of time it takes to drive
;; the segment.

(provide-operator simulate-segment*po*drive
:space simulate-segment
:select-once-only ; drive only once per segment
:op (drive *time <time>)
:when ((top-state *road <road>)
  (segment <seg> *road <road-name>)
  (segment <seg> - ^dest nil)
  (road <road> ^name <road-name> ^time <time>)) ;; entailed

(apply-operator simulate-segment*ao*drive
:space simulate-segment
:op (drive *time <time>)
:when ((state *current-time <current-time>))
(edit :what state
  :replace (current-time :by
    (compute <current-time> - <time>))))

;; be prepared to update resources when no more resource-consuming
;; actions remain.

(provide-operator simulate-segment*po*compute-resources
:space simulate-segment
:op compute-resources)

(prefer simulate-segment*po*compute-resources
:space simulate-segment
:op compute-resources
:value worst)

;; update resources by building a new resource object on the current state.

(apply-operator simulate-segment*ao*compute-resources
:space simulate-segment
:op compute-resources

;; bind information required to build the resource object:
:when ((state *current-weight <weight>
  *current-volume <volume>
  *current-time <time>
  *segment <segment>))
create a new resource object and attach it to the current state:
(edit :what state
   :new (resources <resources>))

; add the updated resources to the object:
(edit :what (:none resources <resources>)
   ; add the segment, so that when the resource object is returned
   ; to the superstate, the superoperator will be able to tell
   ; when to terminate (its segment parameter will be the same as
   ; the segment of the head of the current-resources list):
   :new (segment <segment-name>
      :when ((segment <segment> ^name <segment-name>)))
   ; add the trip, to make sure that the segment and trip stay
   ; tied together in the chunks that update the resources.
   ; if we don't do this, the chunks could transfer
   ; incorrectly to the same segment of another trip.
   :new (trip <trip-name>
      :when ((segment <segment> ^trip <trip-name>)))
   ; add the resources:
   :new (current-weight <weight>)
   :new (current-volume <volume>)
   :new (current-time <time>))

; failure conditions arise when a resource has fallen below zero:
(edit :what (:none resources <resources>)
   :new (failed true)
   :new (reason weight-limit-exceeded)
   :when ((state ^current-weight < 0.0)))
(edit :what (:none resources <resources>)
   :new (failed true)
   :new (reason volume-exceeded)
   :when ((state ^current-volume < 0.0)))
(edit :what (:none resources <resources>)
   :new (failed true)
   :new (reason time-exceeded)
   :when ((state ^current-time < 0.0)))

; this is the failure condition in which white is wanted for a
; crime in illinois, the chunks built are more specific than
; they need to be, because they include all sorts of other
; resources that have nothing to do with failure condition.
(edit :what (:none resources <resources>)
   :new (failed true)
   :new (reason whites-a-criminal-in-illinois)
   :when ((top-state ^city <city>)
      (city <city> ^name <city-name> ^state illinois)
      (state ^segment <segment>)
      (superstate ^driver <driver>)
      (driver <driver> ^name white))))
}

; end of simulate-segment*ao*compute-resources
when the resources object is built, propose returning it:

```
(apply-operator simulate-segment update-resources
 :space simulate-segment
 :op (update-resources "resources <res>")
 :when ((state "resources <res>")))
```

push the resource object onto the list in current resources. If
the list has not been created yet, create it.

```
(space simulate-segment
 :op (update-resources "resources <res>")
 :when ((state segment <segment>)
 (segment <segment> "name <segment-name>"))
```

bind the name of the current segment (see below):

```
:when ((state segment <segment>)
 (segment <segment> "name <segment-name>"))
```

if the current-resources list exists already:

```
:replace (current-resources
 :by (cons <res> <res-list>)
 :when ((superstate "current-resources <res-list>")
 :when ((state "current-resources <res-list>")
 :prevent looping chunks:
 (resources (car <res-list>) - "segment <segment-name>")))
```

if there’s no current-resources list:

```
:new (current-resources (list <res>)
 :when ((superstate - "current-resources")))
```

for a given segment, "load-shipment" non-nil means load at the source
and "unload-shipment" non-nil means unload at the destination.
So for correctness wrt weight-limit and volume, loads have to come first.

```
(compare c*load*unload*better
 :space simulate-segment
 :op1 load
 :op2 unload
 :relation better)
```

Other than that, it doesn't matter what order resources are
consumed in, but it doesn't hurt to do it in a sensible one.

```
(compare c*load*drive*better
 :space simulate-segment
 :op1 load
 :op2 drive
 :relation better)
```

```
(compare c*drive*unload*better
 :space simulate-segment
 :op1 drive
 :op2 unload
 :relation better)
```
;;; this production replaces a soar default production that monitors
;;; evaluations of lookahead operators. It prints the id of the
;;; operator copy, rather than the source of that copy. This is an
;;; improvement, because the copy shows up in the trace, so the id is
;;; useful for reference.

(sp default*monitor*operator*evaluation
   (goal <top> ^object nil ^verbose false)
   (goal <g> ^object <ag> ^state <a>)
   (state <a> ^tried-tied-operator <obj>)
   (goal <ag> ^operator <so>)
   (operator <so> ^type evaluation ^evaluation <e>)
   (evaluation <e> ^ <numeric-value symbolic-value >> <n>)
   (<class> <obj> ^name <name>)
   -->
   (tabstop <tab>)
   (write2 (crlf) (tabto <tab>) " Evaluation of " <class> " " <obj> " "
   " <name> " is " <n>)

;;; end of code
4.3. Execution Trace

PURPOSE
Trace of a sample implementation of the Trucking Task, aka the Shipment Scheduling assistant. The source code is in truck.taql, and chunks produced from this trace are in chunks.soar.

TABLE OF CONTENTS
The trace, followed by print-stats and taql-stats.


Starting Soar5 ...
Allegro CL 3.1.12.2 [DECstation] (11/19/90)
Copyright (C) 1985-1990, Franz Inc., Berkeley, CA, USA

Soar (Version 5, Release 2)
Created August 26th, 1991
Bugs and questions should be sent to Soar-bugs@cs.cmu.edu
The current bug-list may be obtained by sending mail to Soar-bugs@cs.cmu.edu with the Subject: line "bug list"
This software is in the public domain,

This software is made available AS IS, and Carnegie Mellon University and the University of Michigan make no warranty about the software or its performance.

See (soarnews) for news.
; Loading /usr0/althmann/.soar-init.lisp.
; Loading /usr/misc/.Soar5/bin/Soar5.latest.patches.lisp.
<cl>
<cl> (load "/usr/misc/.Soar5/lib/taql/load.lisp")
; Loading /usr/misc/.Soar5/lib/taql/load.lisp.

Disabling selected default productions: #***#***$#$#$#$#$#$#$#$#$
Loading TAQL support productions:
******************************************************************************

TAQL 3.1.4
Created July 15, 1991

Bug reports should be sent to Soar-bugs@cs.cmu.edu.
Send comments on TAQL to Gregg.Yost@cs.cmu.edu or Erik.Altmann@cs.cmu.edu.

t
<cl> (load "/afs/cs.cmu.edu/user/althmann/taql/truck/truck.taql")
; Loading /afs/cs.cmu.edu/user/althmann/taql/truck/truck.taql.
******************************************************************************
***$*(excised DEFAULT-MONITOR-OPERATOR-EVALUATION)

t
<cl> (user-select) ;; for documentation
first
<cl> (learn on) ;; turn learning on
Learn status: on all-goals print trace
on all-goals
print trace
<cl> (run)

Learn status: on all-goals print trace

0  G: gl
1  F: p4 (top-space)
2  S: s73
3  O: o69 (do-task)
4  =>G: g76 (operator no-change)
5  F: p83 (task)
6  S: s104
7  =>G: g130 (operator tie)
8  F: p131 (selection)
9  S: s138
10  O: o143 (((green) (cannonball) (trip3) simulate-trip) evaluate-object)
11  =>G: g171 (operator no-change)
12  F: p83 (task)
13  S: d181
14  O: o182 (((green) (cannonball) (trip3) simulate-trip)
15  =>G: g212 (operator no-change)
16  F: p219 (simulate-trip)
17  S: s230
18  O: o232 (((segment1) simulate-segment)
19  =>G: g238 (operator no-change)
20  F: p245 (simulate-segment)
21  S: s255
22  O: o257 (compute-resources)
23  O: o272 (update-resources)
Build:p277
24  O: o278 (((segment2) simulate-segment)
25  =>G: g280 (operator no-change)
26  F: p287 (simulate-segment)
27  S: s297
28  O: o299 (compute-resources)
29  O: o314 (update-resources)
Build:p319
Build:p320
Build:p323
Build:p324
Build:p325
Build:p326
Build:p327
31  =>G: g331 (operator tie)
32  F: p332 (selection)
33  S: s339
34  O: o344 (((brown) (traveler) (trip1) simulate-trip) evaluate-object)
35  =>G: g363 (operator no-change)
36  F: p83 (task)
37  S: d373
38  O: o374 (((brown) (traveler) (trip1) simulate-trip)
39  =>G: g396 (operator no-change)
40  F: p403 (simulate-trip)
41  S: s414
42  O: o416 (((segment1) simulate-segment)
43  =>G: g422 (operator no-change)
44  F: p429 (simulate-segment)
45  S: s439
46  O: o441 (typewriters 200.0 0.2 1000.0 load)
47  O: o443 (compute-resources)
48  O: o464 (update-resources)
Build:p469
49  O: o470 (((segment2) simulate-segment)
50  =>G: g474 (operator no-change)
51  F: p481 (simulate-segment)
S: 491
O: 493 (compute-resources)
O: 508 (update-resources)
Build:p513
Build:p514
O: 515 ((segment3) simulate-segment)

55
56
57
58
59
60
Build:p558
Build:p559
O: 560 ((segment4) simulate-segment)

61
62
63
64
65
66
Build:p609
Build:p610
O: 619 (trip-succeeds)

68
Build:p613
Build:p614
Build:p615
Build:p616
Build:p617
O: 630 ((gray) (piper) (trip2) simulate-trip)

69
70
71
72
73
74
75
76
77
78
Build:p668
79
Build:p670
O: 687 ((segment2) simulate-segment)

80
81
82
83
84
Build:p730
Build:p731
O: 732 ((segment3) simulate-segment)

85
86
87
88
89
Build:p775
Build:p776
O: 777 ((segment4) simulate-segment)

91
92
93
94
95
96
97
Build:p828
Build:p829
O: 830 ((segment5) simulate-segment)

98
99
100
101
102
103  O: o855 (compute-resources)
104  O: o876 (update-resources)
Build:p881
Build:p882
105  O: o883 ((segment6) simulate-segment) =>G: g885 (operator no-change)
106  P: p892 (simulate-segment)
107  S: s902
108  O: o904 (newsprint 400.0 0.2 6000.0 load)
109  O: o906 (compute-resources)
110  O: o931 (update-resources)
Build:p936
Build:p937
111  O: o938 (weight-limit-exceeded local-constraint-failed)
Build:p943
112  O: e365 (failure final evaluate-state)
Build:p945
Build:p946
Evaluation of operator c374 (simulate-trip) is failure
115  O: o346 (((gray) (piper) (trip1) simulate-trip) evaluate-object) =>G: g948 (operator no-change)
116  P: p83 (task)
117  S: d958
Firing 119:915 p946
119  O: o959 ((gray) (piper) (trip1) simulate-trip) =>G: g981 (operator no-change)
120  P: p988 (simulate-trip)
121  S: s999
122  O: o1001 ((segment1) simulate-segment) =>G: g1007 (operator no-change)
123  P: p1014 (simulate-segment)
124  S: s1024
125  O: o1026 (typewriters 200.0 0.2 1000.0 load)
126  O: o1028 (compute-resources)
127  O: o1049 (update-resources)
Build:p1054
130  O: o1055 ((segment2) simulate-segment) =>G: g1059 (operator no-change)
131  P: p1066 (simulate-segment)
132  S: s1076
133  O: o1078 (compute-resources)
134  O: o1093 (update-resources)
Build:p1098
Build:p1099
136  O: o1100 ((segment3) simulate-segment) =>G: g1104 (operator no-change)
137  P: p1111 (simulate-segment)
138  S: s1121
139  O: o1123 (compute-resources)
140  O: o1138 (update-resources)
Build:p1143
Build:p1144
142  O: o1145 ((segment4) simulate-segment) =>G: g1147 (operator no-change)
143  P: p1154 (simulate-segment)
144  S: s1164
145  O: o1166 (typewriters 200.0 0.25 1000.0 unload)
146  O: o1168 (compute-resources)
147  O: o1189 (update-resources)
Build:p1194
Build:p1195
149  O: o1004 (trip-succeeds)
Build:p1198
Build:p1199
Build:p1200
Build:p1201
Retracting 150:1170 p946
Build:p1202
150  O: o966 ((brown) (traveler) (trip2) simulate-trip)
Evaluation of operator c959 (simulate-trip) is success

Evaluation of operator c182 (simulate-trip) is partial-success
48

Firing 197:1573 p323
Firing 197:1573 p327
Firing 197:1573 p324
Firing 197:1573 p325
Firing 197:1573 p326
Firing 197:1575 p946
Firing 197:1575 p1535
197 0: o115 ((gray) (pipe) (trip) simulate-trip)
Firing 198:1580 p1198
Firing 198:1580 p1202
Firing 198:1580 p1201
Firing 198:1580 p1199
Firing 198:1580 p1200
Retracting 198:1582 p946
Retracting 198:1582 p946
198 0: o112 ((brown) (traveler) (trip) simulate-trip)
Firing 199:1587 p1521
Firing 199:1587 p1525
Firing 199:1587 p1522
Firing 199:1587 p1523
Firing 199:1587 p1524
199 0: ol542 (no-trips-left schedule-succeeds)
200 0: ol02 (final-state)
Space task succeeded in goal q76.
Build:pl548
Build:pl549
201 O: o6 (halt)
Applied task operator o69 (do-task). Final state is s73.
End -- Explicit End
nil
<cl> (list-chunks "/afs/cs.cmu.edu/user/altmann/taql/truck/chunks.soar")
Copying chunks to file /afs/cs.cmu.edu/user/altmann/taql/truck/chunks.soar.

<cl> (print-stats)
Soar (Version 5, Release 2)
Created August 26th, 1991

Run statistics on June 3, 1992
Allegro CL 3.1.12.2 [DECstation] (11/19/90) DECstation id: 385 Ultrix TRICERATOPS.SOAR.CS.CMU.EDU
362 productions (6491 / 27250 nodes)
69 chunks (69 / 362 productions)
109.383 seconds elapsed 25.353 seconds chunking overhead
202 decision cycles (541.50006 ms per cycle)
707 elaboration cycles (154.71428 ms per cycle)
(3.5 cycles/d cycle)
2040 production firings (53.61912 ms per firing)
2.8854313 productions in parallel
9415 RHS actions after initialization (11.61795 ms per action)
392 mean working memory size (1041 maximum, 350 current)
3347 mean token memory size (9279 maximum, 1802 current)
21814 left tokens added, 19337 right tokens added, 41151 total tokens added
21369 left tokens removed, 17980 right tokens removed, 39349 total tokens removed
80500 token changes (1.358795 ms per change)
(8.549278 changes/action)
nil
<cl> (taql-stats)
TAQL 3.1.4
Created July 15, 1991

TAQL statistics on June 3, 1992
Allegro CL 3.1.12.2 [DECstation] (11/19/90) DECstation id: 385 Ultrix TRICERATOPS.SOAR.CS.CMU.EDU
47 TCs (46 user, 1 default)
compiled into 124 productions (101 user, 23 default)
t
<cl> ;; end of trace
References


