CHAPTER 7  Models of Scheduling
7: Models of Scheduling

7.1 Introduction

When I first started thinking about a model of a problem as complicated as the scheduling problem, I didn’t have a clue where to start. One option, which I quickly dismissed, was to specify a set of production rules that implemented a scheduling algorithm. Such a model might display some expert-like behavior on scheduling, but would not expose any learning. It would also contradict the ideas I started out with in chapter 1, namely that the process of learning is more interesting than the actual problem-solving behavior itself. While carrying out the projects discussed in chapters 4 through 6, however, the contours of a scheduling model started to take shape. In this chapter, I will present a model of scheduling that integrates many of the aspects of learning discussed in the last few chapters.

The starting point for the model of scheduling will be the learning paradigm presented in chapter 6 (figure 6.1). The central idea is that problem solving on a new task starts with some initial method. This method produces instances, examples of how the task is solved that can be retrieved later on. Another possible product of the initial method are abstractions, declarative representations of how the problem can be solved. Abstractions can be retrieved and applied to new cases of the problem, and during this application, production rules can be compiled. In the model of the Fincham task in chapter 6, abstractions used a representation that was specific to the task, and always involved the same kind of operation: adding or subtracting days or times. In this chapter, I will develop a generalized abstraction representation, to be used for any type of operation. This is necessary, since the strategies for the scheduling task are not fixed in advance. Another advantage of generalized abstractions is that explicit learning strategies can now operate on these abstractions. As a consequence, production rules themselves are no longer a product of explicit learning, and production compilation can be seen as a truly implicit learning mechanism.

Besides learning, there is another important aspect of the data discussed in chapter 3 that I will investigate in the model, the issue of individual differences. Individual differences have many sources. One source is differences in knowledge. If an individual does not know all the addition facts, they have to do addition by counting. This slows down the problem solving process, or may even disrupt it if working-memory capacity is exceeded. Or an individual may use a particular trick to solve a certain problem, which is not available to other individuals. I will not explore this source of individual differences in this chapter, although the reader might want to refer to the discussion of discrimination-shift learning in chapter 5 for an example. Another source of individual differences is the ability to retain elements in working memory. Although ACT-R does not model working memory explicitly, what is normally referred to as working-memory capacity is closely related to the source activation parameter in ACT-R (Lovett, Reder, & Lebiere, 1997). Source activation is the amount of activation that spreads from the chunks that are part of...
the goal context to other chunks (figure 2.8). Lovett et al. have shown that varying
the parameter between 0.7 and 1.4 with a mean of 1.0 can explain individual
differences in the digit working memory task (Yuill, Oakhill, & Parkin, 1989). In this
task, participants have to read aloud a number of strings of digits that appear on the
screen. Their goal is to memorize the last digit in each string, and reproduce them
after all strings have been read. Both the number and the length of the strings can be
varied. Working-memory capacity is quite relevant in the scheduling task, since
many aspects of the task have to be retained in memory at the same time. We will
therefore look at changes in performance in the model when the source activation
parameter is varied.

A final factor that has to be taken into consideration is randomness in choice. In
ACT-R, noise is involved in almost any choice that is made. This means that ACT-R
predicts that even if participants could be brought into exactly the same situation
twice, they would not necessarily make the same choice twice.

7.2 Generalized abstractions

Abstractions in the Fincham model consist of two parts: a specification of what the
goal has to be like, for example the sport is hockey and we are looking at the day of
the week, and the operation that has to be performed, for example plus2. This
operation has two aspects: on the one hand the plus2 operation has to be
performed, involving either retrieval or subgoaling, and the answer has to be stored
in the goal. The generalized version of abstractions will have the same components,
but will separate out the two aspects of the operation. Furthermore, in the Fincham
model abstractions relied on task-specific rules to retrieve and apply them.
Generalized abstractions will need no task-specific knowledge, but are retrieved
and applied by general purpose productions only. In this section, I will describe the
representation and use of abstractions in general terms. A more elaborate
discussion, which will take care of all the details, can be found in section 7.8.

Representation of an abstraction
The main four components of a generalized abstraction are the following:

1. The type of goal the abstraction can be used for
2. The type of fact that needs to be retrieved
3. A test that is performed on the goal and the retrieved fact
4. An action, which specifies what to do with the retrieved fact and the goal

A generalized version of a Fincham abstraction may therefore look like this:
In order to use an abstraction, it has to be interpreted by production rules. This involves a number of steps, the main ones of which are depicted in figure 7.1. The first step is to retrieve an abstraction that is applicable to the current goal. This abstraction is stored in the current goal. The second step is to retrieve a fact as specified in the abstraction, satisfying the test in the abstraction. In the example Fincham abstraction, a fact of type plus2 is needed in which the argument matches the day in the goal. Finally, the action is carried out: the retrieved fact has to be used to modify the current goal. In the Fincham example, the answer of the plus2 fact needs to be stored in the second-day slot of the goal.

This description looks conspicuously like a description of a production rule, but this is intentional. An abstraction is more or less the declarative counterpart of a production. But since it is declarative, it can be inspected, reasoned with explicitly, and manipulated. On the other hand it has to be interpreted by production rules in order to be executed. While abstractions offer flexibility, production rules offer speed: the whole cycle in figure 7.1 can be done in one step by a task-specific production rule. If both speed and flexibility are needed, both representations can be retained, but if flexibility is no longer necessary, the abstraction may be forgotten.

Using this dual representation of knowledge corresponds directly with theories about skill learning. For example Fitts (1964, cited in Anderson, 1995) discerns three stages in skill learning: a cognitive stage, an associative stage and an autonomous stage. In the cognitive stage, declarative representations (in our case abstractions),
acquired through instructions or examples, are interpreted. In the associative stage, the skill is in transition between a declarative and a procedural representation (abstractions are available, productions only partially). In the autonomous stage the skill is proceduralized completely, and sometimes the ability to verbally describe the skill is lost (all productions are learned, abstractions are forgotten). Anderson has also adapted Fitts’ general skill learning theory for the ACT theories when he developed ACT* (Anderson, 1983). In the chapter about procedural learning in The Architecture of Cognition he already discusses the need for general interpretive productions in a description of how skill learning in ACT* can be accomplished. This skill acquisition aspect has, however, not been elaborated yet in terms of the ACT-R theory.

Chaining abstractions

An abstraction can be considered as a sort of plan for what to do. The example in the previous section was a simple one-step plan. But sometimes a number of steps have to be carried out in a certain order. To allow multi-abstraction plans, two extra slots have been added to the abstraction: a prev slot and a fail slot. These two slots are used to link abstractions into lists of abstractions. Each time an abstraction is completed successfully, a next abstraction is retrieved following the prev links. If an abstraction somehow fails, the next abstraction is retrieved following the fail links.

Figure 7.2 shows an example of a plan that a participant might have in a standard Sternberg memory experiment (Sternberg, 1969). In this type of experiments, the participant first has to memorize a set of letters, the memory set. Subsequently, new letters are presented to the participant, and they have to decide as quickly as possible whether or not the letter is in the memory set. Figure 7.2 shows the plan for this decision process. Each circle represents an abstraction, and the arrows show how they have been linked. In the first abstraction, the letter is read and stored in the goal. In the next step, the second abstraction from the left, a letter from the memory set is retrieved (hopefully the right one). If this already fails, a response of “no” is given.
following the “fail”-link. If a letter is retrieved, the third abstraction checks whether both letters are equal. If this succeeds, the response is “yes”, else it is “no”.

Proceduralizing abstractions
Since an abstraction has a function that is quite similar to a production rule, it is not so hard to proceduralize. The same method is used as discussed in the previous chapter (figure 6.2). Each time an abstraction is retrieved, there is a possibility that a dependency will be pushed as a subgoal, and the four steps in which the abstractions are carried out will be compiled into a single production rule. Due to the use of generalized abstractions, it is no longer necessary to have explicit learning strategies that are activated when a dependency goal has been pushed. The explicit strategies can now operate at the level of abstractions, independently of the production-compilation process. By pulling explicit learning strategies out of the dependency subgoal, the actual process of building dependencies can be carried out by a fixed set of production rules, more or less as a mechanism of the architecture.

7.3 A first model
A first approximation of a model of scheduling has the following components:

1. Production rules that interpret and proceduralize abstractions as outlined in the previous section
2. A top-goal that reads the constraints for the current problem from the screen and pushes a task subgoal upon the goal stack. After the subgoal has successfully terminated, it outputs the answer found in the subgoal.
3. Productions that store elements in a list, and implement rehearsal, both maintenance rehearsal and elaborate rehearsal.
4. A set of abstractions that implements a simple strategy for scheduling.
5. Productions that produce some sort of verbal protocol.

The first item on the list has already been discussed, and the top-goal productions are quite trivial, so I will only elaborate on the last three items of the list.

Storing elements in a list and doing rehearsal
In chapter 4, I discussed a model of rehearsal based on Baddeley’s phonological loop. As we have seen in the protocols of scheduling, participants maintain a list of the partial solution, which they rehearse from time to time. Rehearsal can have two functions: maintenance rehearsal to keep the activation of the list high enough, and elaborate rehearsal to do additional processing on the items in the list. The first model will use elaborate rehearsal to calculate the total duration of the tasks in the
A first model

list. Instead of using an explicit phonological loop, as in chapter 4, ordinary ACT-R chunks are used to represent the list. Details of the implementation can be found in section 7.8.

Abstractions that implement a simple strategy
The first model uses a simple, one shot strategy that involves the following steps:

1. A first task for the schedule is selected by retrieving an order-constraint and picking the first task in this constraint. For example, if ‘D before C’ is a constraint, D is picked as a possible first task. It is then checked if there is no earlier task, indicated by a constraint like ‘A before D’. If that is the case, the earlier task is substituted as candidate first task, else D is accepted as first task.

2. The next task is determined by finding an order constraint that specifies a fact that is later than the task we have just added to the schedule. So if the schedule starts with D, and ‘D before C’ is a constraint, we add C. Repeat this step until no more tasks can be added using this method.

3. Now count how many hours the tasks in the current schedule take (using elaborate rehearsal, as explained above).

4. Calculate how many hours are left for one worker. So, if the tasks currently in the schedule take four hours, and each worker has six hours, two hours are left. If the number of hours left is greater than zero, find a task that has a duration of exactly that number of hours and add it to the schedule.

5. Move to the next worker.

6. Go through the list of all the tasks, and add those to the schedule that are not already allocated to the first worker.

Verbal protocol
An assumption about abstractions is that they can be reasoned about, so they are available to verbalization in a think-aloud experiment. To avoid writing a language-production model, a “verbalization” string is added to each abstraction that describes the action performed by the expression. Whenever an abstraction is executed, this string is added to the verbal protocol. Rehearsal actions also produce verbal protocol, as do reading actions. The verbal protocol not only enables producing “Turing Test”-like results, but is also very useful in debugging the model. Although a fully-fledged language production module will probably require a formidable modeling effort, it may be a very useful tool in a continued research effort on declarative rules.

Results of the model
The model was tested using a set of ten example problems, all of which consisted of two workers and six or seven tasks. Although the problems are not particularly hard, this is not yet important since the answer given by the model is not checked.
The model uses only symbolic learning, and has all subsymbolic learning turned off. New chunks in declarative memory do not have a role in the problem solving process yet. Improvements in performance can therefore be attributed to production compilation. Figure 7.3 shows the learning curve of the model. The graph also shows the data from chapter 3 in comparison (actually the lower-left panel of figure 3.4 multiplied by the average solution time; the data start at problem 2, because participants have already solved one two-worker example problem). Although the data from the model and the experiment cannot be compared properly because different problems have been used, the graph shows the same logarithmic curve for both the model and the data. To get some idea of the rate of learning, the growth in the number of productions is plotted in figure 7.4. The more interesting part is the pseudo verbal protocol produced by the model. To see the impact of procedurization, examples of the output of the first and the tenth problem have been printed in figure 7.5. Clearly, the protocol of the first problem is a protocol analyst’s dream, because participants are hardly ever that precise. But the tenth protocol looks more familiar: many steps in the process are omitted, and we
A first model

Protocol of first problem

There are two workers. Each of the workers has seven hours. Task A takes two hours. Task B takes two hours. Task C takes two hours. Task D takes three hours. Task E takes three hours. Task F takes three hours. Task B before F. Task F before A. First I will find a task to begin with. Let’s look at an order constraint. B before F. Let’s see if there is no earlier task. There is no earlier task. Begin with B. B Can we find a next task just by looking at the order? B before F. Can we find a next task just by looking at the order? F before A. Can we find a next task just by looking at the order? Is this a schedule for one worker or for more? Now I am going to count how many hours we already have B. How long does this one take? Task B takes two hours. Add this to what we have. nothing plus two equals two. F. How long does this one take? Task F takes three hours. Add this to what we have. Two plus three equals five. A. How long does this one take? Task A takes two hours. Do we have enough for one worker? Each worker has seven hours. We can move to the next worker.. B.. F. A. next. Let’s do the rest. Now we are going to look at all the tasks, and see which ones are not yet in the schedule. Let’s start with A. Task A takes two hours. Let’s try to put it in the schedule. A is already in the schedule. OK, what is the next letter? B comes after A. Task B takes two hours. Let’s try to put it in the schedule. B comes after A. B is already in the schedule. OK, what is the next letter? C comes after B. Task C takes two hours. Let’s try to put it in the schedule. C comes after B. B.. F. A. next.. C. OK, what is the next letter? D comes after C. Task D takes two hours. Let’s try to put it in the schedule. D comes after C. D.. E.. A. next.. D. E. Task E takes three hours. Let’s try to put it in the schedule. E comes after D. E is already in the schedule. OK, that was the last task, we’re done! The answer is B F A next C D E

Protocol of tenth problem:

There are two workers. Each of the workers has six hours. Task A takes one hours. Task B takes one hours. Task C takes two hours. Task D takes three hours. Task E takes three hours. Task D before E. Task E before A. First I will find a task to begin with. Let’s see if there is no earlier task. Begin with D. D.. D. E. D. A. next. Can we find a next task just by looking at the order? Is this a schedule for one worker or for more? Now I am going to count how many hours we already have D. E. A. next.. B. D. E. A. next.. B. C. D is already in the schedule. E is already in the schedule. D. E. A. next.. B. C. F. OK, that was the last task, we’re done! The answer is D E A next B C F

Figure 7.5. ACT-R protocol of the first and the tenth problem of a sample run

can only guess why some decisions have been made. This concurs with the general idea that proceduralized skills produce no verbal protocol (Ericsson & Simon, 1984; van Someren, Barnard & Sandberg, 1994).

Although this first model shows some interesting properties similar to real problem-solving behavior, it is far from complete. The current model just takes a single shot at the solution, and does not retry if it is incorrect. Only production compilation had been turned on, so the model will never forget any intermediate results it has found. And finally, the model starts out with a set of task-specific abstractions. One of the desired properties of the model was to start without any task-specific knowledge. These issues will be addressed in the second version of the model. But first the most important of these issues will be discussed separately: where do abstractions themselves come from?
7: Models of Scheduling

7.4 Learning new abstractions

Figure 6.1 specifies the first step of learning a new skill as the ‘initial method’. In the Fincham task the initial method was analogy, because this method had been explained as part of the experiment. In general, analogy is a strategy that takes knowledge from another domain, and modifies this knowledge to suit the current task.

Work by Sander and Richard (1997) indicates that people use the analogy strategy in discovering knowledge for a new domain. In their experiment, participants without any computer experience had to learn to operate a word processor. Since participants did not get any instructions on how to operate the word processor, they had discover the functions by themselves. The tasks participants had to do was to modify a given text so that it would be identical to another text. The total set of possible operations to change text was classified into four levels. Level 1 consists of operations that are also possible on a standard typewriter, a device participants were all familiar with, for example typing new letters, and deleting the last character typed. Level 2 operations are operations that are not possible on a typewriter, but can be considered as normal in the domain of writing, for example inserting a word in a sentence. Level 3 operations come from the even more general domain of object manipulation, for example copying a word and pasting it somewhere else. Finally, level 4 operations are operations not related to any domain. An example of a level 4 operation is to copy strings of spaces. A space is not an object in the real world, so the specific knowledge that a space in a word processor is like any other character is required.

In the experiment participants were strongly encouraged to discover new methods, since each time they tried a method they used before, they were prompted to attempt another method to solve that particular problem. As it turned out, all participants used level 1 operations immediately from the start of the experiment. As the experiment progressed, they gradually discovered level 2 operations, followed by level 3 operations. Level 4 operations were only discovered by a minority of the participants, and only in the last few sessions of the experiment.

The results of this experiment support the idea that when people are in a new situation, they adapt knowledge from a similar domain to initially guide their actions. In word processing, knowledge of a typewriter is the most immediate source. If that source of knowledge is exhausted, knowledge of writing in general can be used, followed by the even more general knowledge source of object manipulation.

In the scheduling task, analogy is also a good starting point. People may not know anything about schedules, but they do know something about lists, and how to construct them. Suppose we need to make a schedule. We may use knowledge about
lists to start with. How do we make a list? First we have to find a first item for the
list, a beginning. Once we have a beginning, we find a next task until we are done.
But how do we find something to begin with, and how do we find a next task? We
may choose to handle these problems by making them subgoals, or we may try to
find mappings between ‘beginning’ and ‘next’ and terms in the scheduling problem.
For example, a mapping can be made between ‘next’ and an order-constraint in the
scheduling problem. The result is a modified version of the list-building
abstractions, with ‘list’ substituted by ‘schedule’ and ‘next’ substituted by ‘order’.
Note that for sake of the explanation, the terms ‘list’, ‘beginning’, ‘before’ and ‘next’
will be used to refer to general terms, and ‘schedule’ and ‘order’ to refer to task-
specific terms. Except for knowledge on how to build a list, the analogy between a
schedule and a list may also offer knowledge on how to retain a list in memory by
rehearsal.

Although these new abstractions may find a start for a schedule, they are not
sufficient to build a complete schedule, mainly because the mapping between ‘next’
and ‘order’ is inadequate. When this abstraction fails to make a complete schedule,
another plan may take over and contribute to the schedule.

An idea that may take over if the list-building plan fails to add any more tasks to the
schedule is the plan that tries to complete the first worker. A useful general plan may
state that whenever something has to be completed, the difference between the
desired size and the current size has to be calculated, after which an object has to be
found with a size equal to this difference.

The central emerging idea is therefore that several strategies from similar domains
are adopted and patched together. This method of adapting old strategies to new
purposes is similar to the script and schema theories, as proposed by Schank (Schank
& Abelson, 1977). Traditional script and schema theories assume that a complete
script is first adapted to fit the current task, and then carried out. The ACT-R model
uses a more on-demand style of adaptation: a new abstraction is created at the
moment it is needed. Again, the details may be found in section 7.8.

### 7.5 The second model

The second model solves some of the shortcomings of the first. It learns new
abstractions as outlined in the previous section. Furthermore, the following aspects
have been added to the model:

1. After a solution has been produced by the model, it receives feedback from the
interface. If the solution is incorrect, the model has the opportunity of reading
the violated constraint, and has to attempt a new solution. If no solution has
been found after 300 seconds, the model has to move on to the next problem.
7: Models of Scheduling

This 300 second boundary is somewhat arbitrary, but gives an opportunity to assess the accuracy of the model: if the model cannot solve it within the allotted time, it is counted as a failure.

2. Base-level learning is turned on. As a consequence, the model can forget all kinds of partial results it derives, most notably the list that contains the partial solution, but also read constraints (which have to be reread in that case), newly derived abstractions, etc. To recover gracefully from all kinds of errors that can occur due to forgetting, the robustness of the model had to be increased. The result of an error is often that a subgoal is popped in failure. This means that if a goal pushes a subgoal, it sometimes has to check whether or not this subgoal has actually succeeded. This is especially important if production compilation is involved, since this may result in learning a faulty production rule that gets ACT-R into endless loops. A base-level decay of 0.5 is used, the recommended value, from which I diverged in the Tulving and Fincham model. No long-term effects of learning were investigated in this model, so there was no need for a smaller decay.

3. The model uses the order in which constraints are presented on the screen. For example, if a task has to be found that takes 3 hours and is not yet present in the current schedule, the list on the screen is used to find the first task taking 3 hours. If that task is already in the schedule, the next 3 hour task is looked for on the screen, etc.

4. Several extra abstractions have been added to ensure that correct solutions are eventually found by the model. The model now tries to satisfy the order constraints for the second worker as well, and uses the feedback it gets when it makes an error as a starting constraint for the next try.

Example verbal protocol

The following protocol fragment, produced by the model, gives an impression of the additional aspects of the model:

There are two workers. Each of the workers has six hours. Task A takes one hours. Task B takes one hours. Task C takes two hours. Task D takes two hours. Task E takes three hours. Task F takes three hours. Task B before C. Task F before A. I have to think of some new way to find a schedule. Let's use what I know about lists. First I will find something to begin with. Let's look at a before constraint. I have to think of some new way to find a before. Let's use what I know about order. Let's use a order fact as a before fact. F before A. I have to think of some new way to find a before following abstraction12. F before A. I have to think of some new way to find a before following abstraction12. Let's look at a before constraint. Let's see if there is no earlier element. Let's use a order fact as a before fact. There is no earlier element.

This doesn't work. Let's start again. First I will find something to begin with. Let's look at a before constraint. Let's see if there is no earlier element. Let's use a order fact as a before fact. There is no earlier element. Begin with F. F. I have to think of some new way to find a schedule following abstraction10. Now I have to find the next thing. F before A. A. I have to think of some new way to find a schedule following abstraction17. Now I have to find the next thing. No more items for the list, let's check whether we're done. F. A.. Is this a schedule for one worker or for more? Now I am going
The second model

to count how many hours we already have F.. How long does this one take? Task F takes three hours. Add this to what we have. Nothing plus three equals three. A.. Add this to what we have. Three plus one equals four. Do we have enough for one worker? No, the schedule is not full, yet. I have to think of some new way to find a total. Let’s use what I know about hours. Let’s use a hours fact as a total fact. Each worker has six hours. I have to think of some new way to find a total following fail-abs23. Each worker has six hours. I have to think of some new way to find a find-remain following a failure. This doesn’t work. Let’s start again.

First I will find something to begin with. Let’s look at a before constraint. Let’s see if there is no earlier element. Let’s use a order fact as a before fact. Let’s see if there is no earlier element. Let’s use a order fact as a before fact. There is no earlier element. Begin with F. F.. Now I have to find the next thing. F before A. A.. Now I have to find the next thing. Now I have to find the next thing. No more items for the list, let’s check whether we’re done. F.. A.. Is this a schedule for one worker or for more? Now I am going to count how many hours we already have F.. Add this to what we have. Nothing plus three equals three. A.. Add this to what we have. Three plus one equals four. Do we have enough for one worker? No, the schedule is not full, yet. Let’s use a hours fact as a total fact. Each worker has six hours. How many hours are there left? Two plus four equals six. F.. A.. Now find the task that fits in. Task C takes two hours. C.. We can move to the next worker.. NEXT-WORKER Let’s do the rest. F.. A.. C.. NEXT-WORKER.. I now try to find any unused order constraints. B before C. This one hasn’t been used, so the constraint has been found. B.. Now we are going to look at all the tasks, and see which ones are not yet in the schedule. Let’s start with A. A is already in the schedule. B is already in the schedule. Let’s move on to the next task. C is already in the schedule. Task D takes two hours. D.. Task E takes three hours. E.. Task F takes three hours. OK, what is the next task? OK, that was the last task, we’re done! This doesn’t work. Let’s start again.

[one more failed search episode]

First I will find something to begin with. Begin with F. F.. Now I have to find the next thing. F before A. A.. Now I have to find the next thing. No more items for the list, let’s check whether we’re done. F.. A.. Is this a schedule for one worker or for more? Now I am going to count how many hours we already have F.. Add this to what we have. Nothing plus three equals three. A.. Add this to what we have. Three plus one equals four. Do we have enough for one worker? No, the schedule is not full, yet. F.. A.. Now find the task that fits in. Task C takes two hours. C.. We can move to the next worker.. NEXT-WORKER Let’s do the rest. F.. A.. C.. NEXT-WORKER.. I now try to find any unused order constraints. B before C. B before C. This one hasn’t been used, so the constraint has been found. B before C. B before C.. Now we are going to look at all the tasks, and see which ones are not yet in the schedule. Let’s start with A. Task A takes one hours. A is already in the schedule. OK, what is the next task? Task B takes one hours. B is already in the schedule. Let’s move on to the next task. OK, what is the next task? Task C takes two hours. C is already in the schedule. Let’s move on to the next task. OK, what is the next task? Task D takes two hours. D.. Let’s move on to the next task. OK, what is the next task? Task E takes three hours. E.. Let’s move on to the next task. OK, what is the next task? Task F takes three hours. F is already in the schedule. Let’s move on to the next task. OK, what is the next task? OK, that was the last task, we’re done! F.. A.. C.. NEXT-WORKER.. B.. D.. E.. The answer is F A C.. NEXT-WORKER B D E

The particular fragment contains five search episodes, only four of which are shown: the first three and the final, successful episode. In the first two fragments, the model is busy figuring out how aspects of the problem can be mapped onto things it knows something about. Unfortunately, the primitive protocol generating part of the model produces some awkward sentences with references to internal symbols. Somewhere along the line the model gets stuck, because it can not keep track of all the constraints.
in the task and all the newly derived abstractions. The third search episode is slightly more successful: it can use the abstractions derived in the first two episodes. Unfortunately, it fails just when it is done, because it cannot retrieve the start of the list anymore for typing in the answer. In the fifth, successful episode some of the earlier derived results can be retrieved. For example, the model immediately starts with “Begin with F” instead of deriving this fact.

The separate search episodes are similar to the episodes participants showed in the experiment (see chapter 3). Once the model gets stuck, it often does not have knowledge to repair the situation other than starting again. In the new search knowledge derived in the earlier episode is sometimes retrieved, so failed episodes do contribute to eventual success. This concurs with the behavior of participants, since they also are hardly ever able to recover from an error in their reasoning process.

**Results of the model**

Figure 7.6 shows the basic, averaged, results of the model. The solution times and the number of learned production rules are similar to the results of the first model. The improvement in solution time is accompanied by an improvement in the proportion of the problems that is solved correctly within 300 seconds, which I will refer to as ‘accuracy’ in the rest of the chapter.
Figure 7.6 shows a gradually improving learning curve. But as we have seen in chapter 5, the averaging process may smooth out discontinuities that may be present in the data of individuals. Figure 7.7 shows solution times of six individual runs of the model, and the results of six participants from the experiment in chapter 3. Neither the model nor the data shows a smooth improvement of performance, only after averaging results is such a result obtained. Again, the comparison between model and data is only an approximation, since different problems were used.

As mentioned in the introduction of this chapter, the source activation (W) parameter is associated with individual differences in working-memory capacity. Since the scheduling task requires participants to keep many aspects of the task in memory at the same time, it should be quite sensitive to changes in this parameter. Working-memory capacity, as modeled by source activation in ACT-R, is not a
buffer of limited size, but rather the capacity to increase activation of currently relevant memory chunks. As the number of relevant chunks increases, the potential for errors increases. Sometimes it is possible to recover from an error, by recalculating the lost fact, but sometimes the information is lost. The number of errors is a non-linear function of the number of currently relevant chunks. Figure 7.8 illustrates this aspect: it shows the results of a small ACT-R model that stores between three and twelve items and then tries to retrieve them. The graph shows the proportion of correct retrievals for three different values of W. The model is allowed a single rehearsal for each item. As can be seen in the graph, at some point the probability of correct retrieval decreases dramatically. For the average W=1 case, this point is around the “magical number seven”, and the low and high W cases roughly represent “plus or minus two.” This decrease in performance is not caused by the fact that some activation resource must be distributed over a number of chunks “in working memory”, but is rather an emergent fact of several aspects of processing at the same time. The real limited resource is time: as more chunks are relevant, less time can be spent on each of them individually. A higher source activation just makes it possible to retrieve chunks that were accessed longer ago. The result is a model of a limited capacity without resources.

The following metaphor may clarify this issue. Suppose you are a baby-sitter and you look after number of small children. To prevent children from getting up to mischief, you have to pay attention to them. You can only pay attention to one child at a time. As long as a child has had your attention not too long ago, it will behave properly. But if ignore a child too long, it will start misbehaving. If you only have a few children too look after, you will have no problems. Any mischief can be corrected easily by giving a little more attention to the particular rascal. But as the
number of children rises, giving more attention to one child means neglecting the
others, causing more and more trouble. So, at a certain number of children, it
becomes almost impossible to keep them all happy. Now if you are a particular good
baby-sitter, you can give them impressive talkings-to, so they will keep from
mischief just a little longer. Or, if you know the particular children, you might know
a few tricks to keep a particular child happy.

In the baby-sitter example, baby-sitters have no particular hard limit or capacity of
children they can keep happy. Neither do they spread attention to all children at the
same time. They just go from child to child and hope for the best. Individual
differences between baby-sitters are reflected by the impact their attention has on the
children: better baby-sitters don’t have more time, they just use it more effectively.

To see the impact of the W parameter, the model was run several times with source
activations ranging from 0.7 to 1.4, the range that covers all subjects in the Lovett et
al. experiment. Figure 7.9 shows that source activation has indeed a high impact on
performance. A low source activation implies longer solution times and a lower
accuracy. The interesting thing about the accuracy is, however, that the differences
are initially very large: for the first problem, the accuracy of the high source
activations is almost perfect, as opposed to the very poor accuracies for low source
activations. But as learning progresses, these poor accuracies improve dramatically
and by the tenth problem are almost as good as the higher source activations. This
corresponds well with the experiment, in which almost every participant eventually
managed to solve the problems, although the time they needed to do this (so the
number of opportunities for learning), differed tremendously. A tentative
conclusion of this model may therefore be that practice eventually overcomes poor
working-memory capacity.

Is proceduralization necessary for mastering complex skills?
In chapter 1, the hypothesis was posed that mastering a complex skill is a gradual
process, in which some cases of a problem can be solved directly, some need
additional search, and some cannot be solved due to the fact that this would take
too much time. In the scheduling model, a similar issue turns up: if part of a skill is
not proceduralized, it puts extra demands on working-memory capacity, and limits
the amount of other non-proceduralized activity. As a consequence, as working-
memory capacity is lower, more proceduralization (i.e., practice) is needed before a
task can be performed successfully. Working-memory capacity more or less defines
how broad the small grey band in figure 1.4 is. The results in figure 7.9 show that
the accuracy for the higher source activations is close to 1 for the very first problem.
If source activation is lower, practice is needed before a high accuracy is reached.

Figure 7.10 shows a graphical impression of the consequences of limited working-
memory capacity for the scheduling task, analogous to figure 1.4. The rectangle
represents all skills involved in scheduling. At the bottom of the rectangle is a black region which represents the some basic skills that even novices in scheduling already possess, such as reading the screen, building lists and doing rehearsal. Using these skills does not require any extra working-memory capacity. Skills in the light grey area do require working-memory capacity. In terms of the model, these skills use chunks that represent the list, but also abstractions that are used in the reasoning.

Figure 7.9. Solution time and proportion solved correctly for source activations ranging from 0.7 to 1.4
process. The white area represents skills, or groups of skills that take too much working-memory capacity. The dark grey circles, finally, represent the skills of doing the scheduling problems the model has to solve. The top part of the figure shows a case of high working-memory capacity (W=1.4). The scheduling problems are already within the grey area, so can immediately be solved by the model. The only advantage of procedural learning is that the solution time decreases. When source activation is low, on the other hand, the skill of solving the scheduling problems is still in the white area, as shown in the bottom-left part of the figure (W=0.7). In order to be able to solve the problem at all, procedural learning is necessary to reach to get the dark grey circles within the grey band.

To examine more closely whether this is the case in the model, a comparison is made between runs with production compilation turned on and turned off. Figure 7.11 shows the results for source activations 0.6, 0.65, 0.7 and 1.4. Clearly for the lower source activations, production learning is essential for successfully mastering the skill. For W=1.4, on the other hand, procedural learning does not contribute to accuracy at all.
7.6 Some empirical evidence for the scheduling model

Although the models of scheduling presented in this chapter address most of the issues raised in chapter 3, the predictions made by the model have not actually been tested yet. Fortunately, Linda Jongman has recently performed an experiment that provides some experimental support for the model. In a study on mental fatigue, she used the scheduling task as discussed in chapter 3, and the digit working memory task that has been modeled in ACT-R by Lovett, Reder and Lebiere (1997).

The digit working memory task was used to make an estimate of the working-memory capacity of a participant, expressed in the ACT-R source activation parameter. This working-memory capacity was related to the performance on the scheduling task. Unfortunately, the scheduling task as it was used in this particular experiment was a mixture of problems with two and three workers with varying difficulty and varying time limitations. It is therefore hard to compare the results directly to the model predictions. Nevertheless some of the more qualitative predictions of the model can be tested with respect to individual differences.

Figure 7.11. Comparison between procedural learning turned on and off, for different values of source activation
The model predicts a strong correlation between working-memory capacity and the performance on the scheduling task. This proved to be the case in the experiment: the correlation between the estimated source activation and the number of successfully solved schedules is 0.56 (with n=16). This correlation increases to 0.66 if the analysis is restricted to the three-worker schedules, the schedules that require most working-memory capacity. Figure 7.12 shows the scatter plot for this latter relation. A more specific prediction of the model is that the effect of working-memory capacity on performance will diminish due to proceduralization. To investigate this prediction, the group of participants is split into eight low source-activation participants (W<0.95) and eight high source-activation participants (W>0.95). The proportion of correct solutions for each of the groups is plotted in figure 7.13. In this graph only three-workers problems are shown, and to average out part of the noise each data point is averaged with its predecessor and its successor. There is a clear convergence between the two curves, as can be seen in the bottom graph that depicts the difference.

7.7 Discussion

At the end of the previous chapter it seemed that production compilation was an uninteresting optimization of declarative knowledge. The scheduling model shows that this was a false impression. Complex reasoning processes in declarative memory can only become more complex because production compilation decreases demands on working-memory capacity.
The scheduling model also reveals insights into a part of the problem-solving process that is usually not part of cognitive models: the acquisition of task-specific rules from instructions. Although the model does not encompass a natural language parser, it is easier to imagine translating an instruction into a list of abstractions than into a set of production rules.

The abstraction representation chosen for this model is not the only possibility: probably a more general and efficient representation is possible. Optimizing the representation might be a good topic to study in conjunction with a more extensive system for creating new plans using old plans. A more general issue of representation that has become clear in this model is the fact that the degree of freedom ACT-R provides in choosing different types of chunks is probably too great. When general rules have to reason with declarative facts, having too many distinct types is a hindrance. The scheduling model uses only a few chunk types. The
Appendix: Implementation of abstractions in ACT-R

downside of a chunk type that can be used for many purposes is that the number of
slots becomes very large. Many of the slots are only needed at the moment that the
chunk is the current goal, and are irrelevant for retrieval later on. For example, the
generic goal type contains slots to store the current abstraction and the retrieved fact,
and other bookkeeping slots. These slots are not needed anymore once the goal is
popped.

Unfortunately, the model cannot yet solve the hard problems that participants had
to solve in the experiment. The current model, however, shows many aspects also
found in the experiment:
  • Separate search episodes
  • Elaborate and maintenance rehearsal
  • Errors due to limited working-memory capacity
  • Large individual differences
  • Deliberate reasoning about the task
  • Proceduralization of declarative knowledge

A number of issues are still unresolved. The way in which new abstractions are
learned is a good starting point for discovering new strategies, but it is not yet clear
whether that is sufficient to discover complicated strategies like the different-worker
and fit-the-hours strategies in chapter 3. Another issue is the fact that ACT-R only
maintains expected gains of production rules, and that it has no mechanisms to keep
track of the quality of declarative knowledge. Some way has to be found to represent
that, for example, a particular abstraction does not work most of the time.

Although these issues may involve even more explicit strategies, there is no
fundamental problem in resolving them within the current framework. The main
problem lies in the fact that people have a lot of relevant knowledge, even for an
abstract task like the scheduling problem, and it is hard to specify all this knowledge
and put it in a model.

7.8 Appendix: Implementation of abstractions in ACT-R

In this section I will discuss in detail how abstractions work. Readers not interested
in the technical details may skip this section.

The basic generalized abstraction
The basic structure of a generalized abstraction is as follows:
7: Models of Scheduling

GENERALIZED-ABSTRACTION
ISA ABSTRACTION
GOAL the type of goal this abstraction applies to
RETRIEVE the type of fact that needs to be retrieved from declarative memory
TEST constraints the retrieved fact has to satisfy
ACTION how the goal is modified using the retrieved fact

The generalized abstractions have many properties of a typical production rule in ACT-R: the goal has to be of a certain type (GOAL), some fact is retrieved from declarative memory (RETRIEVE) satisfying some condition (TEST), and this fact is used to modify the goal (ACTION). Note that tests on the goal itself are part of the condition in the TEST slot. Suppose the goal of the Fincham task looks like:

EXAMPLE-FINCHAM-GOAL
ISA HOCKEY-DAY-GOAL
DAY1 WEDNESDAY
DAY2 NIL

Further assume there are plus2 facts available of the following form:

EXAMPLE-PLUS2-FACT
ISA PLUS2
ARGUMENT WEDNESDAY
ANSWER FRIDAY

An abstraction that specifies that plus2 facts are needed for the hockey-day-goal looks as follows:

EXAMPLE-FINCHAM-ABSTRACTION
ISA ABSTRACTION
GOAL HOCKEY-DAY-GOAL
RETRIEVE PLUS2
TEST DAY1=ARGUMENT
ACTION DAY2:=ANSWER

In English, the interpretation of this abstraction is:

If the goal is of type hockey-day-goal, retrieve a plus2 fact, so that the content of the day1 slot of the goal is equal to the argument slot of the plus2 fact, and put the contents of the answer slot of the plus2 fact in the day2 slot of the hockey-day-goal.

The representation presented above cannot be used directly. It needs to be interpreted, and this interpretation has to be done by production rules. Production rules, however, cannot inspect the names of slots, nor can the type of the goal (i.e., the contents of the ISA-slot) be variabilized. In order to circumvent this problem, some generalized goal representation is necessary with a fixed amount of slots. The representation I will use is as follows:
Appendix: Implementation of abstractions in ACT-R

EXAMPLE-GENERAL-GOAL
ISA GENERIC
TYPE the type of the goal
SLOT1 a general purpose slot to store results, arguments etc
SLOT2 another general purpose slot
SLOT3 a third general purpose slot
ANSWER the answer, or true to indicate a goal that has succeeded
ABSTRACTION slot to store a retrieved abstraction
RETRIEVE slot to store the retrieved fact
TEST slot to store the test
ACTION slot to store the action

Unfortunately, the general purpose goal has many slots. Especially the abstraction, retrieve, test and action slots are necessary for processing purposes and are useless once the goal is popped.

The generic goal makes it possible to interpret abstractions using ordinary production rules. Let’s look at our Fincham example again, and translate the goals into the generic goal:

EXAMPLE-FINCHAM-GOAL
ISA GENERIC
TYPE HOCKEY-DAY-GOAL
SLOT1 WEDNESDAY
ANSWER NIL
(all other slots are nil)

EXAMPLE-PLUS2-FACT
ISA GENERIC
TYPE PLUS2
SLOT1 WEDNESDAY
ANSWER FRIDAY
(all other slots are nil)

The example abstraction now becomes:

EXAMPLE-FINCHAM-ABSTRACTION
ISA ABSTRACTION
GOAL HOCKEY-DAY-GOAL
RETRIEVE PLUS2
TEST SLOT1=SLOT1
ACTION ANSWER:=ANSWER

As we can see, slot names no longer label what is in a slot, making it slightly harder for us (but not for ACT-R) to interpret the meaning of abstractions and rules. The convention for tests and actions is that in a slotx=sloty or slotx:=sloty construction the slotx part refers to the goal, and the sloty part to the retrieved fact. When used in the test slot of the abstraction, it means that slotx of the goal has to match sloty of the
fact, and in the action slot of the abstraction, it means the contents of sloty of the retrieved fact have to be copied to slotx of the goal. It is important to note that ACT-R does not interpret these test and action instructions: they are just labels that are matched by the appropriate production rules.

Interpretation of an abstraction can be handled by four consecutive production firings. I will present the rules, and step through the Fincham example to illustrate it. First, an abstraction needs to be retrieved:

```
RETRIEVE-ABSTRACTION
IF the goal is a generic goal of type type, and the abstraction slot of the goal is nil
AND there is an abstraction with goal type
THEN put the abstraction in the abstraction slot of the goal
```

This rule will be competing with ordinary task-specific rules, so it should have an expected gain that is not too high. In that case, when task-specific rules perform well and have a high expected gain, they will win most of the time, but when the task-specific rules have a low expected gain, abstraction retrieval will be preferred. This competition is comparable to the competition between search and reflection, as discussed in chapter 5. After the abstraction has been retrieved, the contents of its slots are copied to the goal:

```
COPY-ABSTRACTION-TO-GOAL
IF the goal is a generic goal and some abstraction is in the abstraction slot of the goal
THEN copy the contents of the retrieve, test and actions slots of the abstraction to their respective slots in the goal
```

In the Fincham example, these two rules will retrieve the example-fincham-abstraction and store it in the goal, so the goal will now become:

```
EXAMPLE-FINCHAM-GOAL
ISA GENERIC
TYPE HOCKEY-DAY-GOAL
SLOT1 WEDNESDAY
ANSWER NIL
ABSTRACTION EXAMPLE-FINCHAM-ABSTRACTION
RETRIEVE PLUS2
TEST SLOT1=SLOT1
ACTION ANSWER:=ANSWER
```

Now that the abstraction has been selected, the retrieval specified in its “retrieve” slot has to be carried out:
Appendix: Implementation of abstractions in ACT-R

APPLY-ABSTRACTION-RETrieve-SLOT1-SLOT1
IF the goal is a generic goal with an abstraction in the abstraction slot, and the retrieve slot has type retrieve and the test slot equals slot1=slot1 and slot1 has value slot1 AND there is a fact of type retrieve and value slot1 in slot1 THEN put this fact in the retrieved slot of the goal

This rule is specific to the slot1=slot1 test, so a similar rule is necessary for every possible test. In the Fincham example, this rule will look for a plus2 fact with wednesday as slot1 value, and will find our example-plus-fact, transforming the goal to:

EXAMPLE-FINCHAM-GOAL
ISA GENERIC
TYPE HOCKEY-DAY-GOAL
SLOT1 WEDNESDAY
ANSWER NIL
ABSTRACTION EXAMPLE-FINCHAM-ABSTRACTION
RETrieve EXAMPLE-PLUS2-FACT
TEST SLOT1=SLOT1
ACTION ANSWER:=ANSWER

Sometimes the fact that needs to be retrieved is not available in declarative memory. An alternative method to get a fact is to push it as a subgoal. The following rule accomplishes this for the slot1=slot1 case:

APPLY-ABSTRACTION-SUBGOAL-SLOT1-SLOT1
IF the goal is a generic goal with an abstraction in the abstraction slot, and the retrieve slot has type retrieve and the test slot equals slot1=slot1 and slot1 has value slot1 THEN push as a subgoal a goal of type retrieve and set slot1 to slot1 AND store this subgoal in the retrieved slot of the goal

In the Fincham example, this rule would create the following subgoal:

EXAMPLE-SUBGOAL-PLUS2
ISA GENERIC
TYPE PLUS2
SLOT1 WEDNESDAY
ANSWER NIL
(the rest of the slots also nil)

Resolving this subgoal of course needs knowledge in the form of other abstractions or productions to find the answer.

The final step is to carry out the action and remove the abstraction:
**ABSTRACTION-DO-ANSWER-ANSWER**

**IF** the goal is a generic goal and fact retrieved is in the retrieve slot of the goal and the action slot equals answer:=answer

**AND** retrieved has answer in the answer slot

**THEN** put answer in the answer slot of the goal, set the abstraction, action and retrieved slots to nil, and put the original abstraction in the test slot

This production rule takes the answer from the retrieved slot and puts it in the answer slot of the goal, and resets the rest of the slots:

**EXAMPLE-FINCHAM-GOAL**

ISA GENERIC
TYPE HOCKEY-DAY-GOAL
SLOT1 WEDNESDAY
ANSWER FRIDAY
ABSTRACTION NIL
RETRIEVE NIL
TEST EXAMPLE-FINCHAM-ABSTRACTION
ACTION NIL

The abstraction that has just been used is retained in the test slot. Although this has a specific purpose that I will discuss in the next section, it also allows access to the abstraction even when the abstraction is proceduralized. The proceduralized version of the example is:

**IF** the goal is of type hockey-day-goal and slot1 equals day1 and the answer is nil

**AND** there is a fact that day1 plus2 equals day2

**THEN** put day2 in the answer slot of the goal and put example-fincham-abstraction in the test slot of the goal

This rule does exactly what we expect it to do, an leaves behind a reference to the example-fincham-abstraction. Even when the proceduralized version of the abstraction is fired, the declarative version is still available for retrieval, provided that the abstraction still has an activation that is high enough for retrieval. If the activation of an abstraction drops below the retrieval threshold, it becomes a meaningless symbol in the production rule and declarative conscious access is lost. Although the symbol has become meaningless, it still has a function, as we will see in the next section.

**Chaining abstractions**

In order to implement the chaining of abstractions, a few more production rules are needed. The following rule implements handling the prev-links:
Appendix: Implementation of abstractions in ACT-R

ABSTRACTION-DO-NEXT
IF the goal is a generic goal of type type, and the abstraction
  slot of the goal is nil and the test slot has value prev-abs
  AND there is an abstraction with goal type and prev prev-abs
THEN put the abstraction in the abstraction slot of the goal

The rule for handling fail links is slightly different, since it has to fire whenever the
current abstraction somehow gets stuck. The rule has to remove the current
abstraction, and replace it by an abstraction that points to it using a fail link:

ABSTRACTION-REPLACE-FAIL
IF the goal is a generic goal of type type and the abstraction
  slot contains some abstraction abs1
  AND there is an abstraction with goal type and fail abs1
THEN put this abstraction in the abstraction slot of the goal

This production may of course only fire if we are really stuck, so we give it a low
expected gain.

Proceduralizing abstractions
To proceduralize an abstraction, the same method is used as outlined in chapter 6.
After an abstraction has been retrieved, but before its contents have been copied to
the goal (so in between retrieve-abstraction and copy-abstraction-to-goal), a push-
dependency rule may fire that pushes a dependency onto the goal-stack. The
remaining steps are exactly the same as outlined in figure 6.2. The resulting
production rules use the test slot of the goal to make sure steps are carried out in the
right order. For example, the rule that results from proceduralizing abs 2 in
figure 7.2 checks in its condition part whether abs 1 is in the test slot, and puts abs 2
in the test slot in the action part.

Building lists and doing rehearsal
Each item in a list is represented by a separate chunk, using the following chunk
type:

EXAMPLE-LIST-ITEM
ISA LIST-ITEM
  LIST-ID Reference to the first item in the list (to itself if it is the first item)
  VALUE The item that is stored in the list
  PREV Reference to the previous item in the list (nil if it is the first item)

In the scheduling goal, slot3 points to the last item of the current list. Tasks that are
put in slot1 of the goal are added to the list by a production rule that creates a new
list-item that replaces the current last item in slot3. As an example, figure 7.14 shows
how the list “ABD” is represented.
Rehearsal is implemented by a subgoal that retrieves the list-items one at a time. If additional processing on items is required, in the case of elaborate rehearsal, a further subgoal is pushed in which the elaboration is carried out. Figure 7.15 shows a schematic representation of both types of rehearsal. In the case of maintenance rehearsal the content of the goal-tp slot in the rehearse-goal is ‘nothing’, and the list-items are retrieved one at a time without further processing. In elaborative rehearsal, the goal-tp slot of the rehearsal goal stores the goal type of the goal that has to do the elaboration (‘count-hours’ in the figure). For each item in the list a subgoal of that type is created and pushed onto the goal stack. Results of this processing are passed on from subgoal to subgoal, and are eventually passed on to the main goal.

Rehearsal is initiated by abstractions that have ‘rehearsal’ in their retrieve slot, and the type of the elaboration subgoal in the test slot (or ‘nothing’ in the case of maintenance rehearsal). The action slot specifies what has to be done with the results.
of the elaboration. The following set of abstractions implements the strategy that calculates the total duration of the tasks already in the list. To do this, the durations of individual tasks in the list have to be retrieved and added. The first abstraction initiates elaborate rehearsal:

```
START-COUNT-REHEARSAL
ISA ABSTRACTION
GOAL SCHEDULE
RETRIEVE REHEARSE
TEST COUNT-STEP
ACTION SLOT2:=ANSWER
```

When this abstraction is retrieved, a rehearsal subgoal (retrieve=rehearsal) is pushed with its goal-tp set to count-step (test count-step). The final result will be stored in slot2 of the goal (action slot2:=answer). Although the rehearse subgoal is implemented by production rules, the processing in the count-step goal still has to be specified:

```
COUNT-GET-TIME
ISA ABSTRACTION
GOAL COUNT-STEP
RETRIEVE TIME
TEST SLOT1=SLOT1
ACTION SLOT3:=SLOT2
PREV NIL

COUNT-ADD-TIME
ISA ABSTRACTION
GOAL COUNT-STEP
RETRIEVE ADDITION
TEST SLOT2=SLOT1*SLOT3=SLOT2
ACTION ANSWER:=ANSWER
PREV COUNT-GET-TIME
```

The rehearse subgoal puts the currently rehearsed item in slot1, and the current results of elaboration in slot2. It expects the result of the elaboration step in the answer slot. So, at the moment the subgoal of type count-step is pushed, slot1 contains a task, and slot2 contains the running total of the duration. The first step is to retrieve the duration of the task that is currently rehearsed. These durations are stored in chunks of type time, which have the task in slot1, and the duration of the task in slot2. Count-get-time retrieves a fact of type time which matches the task in slot1 (test slot1=slot2). It then stores the duration of the task in slot3 (slot3:=slot2). The next step is to add this duration to the running total in slot2. Count-add-time retrieves an addition fact with the first addend (which is in slot1 of the addition fact) equal to the running total and the second addend (in slot2) equal to the duration of the current task (slot2=slot1*slot3=slot2, a conjunction of two tests: slot2=slot1 and slot3=slot2), and stores the sum in the answer slot of the subgoal (answer:=answer). Whenever something is put in the answer slot of a goal it is automatically popped, so the elaboration subgoal is popped after count-add-time has finished.
Learning new abstractions

Suppose we need to make a schedule. We may use knowledge about lists to start with. How do we make a list? First we have to find a beginning. Once we have a beginning, we find a next task until we are done. A general set of abstractions to create a list might look like:

```
FIND-BEGINNING
ISA ABSTRACTION
GOAL LIST
RETRIEVE BEGINNING
TEST SLOT3=NIL
ACTION SLOT1:=SLOT1
PREV NIL
```

This representation assumes that the list is stored in slot3 of the goal (as illustrated in figure 7.14), and that items in slot1 are added to the list and copied to slot2. Find-beginning specifies that if there is no list yet (test slot3=nil), a beginning has to be found, and this beginning has to be stored in slot1 (action slot1:=slot1). Once list-building productions have added the item in slot1 to a new list in slot3, and have transferred this item to slot2, the find-next abstraction specifies that a next relation has to be found between the item in slot2 (test slot2=slot1), the item we have just added to the list, and some new item, which will be stored in slot1 (action slot1:=slot2).

In the ACT-R model, the process of adaptation does not precede the rest of processing, but rather is part of it. A new strategy is initiated by a production rule that pushes an abstraction as a subgoal. This subgoal only produces the first step in the solution plan: later parts of the plan are generated when needed later on. The rule that pushes an abstraction goal is the subgoaling version of the rule that retrieves abstractions:

```
SUBGOAL-ABSTRACTION
IF the goal is a generic goal of type type, and the abstraction
slot of the goal is nil
THEN set as a subgoal an abstraction with goal type
AND put this abstraction in the abstraction slot of the goal
```

This rule has to compete with the retrieve-abstraction rule, but since it has a higher cost, it will only occasionally win the competition if an abstraction is already available in the current situation (or, it will almost never win if there is a task-specific production rule available with a high expected gain). If there is no abstraction available, the retrieve version of the rule will fail, and subgoal-abstraction will be chosen automatically. Once an abstraction has become the goal, the first step is to find a goal that is similar to the desired goal-type in the goal slot of the abstraction. For example, in the case of scheduling, the abstraction subgoal becomes:
Appendix: Implementation of abstractions in ACT-R

EXAMPLE-ABSTRACTION-SUBGOAL
ISA ABSTRACTION
GOAL SCHEDULE
RETRIEVE NIL
TEST NIL
ACTION NIL

etc.

The current model uses an explicit representation to store relations between goal types, for example, it represents that schedule is related to list, and next is related to order. An alternative, but less reliable, option is to use implicit association strengths to find related goals. The following rule implements the explicit version:

FIND-ASSOCIATED-GOAL-TYPE
IF the goal is an abstraction for goal type goal-tp1 and no associated goal-type has been found yet.
AND goal-tp1 is related to goal-tp2
THEN put goal-tp2 in the test slot of the goal

In our example, this rule stores ‘list’ in the test slot of the abstraction. In the next few steps (for reasons of brevity, I will omit the production rules), an abstraction of the associated type is retrieved and its slots are copied to the abstraction, producing (assuming find-beginning is retrieved):

EXAMPLE-ABSTRACTION-SUBGOAL
ISA ABSTRACTION
GOAL SCHEDULE
RETRIEVE BEGINNING
TEST SLOT3=NIL
ACTION SLOT1:=SLOT1
PREV NIL

No more adaptations are possible for this abstraction, since the beginning type in the retrieve slot cannot be related to any task-specific aspect. As a consequence, applying this abstraction will lead to a subgoal of type beginning. Once the next step in the plan, the find-next abstraction, has been adapted to the schedule goal, the retrieve type can be filled with a task-specific term. The situation is as follows:

EXAMPLE-FOLLOW-UP-ABSTRACTION
ISA ABSTRACTION
GOAL SCHEDULE
RETRIEVE NEXT
TEST SLOT2=SLOT1
ACTION SLOT1:=SLOT2
PREV EXAMPLE-ABSTRACTION-SUBGOAL

In this case a fact of type next has to be retrieved. But since next facts are related to order constraints, next can be substituted by order, producing:
7: Models of Scheduling

EXAMPLE-FOLLOW-UP-ABstraction
ISA ABSTRACTION
GOAL SCHEDULE
RETRIEVE ORDER
TEST SLOT2=SLOT1
ACTION SLOT1:=SLOT2
PREV EXAMPLE-ABSTRACTION-SUBGOAL

Although there are probably more ways to adapt abstractions, this are sufficient for the second model.