The Breakdown of Operators when Interacting with the External World

Garrett A. Pelton   Jill Fain Lehman

February 1994

CMU-CS-94-121

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research was supported in part by the Markle Foundation under Grant No. G93112, and in part by Martin Marietta Corporation.

Keywords: Artificial Intelligence, Learning, Plan Formulation, Plan Execution, Program Transformation, Soar

Abstract

By looking at the simple task of tossing a bean bag from hand to hand, we show how the macro operator method breaks down when formulating agent models that interact with an uncertain external world. A macro operator encapsulates a plan to reach an objective. Occasionally the objective will be found to be unachievable, requiring the macro operator and its plan to be rejected. Letting the macro operator interact with the external world does not, by itself, change this situation, but the fact that the results of the interaction are uncertain, and the agent's knowledge incomplete, does. The key idea is that the agent can't positively determine if progress towards the objective is being made in the external world, and thus errors will be made in rejecting a macro operator that would succeed. We show that there are a number of methods by which the agent can recover from such an operator rejection and continue toward the operator's objective. If we make operator rejection and recovery into a common mechanism, then the operators and the plans they represent will be split by the interaction into a sequence of smaller operators each doing a portion of the work toward the objective of the larger operator.

The models are described in terms of Soar and we assume the reader's familiarity with both the architecture [Laird and Rosenbloom, 1987] and the Problem Space Computational Model [Newell *et al.*, 1991] in our discussions.

1 Introduction

In this paper we examine an agent interacting with an uncertain external world and how this interaction constrains a model of the agent. In particular, we look at agent models formulated as problem spaces in the Problem Space Computational Model (PSCM) [Newell *et al.*, 1991] and implemented in Soar [Laird and Rosenbloom, 1987], and we assume the reader is familiar with these systems. In both the PSCM and Soar an *objective* (or goal) is a definition of some state of both the model and the external world, and an operator is what causes movement towards some objective. Each operator has its own objective, that hopefully is a step toward achieving some larger objective. Given a particular operator, the amount of interaction it can do with the external world depends on both the reliability of the external world in producing a response to a requested action, and the speed of the external world in producing the response. This limit on the amount of interaction would seem to limit the size of the operator's objective, what the operator is trying to achieve. However, we will show that instead, an operator's objective can be shared across many operators each making some progress. In particular, given an objective that is too large, we will show methods for automatically *splitting* the processing of the objective into a sequence of smaller operators so that the objective is still achieved.

We start with the fact that actions in the external world take time, and models that request actions of the world have to do something while the external world is producing a desired response. Currently, most models wait for a response or check that progress towards a response is being made. Since a desired response is not always produced, a method of terminating the waiting has to be available. The basis for this termination is a form of exhaustion, and the knowledge generated from it can be overgeneral, applying in other situations when waiting would be more appropriate. However, we will describe a range of methods that can recover from an inappropriate termination with another operator that can reach the original objective.

The combination of termination possibly followed by recovery we call *splitting*. Splitting is a run-time strategy for a model, and is an alternative to large operators that wait for achieving an objective. Waiting still occurs in systems that split the operator. However, the waiting no longer occurs within the monolithic operator, but rather between the smaller split operators. This allows work to proceed on other tasks while waiting, by simply picking operators for those tasks. This type of multi-tasking occurs without the undesirable task composition that occurs when picking the operators for a second task within a large waiting operator. The downside of splitting is the extra work done for recovery. Although a particular Soar or PSCM model might mix split and monolithic operators to achieve some performance goal, we will show that the mechanisms for handling splits due to failure of the external world producing a desired response must exist. Some method of recovery from overgeneral terminations must exist as well.

The application of an operator is considered complete when the operator's objectives have been achieved. Throughout this paper, a *fanatical model* of PSCM and Soar operator application is assumed. This means that once a PSCM or Soar operator is selected, the completion of the operator will not be inhibited by the architecture of either the PSCM or Soar. This is our interpretation of the definition of a PSCM operator application as "an effective procedure for a function" [Newell *et al.*, 1991].

The fanatical completion assumption is why some form of learning operator terminations and recovery from over-general operator terminations must exist in PSCM and Soar models. When the achievement of an objective is *known* to be impossible, then the operator attempting to achieve that objective must be terminated. This termination redefines the objective of the operator so that the operator is considered complete in this new case. This termination mechanism by itself does not violate the fanatical completion assumption. However, if a mistake is made in applying the learned termination knowledge, then it must be possible to achieve the objective even after the operator is terminated or the fanatical completion assumption will be violated. This is why some method of recovery is required.

In the rest of the paper we describe how interacting with the external world acts as a force for splitting PSCM and Soar operators into smaller operators. We start by looking at how Soar's knowledge compilation mechanism can cause operators to split in Section 2. Since knowledge compilation is within the architecture, these splits occur in a model without the Soar agent doing any deliberate processing to invoke a split. Why this occurs and how to recover in these situations is the major point of the discussion. In Section 3 we discuss the deliberate splitting of operators, and the implications deliberate splitting has for recovery. In general, deliberate splitting removes more information than architectural splitting, making some forms of recovery more difficult. The last topic, described in Section 4, involves PSCM models that are doing multiple tasks and include a goal of high utilization of the cognitive resource. Making the PSCM model's use of the cognitive resource more efficient means that, whenever possible, work on one of possibly multiple tasks is being done. If a task's progress depends on some external world response, then rather than wait for that response a different task should be worked on. We discuss how splitting helps achieve this goal, and also discuss other ways to handle multiple tasks and their relation to splitting. Finally, Section 5 provides a discussion of how these topics fit together and into the general PSCM and Soar picture.

2 Architectural splitting of operators in Soar

Soar is an implementation language for PSCM models. However, Soar only approximates some PSCM functionality, in particular PSCM learning. PSCM learning occurs when an impasse occurs in a problem space, the *upper* problem space, and another problem space, the *lower* problem space provides the knowledge that resolves the impasse. Our simple model of PSCM learning is: Lower problem space knowledge is translated into a useful immediately applicable form for the upper problem space. This translation is assumed to be perfect. However, because the definition of PSCM knowledge is too vague, PSCM learning is also not well specified. Thus Soar is left with attempting to implement perfect PSCM learning without any detailed guidance from the PSCM. Soar's learning mechanism, *chunking*, fails to be perfect in certain cases. This section describes the effects of this learning failure on Soar models, and how Soar models can recover from such a failure.

Chunking fails when the working memory information in the upper problem space changes while the lower problem space is resolving the impasse. In particular, chunking assumes that when a lower problem space decision is based on working memory information in the upper problem space, either that upper problem space information stays constant, or the results of the decision are not used to construct a chunk. We will call a situation where a lower problem space has made a decision based on knowledge in the upper problem space and that information has changed *temporally inconsistent*. Temporally inconsistent situations can arise only if the *persistence* of a decision exceeds the original reasons that supported the decision. A chunk, created from a temporally inconsistent situation is called, a *non-contemporaneous chunk* because all the working memory information in the upper problem space used to construct the chunk is not present at the time the chunk was constructed.

Non-contemporaneous chunks are a problem because they may not apply in the desired situations, thus causing an impasse to occur again. A chunk's conditions are formed from the working memory information that led to its creation. By definition, the temporally inconsistent working memory information is not available in the upper problem problem space when the chunk is created.

Thus, the chunk cannot apply to the exact information that created it. It can apply to similar information if it is available. However, non-contemporaneous chunks might never apply, and thus no knowledge transfer occurs. As an example, we later show a non-contemporaneous chunk associated with catching a bean bag where the bean bag has to be in the left hand to be caught by the right hand. This chunk will never apply. Since persistence is required for a temporally inconsistent situation to exist and persistence is an attribute of operator application, most non-contemporaneous chunks occur in learning operator application knowledge. The failure to apply during operator application usually causes some chain of data dependent chunks to also fail. This failure splits the learned operator application knowledge into two components: those that can apply before the non-contemporaneous chunk and those that cannot apply. The non-application of this chain of chunks can cause an impasse to occur again. The method of recovering from such splits is to restore the context of the problem solving so that the operators that originally applied in the impasse will apply again. The details of this process are the main subject of this section.

One important fact to note before we continue is that the situations that produce non-contemporaneous chunks are unavoidable when interacting with a dynamic external world. Persistence is required for temporally inconsistent situations to exist, and Soar operators show such persistence. The picking of an operator in the lower problem space can depend upon external world objects accessed via the upper problem space. Since the external world can change these upper problem space objects at any time, and the operator persists and can be tested after these changes occurred, we can't eliminate all the situations that produce chunks. Also, since the external world is changing the reasons supporting the operator's initial proposal, we can't in general predict when temporally inconsistent situations will occur. We can change the Soar architecture to modify the definition of persistence so temporal inconsistencies do not occur. This is called S-support [Laird and Huffman, 1992] and how it affects the topic of splitting is described in Section 2.4. However, we will argue that S-support by itself doesn't change the picture, it is just a different mechanism for enforcing the current paradigm. It is an eager method that detects temporal inconsistencies forcing the user to consider them when they occur. Non contemporaneous chunks turns out to be a lazy method that forces the user to handle temporal inconsistencies when the agent attempts to use the inapplicable learned knowledge.

2.1 Cyclic toss task

We will discuss these issues through the example task of tossing a bean bag from one hand to the other and back, a cyclic toss. In solving this task, we will use the following three operators: cyclic-toss, toss, and catch. A short description of these operators is shown in Figure 1. The definition of the cyclic-toss operator in Figure 1 doesn't include the knowledge of how to get the bean bag from one hand to the other, it only has the knowledge to note that the intermediate state is achieved. When we describe the execution of the cyclic toss task, we will show the cyclic-toss operator learning the other knowledge required to achieve its objective.

In Figure 1, we use the terms *preconditions* and *objective* to distinguish portions of the conditions of the productions that implement the operator. Preconditions are a common planning term [Fikes and Nilsson, 1971], and we are using it in exactly the same manner as the planning literature. Preconditions are differentiated from the proposal conditions for an operator, because an operator's proposal conditions state that this operator is applicable to the problem at hand, and the preconditions state that this operator can apply. The objective is more closely associated with the desired information usually put on the Soar goal than the actual Soar goal. The name objective was picked to avoid confusion with the Soar use of the word goal. Separating the Soar goal from the objective allows the objective's persistence to differ from the Soar goal's architecturally defined persistence.

Operator cyclic-toss <object>

Proposal -

Problem space is Juggling
<object> is jugglable
The location of <object> is in <hand>
<other-hand> is a hand
<other-hand> is not <hand>.

Application -

If The location of <object> is in <other-hand>
Then The intermediate state is noted on the objective

Objective - Toss-twice

The location of <object> is in <hand>
The intermediate state is noted.

Operator toss <object> from <hand>

Proposal -

Problem space is Tossing
<object> is jugglable
The location of <object> is in <any-hand>
<any-hand> is a hand
<other-hand> is a hand
<other-hand> is not <hand>

Preconditions -

The location of <object> is in <hand>

Application -

If The name of <hand> is right
Then Toss-right <object> to <other-hand>
If The name of <hand> is left
Then Toss-left <object> to <other-hand>

Objective - Throw-to-other-hand

The location of the <object> is not in <hand>.
The status of <object> is moving toward <other-hand>

Operator catch <object> with <hand>

Proposal -

Problem space is Tossing
<object> is jugglable
The status of <object> is moving toward <hand>

Preconditions -

The location of the <object> is not in <hand>.
The location of the <object> is close-to <hand>

Application -

If The name of <hand> is right
Then Catch-right <object>
If The name of <hand> is left
Then Catch-left <object>

Objective - Catch-with-hand

The location of the <object> is in <hand>
The status of <object> is held

Prefer the hand that has the object

Rule -

If Two Toss operators are possible one for each hand
Then prefer the operator for the hand that has the bean bag

Implementation Juggling consists of Tossing

Rule -

If An Impasse occurred for an operator doing a task in the juggling problem space
and all the preconditions for the operator are true
Then Try the Tossing problem space.

Figure 1: Knowledge for cyclic toss task

To achieve this persistence, objectives are recorded on the state and can be shared. One major issue with the recording of objectives on the state is the removal of these objectives. Removal can be done with specialized productions that recognize when an objective has been achieved and then remove the achieved objective. These removal productions are not shown.

We have left some knowledge out of Figure 1 because it is not required for our discussion. The key piece of knowledge left out is the operator subgoaling knowledge of how objectives of one operator become the desired objectives for other operators. We also assume perfect action execution in this section. Thus the Toss action requests reliably toss and the Catch action requests reliably catch. We relax this assumption in Section 3. The other details of the knowledge shown in Figure 1 will be explained when we describe the execution of the cyclic toss task.

2.2 Task Execution

Figure 2 shows the execution of the knowledge in Figure 1 along with some subgoaling and conflict resolution knowledge. We will use it as an example of learning an operator application. The top part of Figure 2 consists of several snapshots of the model's state across time. The bottom part of Figure 2 describes the problem solving that the agent does to achieve the cyclic-toss operator, with each step marked with a letter in a circle. As shown in the cyclic-toss operator in Figure 1,

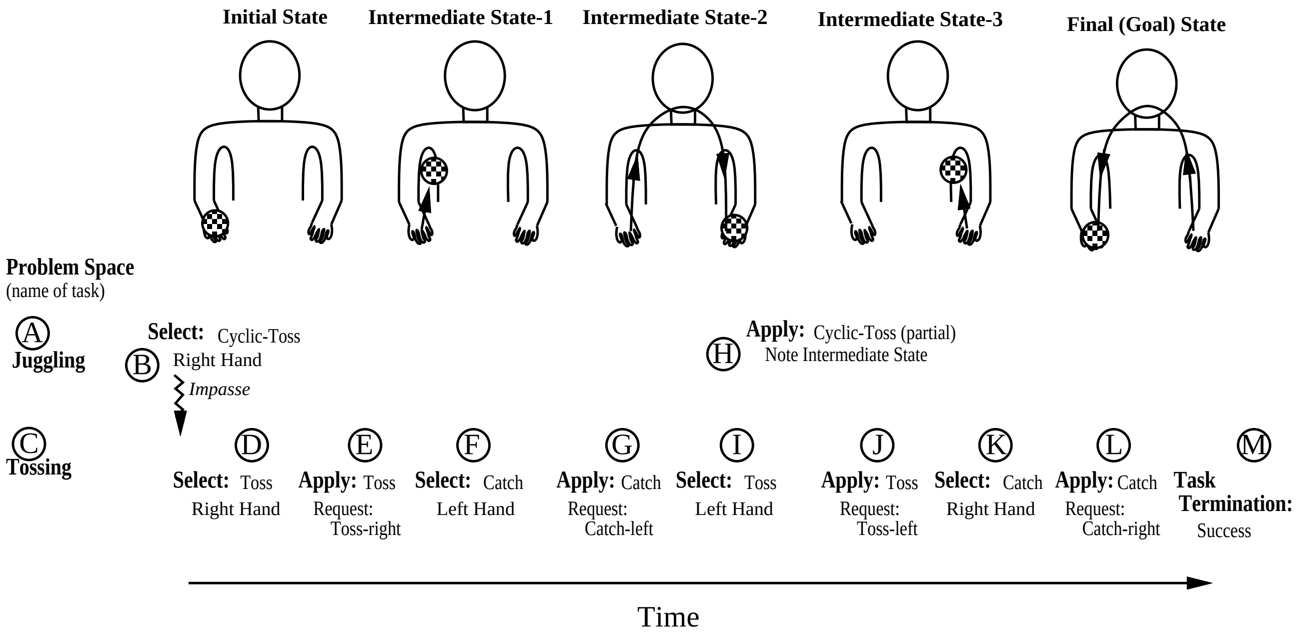

Figure 2: Execution of cyclic toss task

the agent initially knows that an intermediate state (where the bean bag is in the non-initial hand) has to occur and be noted. But the agent doesn't know any more of the details of how to do a cyclic toss. Thus, once the cyclic-toss operator is selected Ⓑ an impasse occurs. The problem space chosen to resolve this impasse is the Tossing problem space because of Figure 1's rule that Juggling is implemented by Tossing Ⓒ. In this problem space, two toss operators are proposed; to toss from the right hand and to toss from the left hand. The toss operator for the right hand is selected Ⓓ, because Figure 1's "prefer" knowledge prefers the hand that has the bean bag. The toss operator applies, requesting the bean bag be tossed to the left hand Ⓔ. Even before the bean bag gets to the left hand the catch operator is selected as appropriate for catching the bean bag in the left hand Ⓕ. Once the bean bag is caught Ⓖ, the original cyclic-toss operator notes the intermediate state Ⓗ, and again we have two toss operators proposed. The toss for the left hand is selected Ⓘ, and applies Ⓙ. Again the toss enables the catch operator to be proposed, and it is selected Ⓚ, and when the bean bag reaches the right hand it applies Ⓛ. The tossing task is now noted as being achieved because all the objectives of the cyclic-toss operator have now been achieved Ⓜ. The knowledge that describes this implementation of the cyclic-toss operator (do toss from right, then toss from left) is added to the Juggling problem space so that in similar applications of the cyclic-toss operator an impasse will not occur.

Implementation Problems

In the bean bag tossing problem of Figure 2 we selected the tosses by using Figure 1's "prefer" knowledge. If that knowledge was removed from the system, then some method would have to be used to pick between the toss (right) and toss (left) operators at the two decision points. Figure 3 shows what happens when the toss (left) operator is picked first, perhaps because means-ends analysis was used and the toss (left) operator resolves the last step in achieving the cyclic-toss.

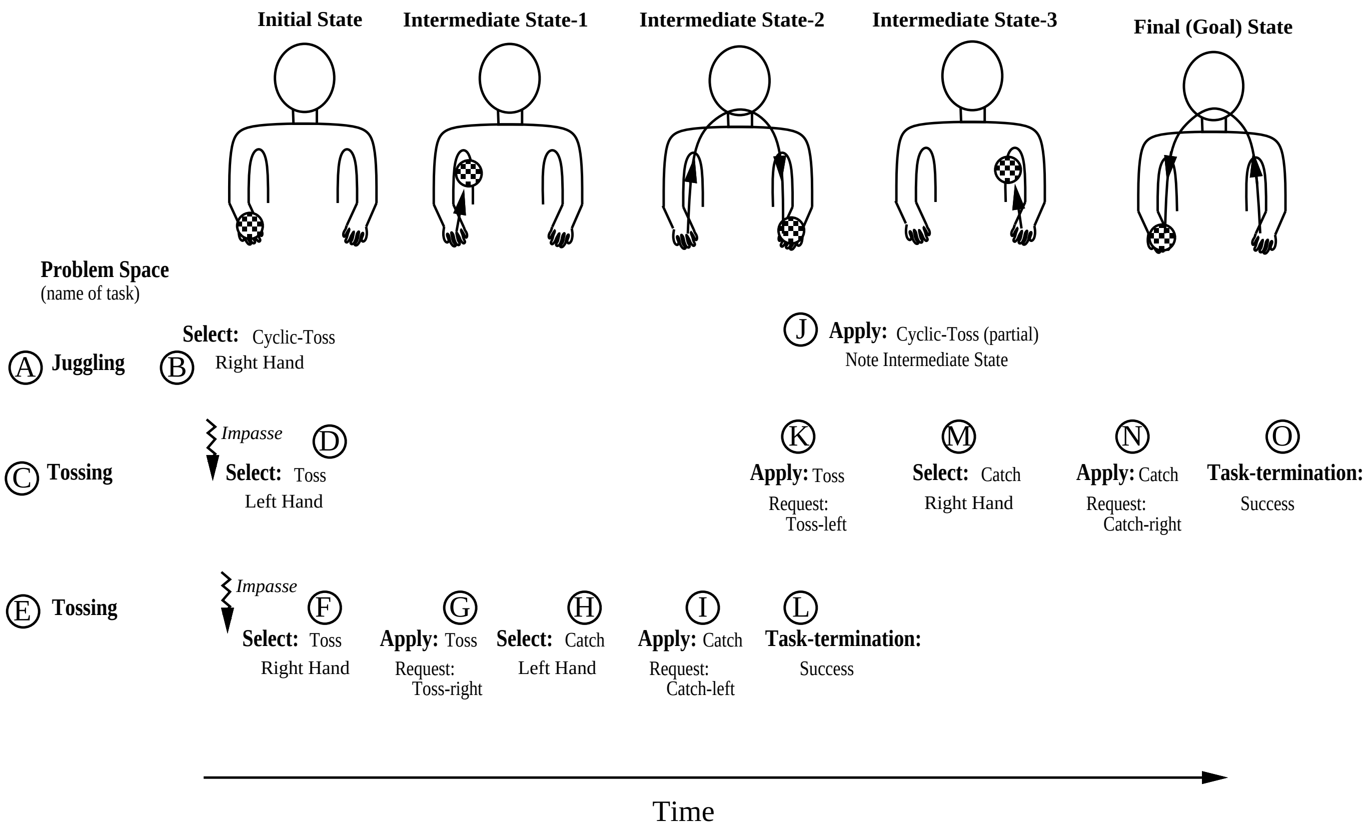

Figure 3: Creation of a non-contemporaneous chunk

The problem solving in Figure 3 follows that of Figure 2, but at Ⓓ the toss (left) operator is picked first. Note that the toss (left) operator has a precondition that the bean bag is in the left hand. This precondition must be true before it can apply. In Figure 3 this precondition causes an impasse and operator subgoaling knowledge selects the Tossing problem space to resolve the impasse with the goal being to get the bean bag to the left hand Ⓔ. The toss (right) operator is selected Ⓕ and applies Ⓖ in this lowest problem space. Since the bean bag is now moving, the catch operator becomes selectable, and is selected Ⓗ. Sometime later, the bean bag is close enough to the hand for the catch operator to apply Ⓘ. The catch operator makes the bean bag be in the left hand resolving the task for the lowest problem space Ⓛ, and allowing both the cyclic-toss operator to note the intermediate state Ⓙ, and the toss (left) operator to apply Ⓚ. As before, the bean bag is caught by the right hand, the Tossing task is resolved Ⓞ and the cyclic-toss is learned.

To an external observer, the scenarios in Figure 2 and Figure 3 are the same, and the knowledge that resolves the cyclic-toss operator is simply to first toss from the right hand and then toss from the left. Unfortunately, the chunk created by Soar for the cyclic-toss Soar operator from the application of the catch (left) Soar operator in Figure 3 is inapplicable. The creation of the toss (left) operator depends upon the bean bag being held in a hand, see Figure 1. If the bean bag was not being held, then perhaps, a different operator would be proposed to achieve the state of holding the bean bag. Testing that the bean bag is being held makes the toss (left) operator in Figure 3 dependent upon the bean bag being in some hand. When the bean bag is tossed from the right hand, this dependency still exists, but it is no longer true. We have a temporally inconsistent situation with regards to the toss (left) operator. So when the application of the catch (left) operator occurs, another chunk is built that incorporates the catch (left) action request and the dependency that

If The problem space is Juggling
the operator is cyclic-toss
the <object> is jugglable
the <object> is moving toward the <hand>
the location of the <object> is not in <hand>
the location of the <object> is in <any-hand>
<any-hand> is a hand
the location of the <object> is close-to <hand>
the name of the <hand> is left
Then catch-left <object>

Figure 4: Example of non-contemporaneous knowledge

If The problem space is Juggling
the operator is cyclic-toss
the <object> is jugglable
the <object> is moving toward the <hand>
the location of the <object> is not in <hand>
the location of the <object> is close-to <hand>
the name of the <hand> is left
Then catch-left <object>

Figure 5: Example of useful knowledge created on second attempt.

the bean bag is in some hand. Thus the catch (left) is dependent both upon the bean bag initially being held by a hand, from the toss (left) operator creation, and being moving and close to the left hand at the time of the catch from the catch's preconditions. However, the temporal aspect of this scenario is lost. The chunk generated for the catch, as shown in Figure 4, requires that the bean bag be both moving and in a different hand than the left (catching) hand. This is hardly a good situation for catching a tossed bean bag.

The chunk in Figure 4 is an instance of a *non-contemporaneous chunk* (see [Laird and Huffman, 1992]). It is splitting the cyclic-toss operator into the two portions. The first portion executes before the chunk doesn't apply, making the first toss of the bean bag. The second can't apply because the bean bag is not caught.

The main issue here is that the cyclic-toss operator should not care whether the method shown in Figure 2 or in Figure 3 was used for solving the cyclic toss problem. The chunk shown in Figure 4 will not apply at the appropriate time because the conditions will not match any juggling situation. Thus the knowledge transfer for the cyclic-toss operator failed to generate effective knowledge. This failure to apply causes the cyclic-toss operator in one problem solving scenario, Figure 3, to not be as effective as in the other, Figure 2, even though no a priori reason exists to prefer one over the other.

Recovery from split

The situation in Figure 3 is not as bad as it might seem because the proper knowledge is built when the cyclic toss is next attempted. This happens because the cyclic-toss operator was implemented with simpler Soar operators that worked entirely based on the situation in the top problem space. When the cyclic-toss operator is selected some time in the future, and the bean bag is in the right hand, the first chunk corresponding to throwing the bean bag from the right hand

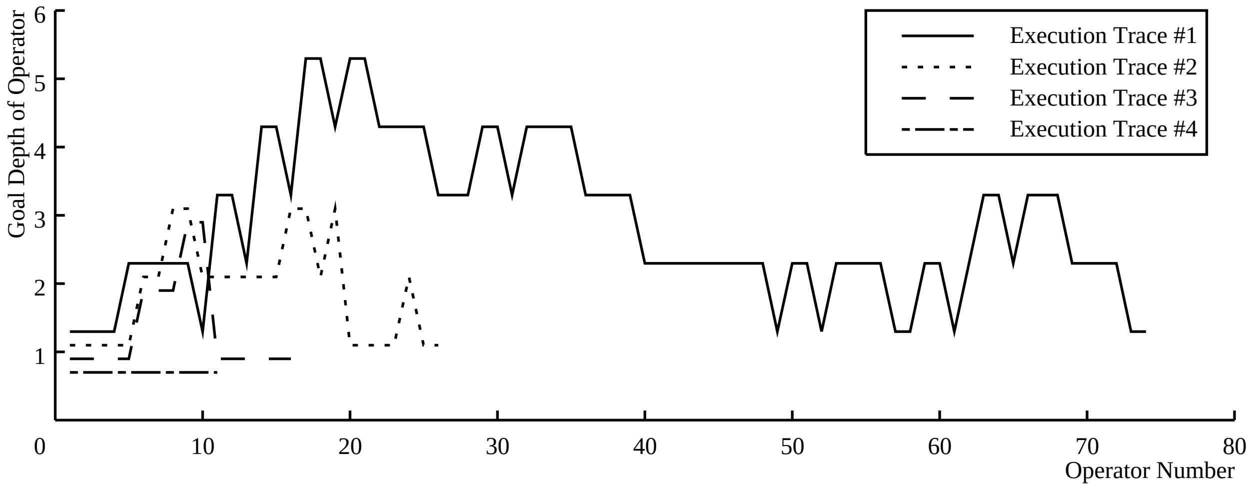

Figure 6: Graph of operator goal-depth by learning trial

will apply, sending the bean bag to the left hand. The second chunk for catching the bean bag in the left hand can't apply because it requires the bean bag to be in the right hand and it isn't. Thus, an impasse will occur because the cyclic-toss hasn't completed. In resolving this impasse, the Tossing problem space is picked again. Since the current situation matches the proposal conditions for the catch operator, it will be proposed and applied, catching the bean bag. This time, however, the catching request creates the useful chunk, shown in Figure 5, for the Soar cyclic-toss operator. Thus, in this case Soar learns the appropriate knowledge over multiple tries rather than simply from the initial learning situation.

The strategy used in Figure 2 and Figure 3 for creating Soar macro operators such as cyclic-toss is very powerful. If the simple Soar operators that define the Soar macro operators either always work in the Soar state that the Soar macro operator is working in, or can always recreate any local state information, then the program can always recover from non-contemporaneous chunks. This is important because it ensures an effective implementation of the PSCM operator, even if the problem solving in Soar is more complex than in the PSCM model.

Figure 6 shows the iterative generation of increasingly general chunks in a more complicated task of a robot pushing a box (described further in Section 4). The X axis of Figure 6 counts the operators required to do the task in each trial. The Y axis is the depth of the goal stack that existed when the operator was proposed. Four consecutive learning trials on the same problem are shown. The problem solving in the first trial creates some non-contemporaneous chunks. These chunks do not fire in the second trial, causing impasses to resolve the missing knowledge. The new chunks from the second trial are useful up to the point that they, too, become non-contemporaneous. This process repeats itself until we have useful immediate knowledge available at the end of trial 3 and do not need to impasse in trial 4.

Failing to recover

When a non-contemporaneous chunk fails to apply, any other knowledge dependent on its actions will also fail to apply. Recovering is not necessarily as easy as described above. The difficult situations occur in operator application, when a portion of the operator applies, thus making some

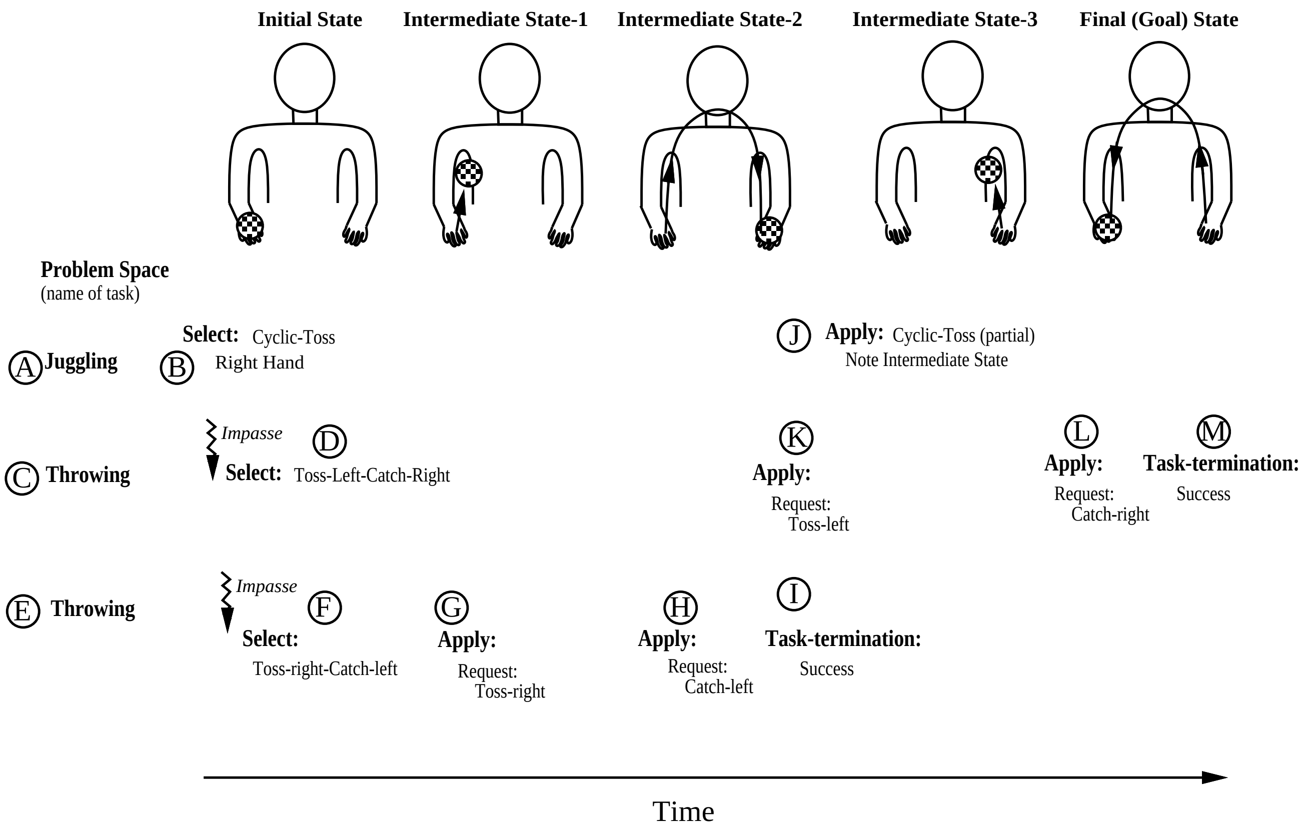

Figure 7: Cyclic toss with symmetric toss-catch operators

previous portion of the operator inapplicable. If the now inapplicable portion of the operator is required for recovery then a problem exists. We can create such a situation by modifying our example, so that instead of having toss and catch operators in the Tossing problem space, we try to implement the cyclic-toss operator with the symmetric toss-left-catch-right and toss-right-catch-left operators in the Throwing problem space. We can assume that the Throwing problem space operators were learned from the Tossing problem space operators some time in the past. Thus we learned how to throw from hand to hand before attempting the cyclic toss. Unfortunately, recovery with knowledge of this form only becomes possible if we expand our current methods,

We will explain why recovery is difficult by going through our same cyclic toss example. We are assuming that the proposal conditions of these complex toss and catch operators are the same as those for the toss operators of Figure 1, and that the catch action occurs when the catch preconditions are true. Figure 7 shows the cyclic toss being done with these operators assuming the left hand is tried first. When the left hand is tried D, the same impasse from Figure 3 of an unresolved precondition occurs and leads to the toss-right-catch-left operator being picked F in a new Throwing problem space. This operator applies tossing the bean bag to the left hand G and catching it in the left hand H. Once caught, the bottom throwing task is terminated I, the intermediate state is noted J, and the toss-left-catch-right operator can now start applying. It requests the toss-left action K and then the catch-right action L. Again when the bean bag is back in the right hand, the task terminates M. The learning is very similar to that in Figure 3, and a non-contemporaneous chunk is created for the cyclic-toss operator for the catch-left action. This chunk looks exactly like our previous chunk in Figure 4.

On the next attempt at a cyclic toss that starts from the right hand, the first chunk for the cyclic-toss operator applies; tossing the bean bag from the right hand. Since the second chunk can't apply, an impasse occurs. The Throwing problem space is picked, but no operators are proposed. Both toss-left-catch-right and toss-right-catch-left operators are only proposed if the bean bag is in a hand. Since the bean bag is in the air, no operators are proposed. This results in a new impasse, but we have no knowledge relating to the handling of this impasse, so the bean bag falls to the floor.

Not being able to independently catch the flying bean bag in the Throwing problem space is the root problem in this example. The knowledge about catching in the left hand was available in the Throwing problem space, it was just embedded in the wrong operator. Even if the Tossing problem space with our original toss and catch operators was available to resolve the impasse, we could not have recovered, because the issue to be resolved by this impasse is that no operator is available in the throwing problem space. To recover, somehow we have to extract the catching knowledge and make it available in the Throwing problem space in the form of a new operator.

2.3 Generalizing Recovery Methods

This section describes how to write Soar models that exhibit the recovery capability we saw in the previous section, while avoiding the problems. In particular, it is concerned with non-contemporaneous chunks that arise out of interaction with the external world. Non-contemporaneous chunks can also be created by strictly internal problem solving, and the same methods help in these situations, especially if the problem stems from planning and using knowledge about the external world responses.

Recovery consists of two parts: *reconstructing the context* of the problem solving up to the point it was disrupted (i.e. when the non-contemporaneous chunk didn't fire) and *continuing the problem solving* after the disruption. When the learned application of the cyclic-toss operator was disrupted by the non-application of the non-contemporaneous chunk in Section 2.2's positive example, the problem solving continued using the original problem solving knowledge. In general, non-contemporaneous chunks disrupt the immediate use of operator application knowledge. The disruption causes an impasse that can be used to learn knowledge that can replace the non-contemporaneous chunk, if the correct context can be created. In Section 2.2's negative example, the context could not be created that would allow the catch portion of the toss-right-catch-left operator to apply. The context reconstruction part of recovery has a continuum of solutions that range from doing nothing, because all the state between both problem spaces is shared, to completely reconstructing the context in the lower problem space. In the first case the problem solving in the lower problem space duplicates all its results in the upper problem space. Thus, when the non-contemporaneous chunk refuses to fire we have precisely the context needed to proceed. In the second case we keep no intermediate results but instead have a method for reconstructing the problem solving whenever it is needed. We now describe two example systems that can be classified as different points along this continuum.

Sharing state

The cyclic-toss operator of Figure 3 is an example of sharing the major problem solving results of the problem solving in a lower problem space with the upper problem space. A shared portion of the state holds the problem solving context. In Figure 3 the shared problem solving context is the location and status of the bean bag, and the *objective* of the cyclic-toss operator. This example only requires the perceptually determined location and status of the bean bag. When the impasse occurs in the cyclic-toss operator the second time, the bean bag's status of "moving" is what enables the catch operator's selection and thus the learning of the correct knowledge. In Figure 7 the problem solving context for each of the catch actions includes the fact that the associated toss was done. Thus in this case, all the context is not shared, the fact that the associated toss was done is "known" only in the sub-context. Thus, when the impasse occurs after the toss, the system can't continue because it can't recreate this specific context for the catch action.

In Figure 3 the objective does not need to be shared, to solve this particular problem. The sharing is done for demonstrative purposes, since sharing of the objective is required in more complex problems. In some more complicated systems objectives are linked to implement a stack mechanism to focus the effort.

Replanning

At the other end of the spectrum, the reconstruction process can re-derive the context information as needed. Consider Mitchell's robot, that has the multiple goals of finding a cup and keeping itself charged [Mitchell, 1990]. The robot determines an action by planning and then requests the action forgetting all the planning knowledge involved in picking that particular action. If Mitchell's robot didn't learn, it would completely replan for each action request. Mitchell makes this replanning efficient by having the robot cache the planning results as stimulus-response rules. The conditions of the rules are generated through an explanation-based generalization of the original planning. Thus, the next time a similar situation arises, the cached rule will fire providing the action request and make replanning unnecessary. What Mitchell does not do is cache intermediate planning results, only the final ones.

Mitchell's robot is an example of rederiving the context information when it is needed. Figure 8 shows one possible PSCM version of Mitchell's method learning the cyclic toss. The execution trace on the left shows the planning that initially occurs and the execution trace on the right the application of the learned rules. In Figure 8 the operators are shown in bold. The action requests and any state changes appear in normal roman font, and indentation indicates processing within the operator. The text in sans serif font, is the planning activity. Changes made to the state in the sans serif section are removed when the action request is actually made. If an operator occurs within another operator, the sub-operator must be in the context of a sub-problem space.

Comparing the left hand side of Figure 8 to the cyclic-toss example in Figure 2, Mitchell's robot does more planning to determine that the toss (right) action request will lead toward the objective. When the toss (right) action request is made a rule is created that will request a similar action in a similar situation. All the rules created in Figure 8 are very similar to the chunks created in Figure 2, except that the intermediate state information is included in all the chunks. The intermediate state information is included because a complete plan is used to determine the actions and in the complete plan the intermediate state value is important. Proceeding down the left hand side of Figure 8, when the bean bag leaves the right hand moving toward the left hand, the process of determining the next action starts anew from the cyclic toss goal and the perceptual knowledge that the bean bag is moving toward the left hand. The process rebuilds whatever context is needed to determine that the catch (left) action is the next action that leads toward the objective and thus should now be requested. A new rule for this plan is now created. This process continues until the cyclic toss objective is reached. The right hand side of Figure 8 shows the application of the learned rules, one for each cyclic-toss operator. The cyclic-toss operator is serving as the goal in Mitchell's system.

Initial Problem solving Trace
(before learning)

cyclic-toss
    toss (right)
        toss-right action request
    catch (left)
        catch-left action request
    Note Intermediate State
    toss (left)
        toss-left action request
    catch (right)
        catch-right action internal request
  toss-right action request
cyclic-toss
    catch (left)
        catch-left action request
    Note Intermediate State
    toss (left)
        toss-left action request
    catch (right)
        catch-right action internal request
  catch-left action request
cyclic-toss
    Note Intermediate State
    toss (left)
        toss-left action request
    catch (right)
        catch-right action internal request
  Note Intermediate State
  toss-left action request
cyclic-toss
    catch (right)
        catch-right action internal request
  catch-right action request

$\Longrightarrow$

Final Problem Solving Trace
(after learning)

cyclic-toss
  toss-right action request
cyclic-toss
  catch-left action request
cyclic-toss
  Note Intermediate State
  toss-left action request
cyclic-toss
  catch-right action request

Figure 8: Mitchell's system - Always re-planning

The PSCM implementation shown in Figure 8 differs from Mitchell's work in two ways. First Mitchell's robot doesn't have operators, meaning that the stimulus-response rules apply immediately without any decision process occurring. The only issue this raises is that some reasoning may be occurring in Figure 8 (i.e. in Soar) between the creation of the operators, and the selection of the operators. Mitchell's system encapsulates this reasoning in the left-hand side of the stimulus-response rule. Such an encapsulation assumes that no new goals or knowledge will change the reasoning process. The second difference between Figure 8 and Mitchell's work is that action requests are idempotent, that is they can be requested multiple times and have the same effect as if they were requested only once. Our action requests recognize the action has already been requested and uses the recognition to withhold the same action being requested.

Mitchell avoids non-contemporaneous stimulus-response rules by working from a snap-shot of the world, having the only result be the action request, and always re-planning. Since he works from a snapshot, the external world cannot change while he is doing the planning, eliminating one source of non-contemporaneous stimulus-response rules. Since he has no persistent objects in his planning sub-goals (he always re-derives the objects from the new snap-shot), he has eliminated the other source of non-contemporaneous stimulus-response rules. A Soar model can always do the re-planning, but, as we will show in Section 4.2, this has implications for the number of Soar operators required to achieve an objective. A Soar model can also specify that all planning is done in a snap-shot of the external world, though it is not usually done.

Making recovery work

Once the problem solving context is restored, then the original knowledge can automatically apply and the problem solving continues. Creating a system so that this process can occur easily is more an issue of context reconstruction than problem solving continuing. We have shown two ways to reconstruct the context, both of them benefiting from the use of small operators at the base of the operator recursion. The smallest operators are those that do one change to the state, and then terminate. Building productions from the small operators means that when an impasse occurs in the application of a complex operator, other operators can apply. Unfortunately, this only works if the small operators are available in the problem space that resolves the impasse. This section describes a method to make the small operators, and thus their knowledge, available again, through the creation of a new operator. The method is not automatic today, but it shows how a Soar model can be changed to work around this problem.

The lack of an operator arises because the impasse can occur in the middle of the complex operator's implementation. Also, the impassed situation might not match any of the sub problem space's operator's proposal conditions. We saw this in Figure 7 after the initial learning. The cyclic-toss operator impassed after the first toss because the chunk to do the catch didn't apply. The Throwing problem space was selected to resolve the impasse, but the proposal conditions of the two operators in the Throwing problem space didn't match the state of the bean bag being in the air.

One way to ensure that the small operators are available is to do all the work within a single problem space. However, making all the operators available in one huge problem space precludes problem spaces' organizational advantages. Also, the existence of multiple problem spaces is not the issue, we assume the complex operator's implementation was learned by a previous traversal of the problem space structure. The creation of a new operator will force the problem space structure to be traversed again, relearning the particular knowledge that is appropriate to this new situation. This new operator will also encapsulate this new knowledge.

Creating another operator could be done automatically in the impasse that occurs because no operator is available, but care has to be made so that the proposal conditions of the new operator are reasonable. In this situation, we don't want to learn to catch just any object, only those objects thrown from the other hand. How to automatically create the operator is beyond the scope of this paper. The robot work described here shares a common problem space and thus does not have this particular problem. The work that has been done in this area has created overly specific chunks that were linked to the particular objectives of the cyclic-toss operator.

The reason for creating a new operator is that the model made a commitment to a particular set of knowledge in a particular set of operators, and that structuring has turned out to be wrong. It is recognizing that the current organization of the knowledge into operators is causing the problem. The inapplicability of knowledge that exists within a different operator just means that a new operator has to be created that can also encompass that knowledge, possibly duplicating it.

2.4 Eager vs. Lazy Detection

So far we have been describing how non-contemporaneous chunks can cause operators that do multiple actions to split. In essence the problem is that the learning method in Soar does not correctly handle non-contemporaneous information, and the fact that a temporally inconsistent situation did exist is detected only when the chunk fails to apply. An alternative approach is to change the architecture to detect when objects become non-contemporaneous and take the appropriate actions so that a non-contemporaneous chunk is never created. Such a change to Soar has been suggested in a mechanism called S-Support [Laird and Huffman, 1992]. The main difference between a Soar system without S-support and one with S-support is when the fact that non-contemporaneous information was used is discovered. A Soar system without S-support detects this situation when an attempt is made to use knowledge built on non-contemporaneous information, because the chunk fails to apply. We call this *lazy* detection, because it is happening as late as possible. The Soar system with S-support detects and removes the non-contemporaneous information even before it can be used. This is *eager* detection, because it is as early as possible. The purpose of this section is to show that early detection also splits the operator, requiring some form of recovery as in the previous section.

S-support also splits

First we note that learning in Soar requires an impasse and, for any impasse, we have a super-context (the impassed one) and a sub-context. The creation of a non-contemporaneous chunk depends upon the existence of at least one persistent sub-context object generated from a super-context object that has changed since the generation. In the example shown in Figure 3, the persistent sub-context object is the toss (left) operator that was created when the bean bag was in the right hand Ⓓ. The bean bag's location is the changed super-context object. S-support is a proposed change to the Soar architecture that changes the persistence rules in Soar to remove all sub-context objects that were generated from changed super-context objects. Removal of the non-contemporaneous objects removes one of the conditions (persistence) that allow non-contemporaneous chunks to be created. Thus, the Soar program cannot create a non-contemporaneous chunk. However, further problem solving might be dependent on the object that S-support removed. For this problem solving to continue a new similar object with S-support has to be created. The process that creates the new object turns out to be the same process that the recovery method uses to re-construct the context after a non-contemporaneous chunk.

As an example, in Figure 3 the original toss (left) operator was dependent upon the bean bag being in a hand (as seen on the left in Figure 9). The right side of Figure 9 shows that when the bean bag appeared in the left hand this toss (left) operator lost one member of its super-context support set, and thus would be terminated by S-support, indicated by the grey "X" in Figure 9. Something that was not seen in our simple example was that when the bean bag was in the air a catch (left) operator was proposed and this catch (left) operator is dependent upon the bean bag being in the air. However in our original system, since we had what appeared to be a perfectly good operator already selected this new proposal was ignored. With S-support terminating the original toss (left) operator, the new catch (left) operator can be selected. Thus, the example in Figure 3 with S-support would learn all the right chunks in the first pass, rather than the two passes used with non-contemporaneous chunks. S-support assumes that problems will occur when a temporally inconsistent situation arises before any chunks are built. Note that the method of recovery is the same here as when the non-application of a non-contemporaneous chunk showed a past temporal inconsistency. In the example in Figure 3 both cases have to select and apply a catch

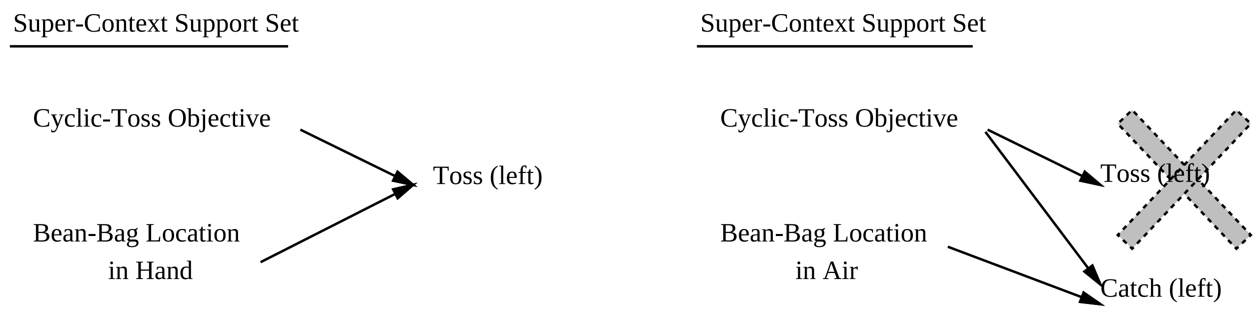

Figure 9: Changing the super-context support

(left) operator. Using non-contemporaneous chunks is lazy because the new catch (left) operator is applied at the latest possible time; S-support is eager because it applies it as soon as possible.

The interesting concept here is that recovering from an S-support retraction of support is exactly the same as recovering from a non-contemporaneous chunk. Thus, S-support should not affect programs that can recover from non-contemporaneous chunks, and programs that can't recover from the loss of S-support to objects cannot recover from non-contemporaneous chunks. This suggests that S-support does not change the set of effective Soar programs, but is, instead, an implementation mechanism that speeds up learning by forcing Soar programs to deal with the loss of super-context support immediately. The situation is analogous to the type system in modern programming languages like ML. The ML language states that all programs will be type correct. Implementing compile-time type checking in ML is not a change to the set of valid ML programs, it is just an implementation mechanism that forces one to deal with type problems at the compile stage.

2.5 Relationship of the methods

We have described two methods (s-support and non-contemporaneous chunks) for detecting a temporally inconsistent situation, and two methods (replanning and sharing state) for recovering from the effects of such a situation. The reasons for picking a detection and recovery method are independent, and thus can be mixed depending upon the model's requirements. To summarize:

Detection - Since this is architecturally defined we have an either/or situation.

- S-support -
  - Early detection of temporally inconsistent situations.
  - Faster learning.
  - Unnecessary work done by architecture removing items that will not be used.
- Non-contemporaneous Chunks -
  - Late detection of temporally inconsistent situations.
  - Slower learning. Often multiple presentations of same problem.
  - Necessary matching occurring on impossible to use productions.

Recovery - Here we have a spectrum of opportunities, where the model's tradeoffs will determine the mix of replanning vs. state sharing. This can change for different parts of the model.

- Replanning
  - Assumes that all the planning decisions should be remade for any new situations. Thus the stability of the external world is low.
  - Results in shorter elaboration chains as a reasoning chain is encoded in a chunk.
- Sharing State
  - Assumes problems will occur.
  - Constantly duplicating results, perhaps unnecessarily.
  - Longer elaborations, each intermediate result is encoded in a chunk.

Furthermore, the recovery methods assume that the organization of the application knowledge into the correct set of operators can be a problem. The reorganization of that knowledge means duplicating it in some cases, and requires the ability to either create new operators for a problem space, or to use the same operator in different problem spaces.

3 Deliberate splitting: Learning PSCM operator terminations

In our bean bag tossing example in Figure 2, we purposefully ignored the fact that the toss from hand to hand takes time in the external world. In general the external world takes time to produce a response when an action is requested. When the external world's response time is included in an operator we call it the *slack time* of the operator. An operator has slack time when the operator requires a specific result from the external world to continue applying. We explore in this section how the external world's response time and the uncertainty of the duration of the response time affects PSCM problem solving.

In addition to taking time, the external world may not respond as expected. If a PSCM operator needs a specific result from the external world to continue, then the lack of this result could keep this PSCM operator from completing the "effective" function the operator is supposed to compute. Normally, an effective operator terminates when it has reached its desired result. To deal with the uncertainty of the external world producing the desired result, we describe the need for an additional class of PSCM operator knowledge, which we call *operator termination*, that defines when the operator has completed. Operator termination knowledge is a dynamic redefinition of the PSCM operator's completion. Sometimes this termination knowledge will be overly general, applying at inappropriate times.

When overly general operator termination knowledge applies, it removes the context the operator provides even though the objective of the operator might still be achieved. Indeed, the fanatical completion assumption would have us achieve this objective. To achieve the objective, we will describe a process similar to the recovery from non-contemporaneous chunks in Soar described in Section 2.3. The recovery process again consists of context restoration and continuation with previous operators. An operator provides both a context for the internal changes to the state and a context for comprehending external changes to the state. This is the context that is lost when the operator termination knowledge applies. In this section we are interested in the context for comprehending external changes, because the lack of a particular external result is what generates the operator termination and recognizing the appearance of the external result is the key for when to restore the context.

When an operator has been terminated before reaching its objective because the external world did not provide a result, we say it has been split by that termination. All the processing leading up to the termination is the before-split portion. The restoration of the context upon recognizing the result, and the continuing of work on the objective of the operator, is the after-split portion. This is very similar to the architectural splits of Section 2. We will show that in Soar, splits of this type happen only when the external world's results aren't immediately produced, because quiescence is required for Soar operator termination knowledge to be utilized. The definition of when Soar utilizes operator termination knowledge gives us a more dynamic Soar operator that can adjust to the reaction time of the external world, splitting only when required by the current external world interaction. Yet this Soar operator can still achieve the original PSCM objective.

3.1 Actions take time

Figure 10 shows the problem-solving involved in learning the first part of the cyclic-toss operator from Figure 2 with the transit time of the bean bag included. In Figure 10, the agent's first action request is to toss the bean bag from the right hand Ⓐ to the left hand. The second operator selected is to catch the bean bag with the left hand Ⓑ, but this operator must wait for the bean bag to get close to the left hand to apply. As before, though it is not shown in Figure 10, only when the bean bag is recognized as back in the right hand does the cyclic-toss operator terminate. In this example, the movement of the bean bag from the right hand to the left takes time. This time is considered to start when the agent makes the action request Ⓐ to throw the bean bag, and end when the bean bag is in the left hand Ⓔ. Since the movement of the bean bag takes time, the agent has to do something while the bean bag gets to the left hand so that the catch (left) operator can apply. The simplest action for the agent to do is to wait. This is shown at the bottom of Figure 10 as the execution of the check-progress operator in the Wait problem space. This new problem space is created in response to the lack of immediately available knowledge to do anything else in either the juggling or toss problem spaces once the bean bag has been tossed. The basis for formulating waiting as the task of this problem space is that an action request has been made (toss (right)). The check-progress operator in the Wait problem space is checking that progress is being made toward the goal of the bean bag getting to the left hand. It is the presence of the action request and the recognition of progress that defines this impasse as slack time.

The example in Figure 10 shows that resolving a lack of knowledge can lead to external world actions, the results of which are interpreted as leading toward the goal state. These types of external world requests usually happen when resolving two particular types of lack of knowledge; an inability to apply an operator (exactly the situation in Figure 3 where the precondition of the toss (left) operator is not met), and not knowing what actions the operator should request to reach the objective (the situation in Figures 10 and 3 for the cyclic-toss operators). Both have to do with deciding how to apply the selected operator to the state. PSCM operators built in this way do not separate action requests from the comprehension of the results as both action request and result comprehension occur within the learned PSCM operator. Figure 11 shows the complete cyclic-toss operator with the top left part of Figure 11 corresponding to the description in Figure 10, and the right side corresponding to the final learned operator. As in Figure 8, the operators in Figure 11 are shown in bold with the actions done by the operator in a normal font. However, the waiting in a sub-problem space in Figure 11 is shown in italics and this will be the convention in this type of figure from this point on. In Figure 11 both toss and both catch operators are sub-operators of the cyclic-toss operator. The waits are at a different indentation level indicating a sub-space where progress could be checked. In the first case the agent is waiting for the precondition of the catch (left) operator to be met. Figure 11 shows that the cyclic toss operator will end up with two slack times that correspond directly to the time it takes the bean bag to actually travel from the right hand to the left and back again. It is the actual presence of the bean bag close to the left-hand that allows the action of catching the bean bag in the left hand to be requested. This means the

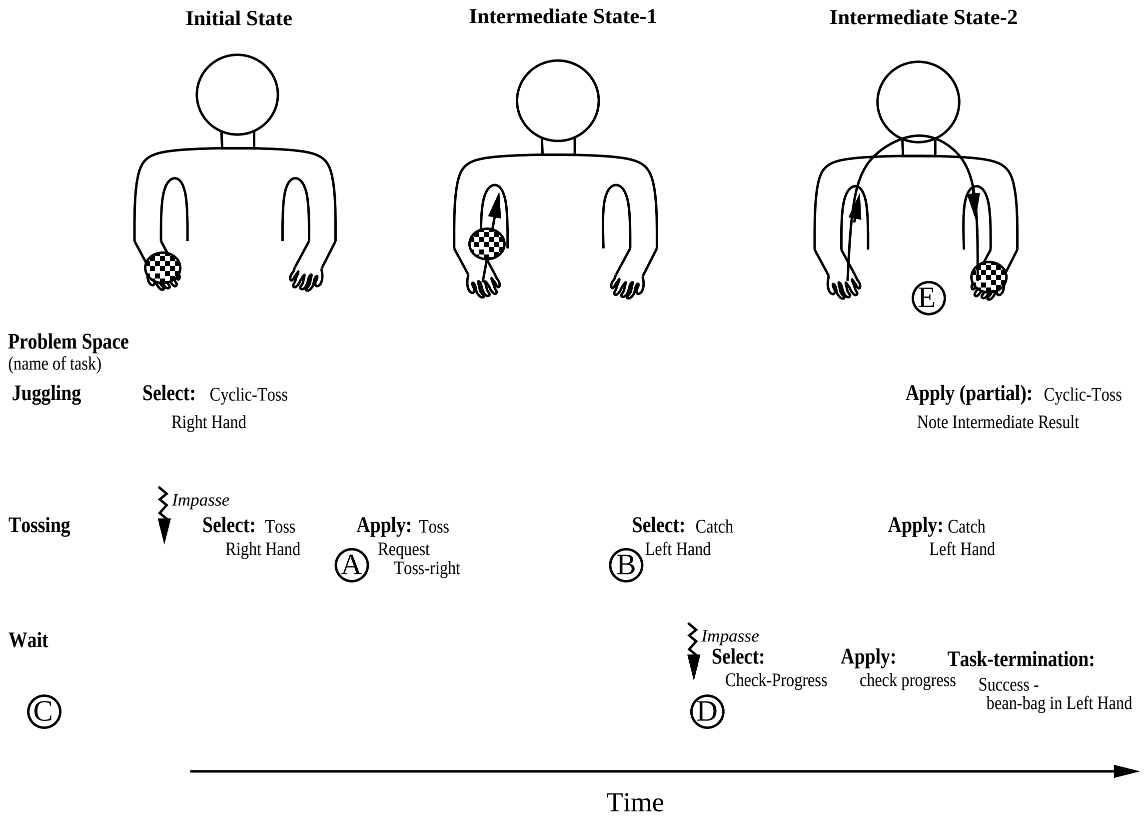

Figure 10: Learning the throwing of a bean bag when toss takes time.

Initial Problem solving Trace (before learning)		*Final Problem Solving Trace (after learning)*
cyclic-toss		**cyclic-toss**
toss (right)		toss-right action request
toss-right action request	$\Rightarrow$	*Wait for Toss Completion*
catch (left)		catch-left action request
Wait for Toss Completion		Note intermediate state
catch-left action request		toss-left action request
Note intermediate state		*Wait for Toss Completion*
toss (left)		catch-right action request
toss from left action request		
catch (right)		
Wait for Toss Completion		
catch-right action request		

Figure 11: Learning when waiting for responses

operator application is suspended while waiting for the requested action of tossing to the left hand to be completed in the external world.

In our example, we simply checked for progress towards the desired result during the slack time. Section 4 describes why you might want to do other activities during this time, and gives the methods for doing so. In the next two parts of this section we discuss what happens if either the agent is able to determine that the result will never be achieved or the rate of progress towards the result in the external world is too slow. We call both of these situations *operator frustration*. The knowledge learned from operator frustration act to redefine the Soar operator's objective allowing the Soar operator to be considered completed prior to actually accomplishing the operator's PSCM objective. This redefinition is operator termination knowledge. Other knowledge might also be learned when learning termination knowledge that would restrict the use of the operator in similar situations. We do not consider that type of knowledge here.

We show in the following sections that checking for progress is a difficult and knowledge rich process, given the possibility of an increasingly hard to detect difference between slow progress and no progress toward some result on the way to the objective. However with the fanatical completion assumption, the PSCM has a qualitative difference between slow progress and no progress. No progress indicates the objective can be discarded, and slow progress indicates the agent should wait for the result because the objective can be achieved. With a diminishing difference between slow progress and no progress, mistakes will be made. Recovering from these mistakes uses techniques similar to those in Section 2.3. These techniques can also be used as the standard methods to handle situations where how to measure progress is not known.

3.2 Operator frustration over unachievable goals

PSCM models can refine their knowledge by interacting with the external world. These experiences may indicate that some aspect of the original objectives is currently unachievable and that this aspect, and maybe the entire objective, should be discarded. Discarding an objective is not a violation of our fanatical assumption, because it is not an artifact of the PSCM or SLCM architecture. Indeed, we will show discarding objectives is the mechanism that lets operator applications continue to be effective.

As an example of refining knowledge, we will modify the cyclic toss example shown in Figure 10. Figure 12 shows an attempt by our agent to apply the knowledge of cyclic tossing using a helium balloon instead of the trusty bean bag. As shown in Figure 12, this attempt fails miserably, since the balloon flies into the air rather than taking the standard trajectory to the other hand. The agent now has to determine how to recover from the current situation and whether the balloon will ever land in the left hand. Recovery could include grabbing for the balloon. The important issues addressed here are how the agent learns that tossing the balloon up will not work, and what actions are taken given that knowledge.[1]

To determine that the helium balloon will never come down the agent needs knowledge that might have been unavailable prior to tossing the balloon. It could, however, suggest the explanation that helium balloons don't come back down to the same location. Without knowledge such as this, no basis exists for believing that the balloon won't land in the agent's hand some time in the future. Observation by itself is not enough to be sure because we are looking at a continuous process. At some time we have to draw a line distinguishing whether the agent believes the balloon will come down or not. Once the agent has decided that tossing a helium balloon was a bad idea, this knowledge needs to be used to terminate this chain of actions for achieving the goal of juggling. The catch (left) operator needs to be augmented with the knowledge that it will be ineffective when attempting to catch a tossed helium balloon. Likewise, the cyclic-toss operator needs to be augmented because it will also never reach its objective of a cyclic toss of the helium balloon. Both of these additions are operator termination knowledge. Of course, other knowledge could be learned that would also refine the cyclic-toss operator and inhibit it when the object to be tossed was a

[1] The agent could, of course, toss the helium balloon down and catch it as it rises. For simplicity we assume our agent does not have the knowledge that allows it to consider this form of juggling.

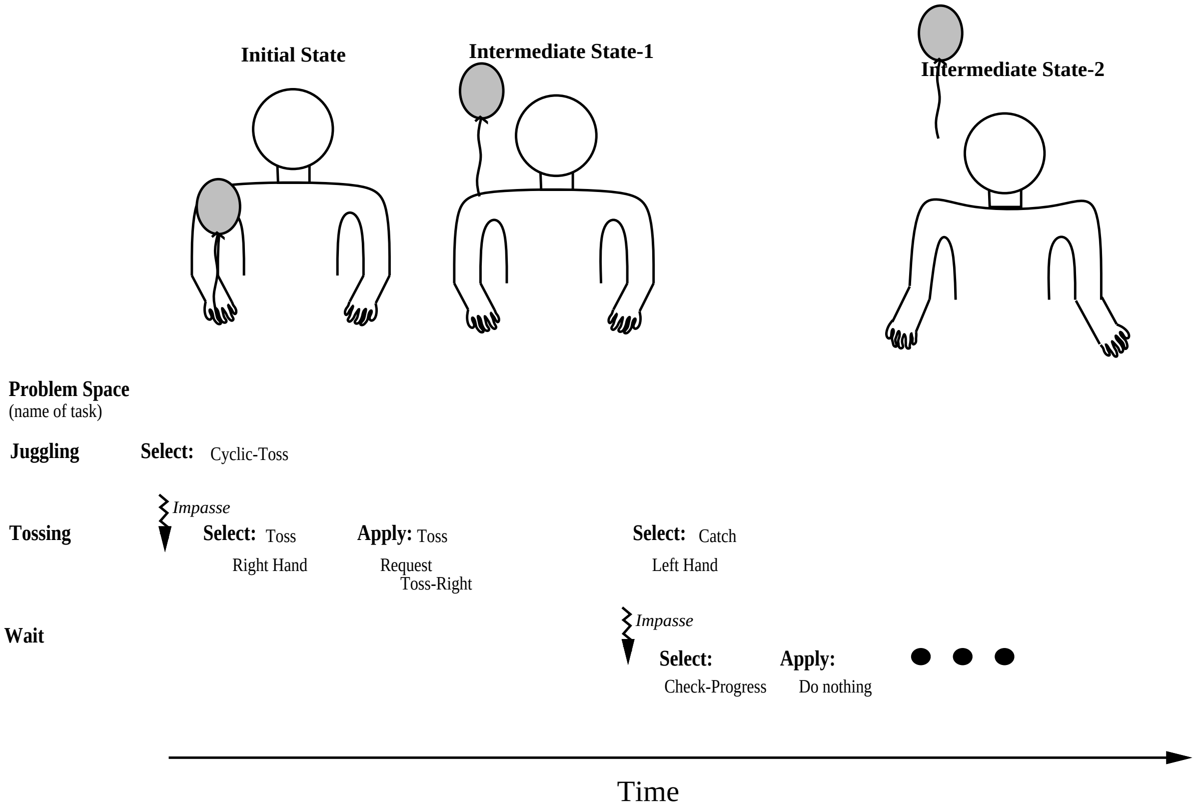

Figure 12: Frustration when tossing a helium balloon

helium balloon. We do not address the need to change the juggling goal, because it is unclear that it is unachievable. There could be other jugglable objects available.

Finding that an objective is unachievable can happen at any time during the problem solving involved in trying to achieve it. If the problem solving is done completely internally, the initial situation can be restored. The external world cannot always be restored to the initial situation, so handling unachievable objectives is mainly an issue when working with the external world.

We do not treat this form of frustration further and have included it mainly to show that a mechanism must exist to terminate frustrated operators and that this mechanism may need to be knowledge rich. Unfortunately, even if the knowledge available for determining that the balloon will not return is inadequate, the cyclic-toss operator in our example must be terminated as the balloon really won't come down. Termination of the operator without using domain knowledge is an example of the second type of operator frustration – frustration over the lack of progress.

3.3 Operator frustration over lack of progress

In many cases of interaction with the external world, it is difficult to tell whether the agent should continue waiting for a result to occur or abandon the pending operator. This situation can occur either because the agent has a lack of knowledge about how to measure progress, or because the rate of progress is slow. In this type of slack time impasse, the external world has been requested to perform some action, but the changes (or lack of changes) in the external world do not indicate

If The problem space is Juggling and
the operator is cyclic-toss and
the toss-right action has been requested for an <object> and
the location of <object> is over the agent's head
then the cyclic-toss operator should be considered complete and
any objectives established by cyclic-toss should be removed

Figure 13: Example of over-general operator termination knowledge

that progress is being made towards the objective associated with the action.

We can modify our tossing example to create an external world situation of this type of by tying the balloon to the bean bag. By changing the relationship of the balloon's lift and drag to the bean bag's weight, we can continuously vary how long it will take for the bean bag to get from one hand to the other, and likewise the rate of progress of the toss. Thus, we can ensure that the combination of balloon and bean bag will get to the left hand, but that it will take an arbitrarily long time (we have a similar situation when we are juggling a helium balloon but don't have knowledge about helium balloons). At what point does the agent give up? And what is it giving up on? Tossing, certainly, but what other components of the situation are considered important?

There doesn't seem to be a single answer to when the agent should give up. Further, no basis exists for expecting that, when juggling the balloon, the balloon should be considered the cause of the decision to give up. It could be wind or any number of other items in the situation that are identified as the cause. In short, with no knowledge to help, the agent can't determine how to assign credit or blame. In our case, if the helium balloon cannot be assigned blame for the problem, then the termination knowledge might be applicable as soon as any object is tossed.

The agent could take the radical approach of terminating the operator as soon as it finds that it cannot measure any progress. This approach has two problems that can be illustrated with juggling the balloon bean bag combination. The first is that when the balloon bean bag combination comes down in the left hand the agent should be able to recognize that this event occurred because the balloon bean bag combination was thrown in the past, and that this throw was part of a cyclic toss so that the balloon bean bag combination can be tossed back if it is still appropriate. This is a direct inference from the fanatical completion assumption. The fact that the balloon did come down means that the operator did not need to be terminated, and thus the learned termination knowledge is over-general and unfortunately will be applicable in other situations where waiting could possibly be more appropriate. An example of over-general termination knowledge for the cyclic toss operator is given in Figure 13 where, any time the object being tossed goes over the agent's head the cyclic toss operator will be considered completed. In the next section we will briefly touch on the problem of recovering from an over-general operator termination to reach the original objective. Recovery here is similar to that in Section 2.3 and it means setting up the conditions so that work on the original objective can continue. The second problem is a form of the masking problem [Tambe and Rosenbloom, 1993]. Since the agent should use the experiences in the external world to build up its knowledge base, sometime in the future the agent would want to create more specific knowledge that utilizes the expanded knowledge base. However, the original termination knowledge will mask the new knowledge because an impasse will not occur allowing the new knowledge to be utilized. We will not address the masking problem further in this paper.

3.4 Recovering from an over-general operator termination

Terminating the PSCM operator means that the context the operator provides is not available for interpreting what is happening in the external world. As an example, suppose a high toss of a bean bag comes down into the left hand some time after the catch (left) operator and cyclic-toss operator have been terminated by the knowledge in Figure 13. How does the agent know to continue with the next toss? To continue, the agent has to understand that this change in the external world is a result of the toss (right) action request and is part of a cyclic-toss. Thus, to understand what the bean bag coming into the hand means, the PSCM context has to be restored; then the processing can continue if it is still appropriate.

In Section 2.3 we showed two different methods for restoring the problem solving context after a non-contemporaneous chunk didn't apply, that let us recover from the disruption caused by the non-contemporaneous chunk. Once the context was restored, then the problem solving continued naturally, and the objective of the operator was achieved. The first method restored the context via re-planning from the original objectives and the new situation. The second method continually saved the state of the planning activity, so that the problem solving context was immediately available when the disruption occurred. Variations of both these methods can be used in recovering from the over-general operator frustration knowledge, providing us again with a continuum of solutions. Re-planning is exactly the same as in Section 2.3 and generates a new context that is like the old one, different only in that new symbols may be generated for the same objects. However, the method of sharing state in Section 2.3 doesn't work directly because the termination knowledge removes all traces of the terminated operator. Instead, before the operator is terminated, appropriate portions of the context can be memorized. The memorized context can then be used to recreate the context at the correct time. This memorization and re-creation method is similar to the shared state of Section 2.3 because the problem solving continues from where it was stopped without re-doing any of the planning.

Restoring the context through planning

Mitchell's robot system [Mitchell, 1990] architecturally plans until an action is known to be on the path to achieve the goal, then it always terminate the operator, and re-plans for the next action. Figure 8 showed Mitchell's system learning the cyclic toss, ignoring the external world response time. Figure 14 shows how a PSCM implementation of Mitchell's method would learn the cyclic-toss operator when external world response time is considered. As in the problem solving trace where the agent waited, Figure 11, the left hand side of Figure 14 shows the problem solving trace while learning and the right hand side shows it after learning. In Figure 14 the cyclic-toss operator is terminated as soon as a real action request is made. This puts waiting for a response outside of any of the operators, as shown in Figure 14 by the lack of indentation.

Re-creating a previous context

The context that needs to be restored by recovery includes the objectives of the operator (i.e., the goal that describes what the operator is trying to achieve) and, possibly, other state information. The actual operator doesn't need to be restored if processing for the objective can continue. Restoring the context when a result is observed requires some methodology for recognizing when an external change should be considered a result. This methodology can be knowledge-lean and assumption-rich, assuming the first observable change in the external world is the result of the

Initial Problem solving Trace
(before learning)

```
cyclic-toss
    toss (right)
      toss-right action request
    catch (left)
      catch-left action request
    Note Intermediate State
    toss (left)
      toss-left action request
    catch (right)
      catch-right action request
  toss-right action request
Wait for Toss Completion
cyclic-toss
    catch (left)
      catch-left action request
    Note Intermediate State
    toss (left)
      toss-left action request
    catch (right)
      catch-right action request
  catch-left action request
cyclic-toss
    Note Intermediate State
    toss (left)
      toss-left action request
    catch (right)
      catch-right action request
  Note Intermediate State
  toss-left action request
Wait for Toss Completion
cyclic-toss
    catch (right)
      catch-right action request
  catch-right action request
```

$\Longrightarrow$

Final Problem Solving Trace
(after learning)

```
cyclic-toss
  toss-right action request
Wait for Toss Completion
cyclic-toss
  catch-left action request
  Note Intermediate State
cyclic-toss
  toss-left action request
Wait for Toss Completion
cyclic-toss
  catch-right action request
```

Figure 14: Learning when always re-planning and actions take time

requested action. The methodology can also use various knowledge sources, such as previous experience in this domain, or an internal model. The key is not the restoring of the context, for that is available at the time the termination knowledge is learned, but rather recognizing *when* the context should be restored, as this situation is not available when the termination knowledge is generated.

Creating the expected situation and then using this created situation as the source of knowledge of when the restoration should occur, is one way to generate the restoration knowledge. In our example, the expected response is the bean bag (or balloon) in the left hand. We want the recognition of this response to lead to continuing the cyclic toss. To continue the cyclic toss we need to install the toss-twice objective and mark the objective to indicate the object has completed one toss. Thus, the restoration knowledge might be what is shown in Figure 15. The knowledge in Figure 15 restores the context that was available when the operator was terminated. Once restored, the catch (left) operator can apply, completing the next step towards the original operator's objective (do cyclic-toss). The objective can be removed, if necessary, by the same process described in

If The problem space is Juggling and
the toss-right action has been requested for an object and
the status of the object is moving
the location of object is close-to <hand>
<hand> is the left hand
then the cyclic-toss operator should be proposed with
the toss-twice objective installed as a sub-task of juggling

Figure 15: Example of over-general restoration knowledge

Initial Problem solving Trace
(before learning)

cyclic-toss
  toss (right)
    toss-right action request
  catch (left)
      Slack time starts
      Visualize-result (of toss)
      Build-Recognition
      Build-Termination
Slack time continued
  Wait for Toss Completion
cyclic-toss
  catch (left)
    catch-left action request
  Note intermediate state
  toss (left)
    toss-left action request
  catch (right)
    Slack time starts
    Visualize-result (of toss)
    Build-Recognition
    Build-Termination
Slack time continued
  Wait for Toss Completion
cyclic-toss
  catch (right)
    catch-right action request

$\Rightarrow$

Final Problem Solving Trace
(after learning)

cyclic-toss
  toss-right action request
Slack time starts
  Wait for Toss Completion
cyclic-toss
  catch-left action request
  Note intermediate state
  toss-left action request
Slack time starts
  Wait for Toss Completion
cyclic-toss
  catch-right action request

Figure 16: Learning when saving planning state

Section 2.3.

Note that, just like the termination knowledge in Figure 13, the restoration knowledge show in Figure 15 is overly general. Some time in the future, when the agent is juggling and the juggled object is close to the left hand, the agent will suddenly have the goal of doing the last half of a cyclic-toss! The problem is that the restoration knowledge is not linked to the initial operator or to the specific type of juggling goal that led to the cyclic-toss operator being proposed. This linkage can either be explicit, conveyed through some constant shared by this specific juggling goal and the restoration knowledge, or by linking to the previous step in some manner, or through some more complex mechanism like an intention. We will assume that the restoration knowledge can be made specific enough.

Figure 16 shows a problem solving trace for doing the cyclic toss both before and after learning when the state of the plan is saved. Its format resembles the other traces except that the start of slack time is explicitly represented because the slack time starts as part of the operator and is continued after the operator is terminated.

The trace in Figure 16 begins like the trace in Figure 11 until the slack time occurs. When slack time within an operator is initially detected, rather than just waiting, three operators apply. These operators generate the knowledge to both split the operators above and recognize when the proper external world events have occurred so that the context can be restored. The visualize-result operator in effect does a one-step lookahead to determine the action's expected result. In this scenario, the toss-right is the only action request that has been made, so visualize-result changes the current situation to one showing the bean bag close to the left hand. Build-recognition then generates restoration knowledge such as that shown in Figure 15 for the cyclic-toss operator using the visualized result and portions of the current context.[2] Build-termination creates the termination knowledge for the cyclic-toss operator, like that shown in Figure 13, but not based upon any domain knowledge. The termination knowledge for the catch (left) operator is based solely upon the existence of slack time. Slack time in this case is recognized by both the lack of immediately available knowledge and the fact that an object was requested to be tossed from the right hand. The cyclic-toss operator's termination knowledge is based upon the same information as the catch (left) termination knowledge, and the non-existence of alternative operators to implement the cyclic-toss. This termination knowledge applies, terminating the operators in Figure 16, but doesn't end the slack time (the external world has to respond for the slack time to be over). Thus, the agent now waits for the bean bag to come close to the left hand outside of the operator. These three operators have split both the cyclic-toss and catch (left) operators, while building the recovery mechanism that keys off the expected result of the bean bag being close to the left hand. This is the basic mechanism of saving state.

We continue the processing in Figure 16 when the bean bag is close to the left hand because the recognition knowledge re-establishes the removed context. The context is exactly the same as before the split, enabling the agent to select the cyclic-toss operator again, and the catch (left) operator under it. After catching with the left hand and noting the intermediate result, the agent still has the objective to toss the bean bag back to the right hand, and the toss operator is used to begin achieving this goal. Once tossed, we again have slack time, this time occurring within the catch (right) operator. This catch operator and the cyclic-toss operator are again split in a similar manner. The previous chunks did not apply because they were dependent upon the left hand catching, not the right. Finally, when the bean bag is close to the right hand, it is caught and the cyclic-toss is completed.

The right hand side of Figure 16 is much simpler. It shows the cyclic-toss operator being selected and applied three times in achieving the cyclic-toss's objective. Once the toss from the right hand is requested the cyclic-toss operator is terminated by the previously learned termination knowledge. This knowledge also removes the objectives from consideration. When the bean bag gets close to the left hand, the restoration knowledge applies, re-creating the context, and the cyclic-toss operator is selected and applied, this time catching the bean bag, noting the intermediate state and tossing it back to the right hand. This toss also takes time, so this instance of the cyclic-toss operator is also terminated. Finally when the bean bag approaches the right hand, the last instance of the cyclic-toss operator is selected, catching the bean bag, and achieving the cyclic-toss's objective.

[2] Creating this knowledge requires data chunking [Newell, 1990] the context. However, in this instance we have a generator for the context in the original context-building productions.

	Always Split Save state	Always Split Re-plan	Never Split (Always wait)
Cyclic-toss	3	4	1
Robot pushing a box	43	40	9

Figure 17: Number of operators with splitting at PSCM slack time

Continuing the PSCM operator

Continuing the work on the objective of the PSCM operator involves applying other operators. An important issue, then, is whether the correct operators will be available. Our contention is they are available because the applicable operators (after restoring the state if necessary) are exactly those that were originally available. In the left hand side of Figure 16, the first split occurs after the bean bag is tossed. A new proposal for the cyclic-toss operator was created by the build-recognition operator during the slack time. This proposal makes the cyclic-toss operator appropriately available at the end of the slack time. The catch (left) operator is available because the context provided by the recognition knowledge is similar to the original context

3.5 Dynamic operators

We have shown that when interacting with an external world that has uncertainty in the results it provides to actions, then operator termination knowledge must be learnable. However, distinguishing between the situations where operator termination knowledge should be learned and those where it should not is difficult. Also, the operator termination knowledge may be over-general, applying in situations that are similar to the learning situation but that don't require the termination knowledge. To recover from the application of operator termination knowledge and continue the problem solving requires that the context that existed before the termination be restored. The use of operator termination knowledge separates the knowledge in an operator into two components: an *action-request* component, that determines what action should be done, and a comprehend-result component, that completes the problem solving associated with the original operator. In this section we show that if operators are always split when an external world action request is made, then the number of operators executed to achieve some result can grow tremendously. Although growth can be moderated by splitting in a less pedantic manner, the reason for splitting remains: progress in the external world is too slow. In short the external world's response time defines the granularity and number of PSCM operators that a model uses to do a task.

Figure 17 shows the number of operators used to achieve two tasks where either the operators are always split after an action request or never split. The first task is the familiar cyclic toss example and the second is a more complex task of a robot pushing a box. The model used for each task sent action requests to the external world at each step of its planning and used the results to verify that the plan was working according to its expectations. The data in the first and second column comes from a Soar system that always splits the operator whenever an action request is made, and then restores the context when the desired result becomes available, so that the processing toward the operator's objective can continue.[3] The data in the third column comes from a similar Soar system that never splits the operator when an action request is made, but instead waits until the desired

[3]The robot "Always split with re-planning" is a hand simulation and the "Always split saved state" implementation actually saved the planning state rather than getting rid of it at termination and later restoring it.

response is found in the situation, allowing the operator to continue. Figure 17 shows that always splitting means many more operators in both recovery scenarios. These extra operators can limit how effective the agent can become in doing some task. This limitation shows up if operators are being assigned some real time, like 50 msec [Newell, 1990], so that a model can be correlated with data from people. The extra operators may make the model overpredict the data. The limitation also rears its head when doing planning, for planning is a NP-hard task [Chapman, 1987] and more operators reduces the effectiveness of any planning activity. The number of operators becomes even higher as the granularity of interacting with the external world becomes finer. To ease this growth effect, we notice that we only want to split operators when no progress is being made during slack time, not at every action request. The number of operators used by a model that uses this splitting philosophy to do a task would be somewhere between the never-split and one of the always-split cases. Since splitting in this model is dependent upon how quickly the external world responds, the number of operators is ultimately dependent on the response time characteristics of the external world.

The number of operators to do a task would seem to only grow via splitting. However, one of the interesting properties of operator termination knowledge is that PSCM operators split by such knowledge can *recombine* to become a single operator again, if the PSCM is defined as waiting for all immediate knowledge to apply before selecting a new operator.[4] Recombining happens if the external world responds immediately, so no slack time exists in the previously split PSCM operator. Thus the conditions may be right for operator termination, but with the external world's response available the rest of the knowledge associated with the PSCM operator can still apply. If the operator is allowed to continue, delaying the use of operator termination knowledge, then this execution of the operator can achieve its objective in the normal manner. This recombining of an operator, after it was split, again shows how the response of the external world defines the amount of work an operator can achieve.

As a final observation, when working completely with internally generated results (e.g. simulating a sequence of events as in planning), split operators recombine. This happens because an internal model of the external world provides the expected response to an action request immediately. This recombining makes internal processing, like planning, simpler because the number of operators is kept low.

4 Splitting with Multiple Objectives

In Section 3 we were concerned only with checking that progress was being made during slack time. However, if we have multiple tasks, waiting during slack time could be considered inefficient use of the cognitive resources available. The PSCM has a sequential operator bottleneck, that is, it executes just one operator at a time. When waiting, the PSCM is observing progress being made in one of its tasks. Rather than just observe progress, it might be able to make progress on a different task and return to the original task sometime later. To achieve maximum use of the cognitive resource, we would like to have the PSCM operate on one of its other tasks when the work on the currently executing task is suspended (waiting for an result to be observed). This section describes

[4]Soar currently works this way, but the definition of the PSCM is unclear about when available operators are considered for selection. A PSCM operator could implement several functions, each defined by different operator termination knowledge. Over time, as the operator applies, different termination knowledge can become applicable possibly starting the selection process for a new operator. The PSCM is unclear on whether the first instance of applicable termination knowledge starts the selection process, or if some other condition of the PSCM architecture is used to determine that some of this knowledge can be ignored. Soar uses quiescence to determine the termination knowledge that defines the termination conditions of an instance of an operator.

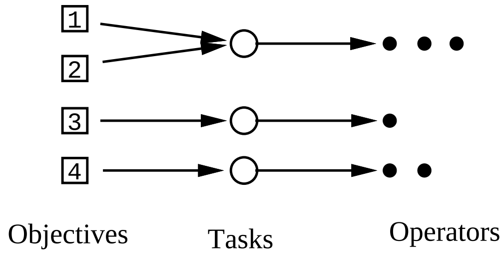

Figure 18: Multiple Objectives and Multiple Tasks

what keeps this transfer of control from happening in the PSCM, then discusses two solutions and how models built using those solutions are related. By transferring the control between multiple tasks we are able to use the available cognitive resources efficiently.

A task is an abstract description of a process to achieve an objective. In PSCM terms, a task can be represented by either a large operator that may include many steps, or multiple smaller operators each doing only a few steps. When working with multiple objectives, either one has a single task that can achieve all the objectives, or multiple tasks that one hopes can achieve all the objectives together. Figure 18 shows a PSCM model with multiple objectives, the first two of which are associated with a single task that has multiple operators to implement it.[5] The third objective is resolved in a task that has only a single operator, while the fourth objective in Figure 18 has two operators.

When working with multiple objectives and multiple tasks, external events that occur while processing any of these tasks can either indicate that the desirability of working on one of the tasks has changed, that the application of one of the operators can proceed, or that the achievement of a objective has occurred. As an example, the external world could change (my terminal might burst into flames) so that an important task (saving my skin) should interrupt a less important task (typing these words). In PSCM terms, the changing of the external world would make the operators for fleeing, getting a fire extinguisher, etc. more salient. We want these more salient operators to be selected and applied, ignoring what we are currently doing. In general, we would like the processing of multiple tasks at the PSCM level to occur as just a sequence of operator selections followed by their application, always working on the most salient task.

Unfortunately, it is unclear how such a sequence of PSCM operators could take advantage of the slack time of the operators. Utilizing slack time within an operator means that one of the following must occur:

- Split the PSCM operators with slack time into a sequence of smaller action-request, comprehend-result operator pairs.[6] This split uses the operator termination mechanism described in Sections 3.2 and 3.3 and the recovery mechanism in Section 3.4 to remove the slack time from the operator but allow the task to continue processing in the future.

  This option removes the slack time from the operators. Thus we interleave small operators, and a lack of something to do is the only reason to wait.

- Combine operators from multiple objectives into larger operators. Since the PSCM has no

[5] This is a modification of Figure 1 in [Covrigaru, 1992].

[6] If the external world is very close to the internal model then perhaps the result does not need to be comprehended and just a sequence of action-requests is needed. Throughout this paper, though, we will refer to these operators as pairs.

sequentiality requirements within an operator application, only the sequentiality apparent in the data dependencies will restrict multiple objectives from being pursued in parallel. This creates situations like at the top of Figure 18 where the first two objectives are processed by the same task.

This option overlaps all the available processing so that it occurs within a single operator. Thus, any waiting that exists after combining, is due simply to lack of something to do.

- Break the sequential selection and application of operators allowing multiple operators to be applied simultaneously in the same problem space. Rosenbloom has proposed changing Soar to allow this [Rosenbloom, 1993].

  This option also overlaps all the available processing, so that it occurs simultaneously. However, it provides some architectural assistance to the handling of the simultaneous tasks.

Investigating the implications of supporting multiple simultaneous operators is beyond the scope of this paper. The next two parts of this section investigate the first two methods and how they are related. We look at splitting operators at slack time and then interleaving first, because the splitting portion should be familiar by now. The interleaving is fairly simple and all the mechanisms for interleaving are in the Soar and PSCM architectures. However, interactions can exist between the two tasks. As an example, given the two objectives of painting the ladder and the ceiling, the agent should paint the ceiling first. This is a real problem when working with split operators because during a split we have removed the information that other operators could use to constrain their behavior. After interleaving we look at two methods of combining operators, one that requires the knowledge associated with a strong model of the interactions between tasks, and one that doesn't. We show that the knowledge-lean system, breaks both the PSCM concept of an operator implementing a function from one state to another and the fanatical completion assumption. Thus we discard it. The knowledge-rich strong model system comes from [Covrigaru, 1992], and we review it as a method of planning actions and parallelizing independent interactions with the external world.

4.1 A second task: Robot pushing a box

We begin by modifying our tossing example so that we have multiple operators available. In addition to tossing a bean bag from hand to hand, we make our agent a robot and give it a second objective of pushing a box from room to room. An example problem in the robot domain is shown at the top of Figure 19. At the bottom of Figure 19, the first steps toward solving the problem are shown. The Push-Box-Thru(Door) operator has been selected Ⓐ in the move-robot problem space as the first operator to try for this task. This operator cannot apply because of an unresolved precondition. As before, the PSCM casts the lack of knowledge of how to resolve this precondition as a task to be solved in a problem space. The task is formulated to have an initial state (the current state of the robot) and an objective of the unresolved precondition of the push-box-thru(Door) operator (the robot and box are at the door of Room 2) in the same Move-Robot problem space Ⓑ. In this Move-Robot problem space the go-thru(Door) operator is selected Ⓒ as the most useful operator on the path to achieve the objective of getting the robot into Room 2. However the go-thru(Door) operator also cannot apply and thus another new problem space is created Ⓓ to resolve its precondition. Finally we bottom out when the go-to(Door) operator is selected Ⓔ and actually applies Ⓕ. This application changes the state of the external world by bringing the robot to the door. Once it is at the door, the termination criteria of the Move-Robot problem space for the "go-to" task is reached Ⓖ and at the same time the preconditions of the go-thru(Door) operator are resolved so it can apply Ⓗ. This type of processing is repeated until one of the push-box-thru(Door)

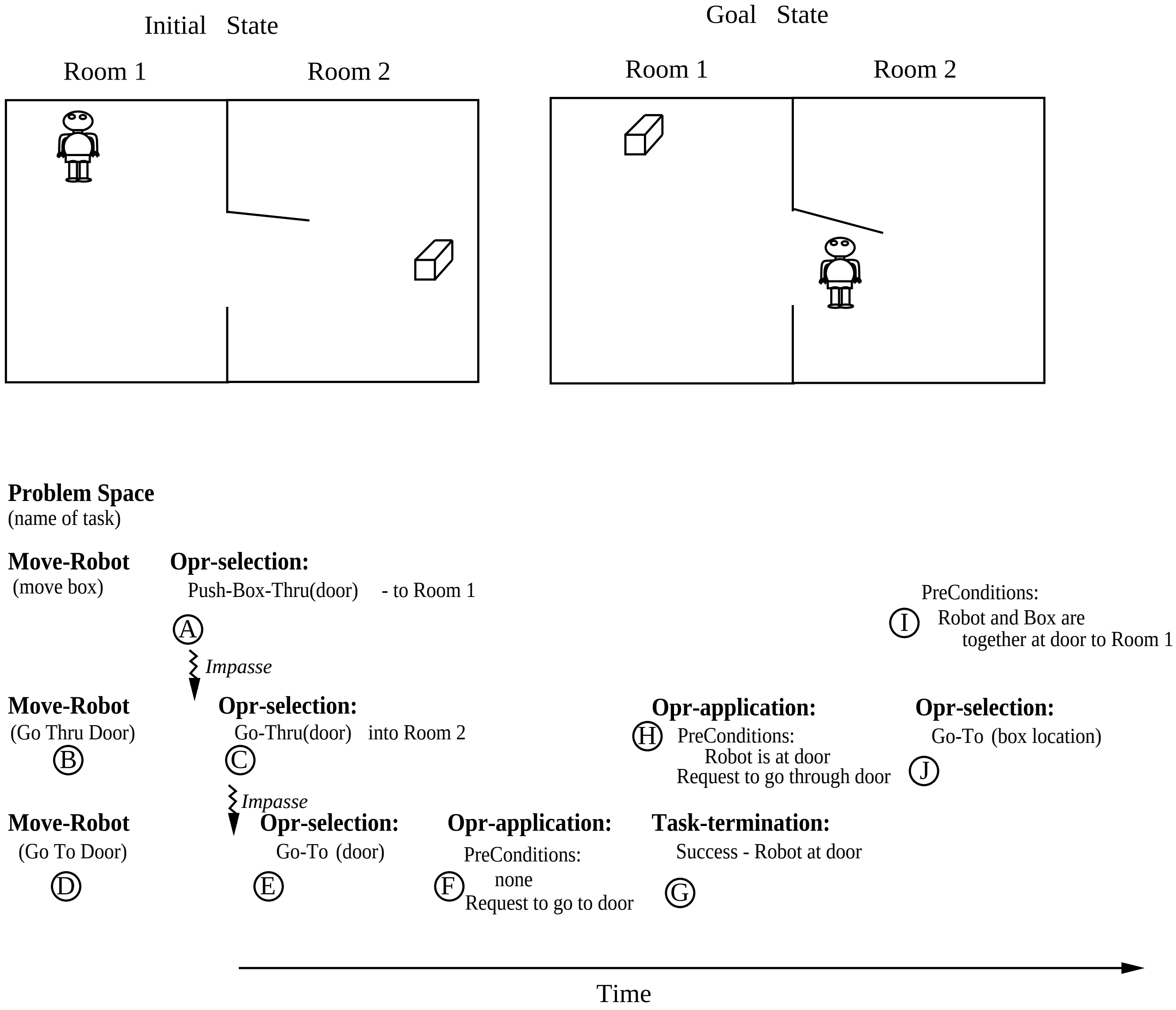

Figure 19: Second Task: Robot pushing a box

operator preconditions is met (I). Note that we still have not reached the point that the push-box-thru operator can apply. So the termination condition for the second Move-Robot problem space has not been reached and other operators should be available for selection and application in that space. Thus, we see another selection of the go-to operator (J) on the far right that sends the robot to the box's location.

Now that we have two tasks defined, what are the issues in combining them so that we can juggle and move at the same time?

4.2 Multiple tasks via interleaving

We showed in Section 3 that we couldn't just wait within an operator for a desired external world response because of the uncertainty of the external world producing the desired response. We then showed that splitting an operator into an action-request component and a comprehend-result component removed the waiting time for the desired result from within the operator and still achieved the objective of the operator. In this section, we have a different reason for not wanting to wait within an operator; we want to make progress on other tasks while waiting. Splitting the operator allows us to make progress on other tasks, because it removes the operator for the suspended task allowing other operators to be selected. If these operators are for other tasks then the agent can make progress on these tasks while the first task is suspended.

The problem with simply splitting is that the operators for the second task might interfere with the operators for the first task. Nonlinear planning handles this situation if the knowledge of which tasks are being combined and what all the operators do is available. However, splitting removes the context of the split operator and thus the knowledge that would be the most useful to a planner. An example interference scenario is the robot attempting to juggle and open the door at the same time. Since the robot requires the hand that opens the door to be empty, throwing the bean bag to the hand that has been directed to open the door interferes with opening the door. Likewise starting to open the door with a hand, might also interfere with the catching of a previously thrown bean bag to that hand. This is not simply a case of nonlinear planning, because the splitting removes the easily checked context and expectations about both the tossing and opening objectives. A nonlinear planner would use the context to generate the constraints on the actions.

Controlling the interaction, when splitting, requires knowledge about both the intention to catch the bean bag with the hand and the intention to open the door with the hand. Once we have knowledge of both intentions, then knowledge can be learned that constrains the solution so that the interference is avoided. This constraining knowledge can be generated before the scenario occurs by planning, or it could be generated after the scenario occurs by replaying what happened. Generating the constraining knowledge by planning is difficult, because there is no focus to the plan other than time going forward. Let's go back to our example where the bean bag is in the air and the agent wants to open the door with the catching hand. To determine a priori that the catching hand should not be used, we have to set up the situation so the conflict is apparent. This implies that we have to set up the conditions of the bean bag being close to the catching hand. But how do we determine that this is the salient condition to set up? Perhaps the movement of the bean bag would provide the necessary clue. However, doing this by planning does not seem as straightforward as looking at an error caused by an interaction.

The constraint for tossing to the hand opening the door would be similar in form to the recovery knowledge of Figure 15. It would also recognize the intention of using the catching hand for opening the door. The action of the constraint would be to restrict the bean bag from being tossed to the hand engaged in opening the door. Like the recovery knowledge, the constraint knowledge also has to be linked to the intention to toss the bean bag, or it would be over-general. As mentioned before, these intentions could take many forms. The issues surrounding their exact form and persistence are the subject of further research.

4.3 Multiple tasks via combining operators

Combining operators means taking the portion of a set of tasks represented by a set of operators and deciding to make a single task out of doing the operators together. This new task is represented by a single operator (we'll call it the C-operator) and the actual procedure is similar to the way that the cyclic-toss operator was learned in Figure 2. Once begun, if more operators become available, these new operators might join the combining process. Combining operators is important because it can cause previously independent tasks to be operated on in parallel, thus reducing the time to do both tasks. It leads toward larger and larger operators that encompass more and more tasks.

The first issue with combining operators is that the tasks they represent might not be completely independent. Non independent tasks interact with at least some of the actions in one task affecting the performance of the other tasks. This is the same nonlinearity in plans to reach conjunctive objectives that we saw in Section4.2. Here we are simply interleaving in a sub-goal. The agent still has the interacting task problem and can either plan how to achieve all the tasks to determine the "best" way to handle the interactions, or it could assume interactions are uncommon, and fix any problems when they come up.

The lack of simultaneous completion of the operators in the combination process is the second issue with combining operators. A range of options is available for terminating the combination process and its associated C-operator. At one end, termination occurs when all the operators being combined are completed. At the other end, we terminate when the first operator completes. We will explore systems at both ends of this range.

We will go over two methodologies for combining operators. The first methodology is from [Covrigaru, 1992] and stresses the planning aspects of combining. Covrigaru terminates the C-operator only when all the operators in the combination process have terminated. The second methodology is knowledge-lean. It always starts the combination process when slack time is encountered for an operator, this initial operator is the C-operator. It ignores the planning aspects so it has no need of a model of how operators interact. It terminates the combination process when the C-operator has completed its original function, possibly leaving only partially completed the other operators being combined.

Planning the combination

Covrigaru combines operators in a deliberate way using a strong model of the interactions between actions and generating a plan that optimizes the issuing of all the actions from the combining operators. A strong model has interaction information for all the possible interaction situations. He also uses the interaction model when a new operator becomes available to decide if the operator should be added to the combination process. This planning is the strength of Covrigaru's work. By careful planning larger and larger PSCM operators are created that encompass the knowledge of dependencies between all the actions they can issue. Thus the actions are not only issued in an implementable order, but in parallel when possible. The weakness of the work is that it requires a model of the interactions between actions to do the planning and to decide if a new operator should join the combination process.

Covrigaru starts his process of combining operators by explicitly creating a new PSCM operator, called a "merge" operator, whenever the correct conditions exist for merging the tasks that the available operators represent. This merge operator executes the actions of all the to-be-merged tasks. Figure 20 gives an example of merging by showing three operators (the circles at the top of the triangles) with their associated actions (the sequence of boxes under the operators) in the top part of the picture. In this task only two of the operators, push-box-thru(Door) and toss (right), are initially available for selection. The operator toss (left) becomes selectable when the bean bag is in the left hand. Instead of processing the operators sequentially, a new operator is created that merges the actions of the three operators as shown in the bottom part of Figure 20.

Covrigaru uses a declarative form of the operator actions and a model of how the actions interact to control merging so that the resulting operator issues the action requests in at least an achievable, and possibly optimal, manner. When an action from a sub-operator becomes available

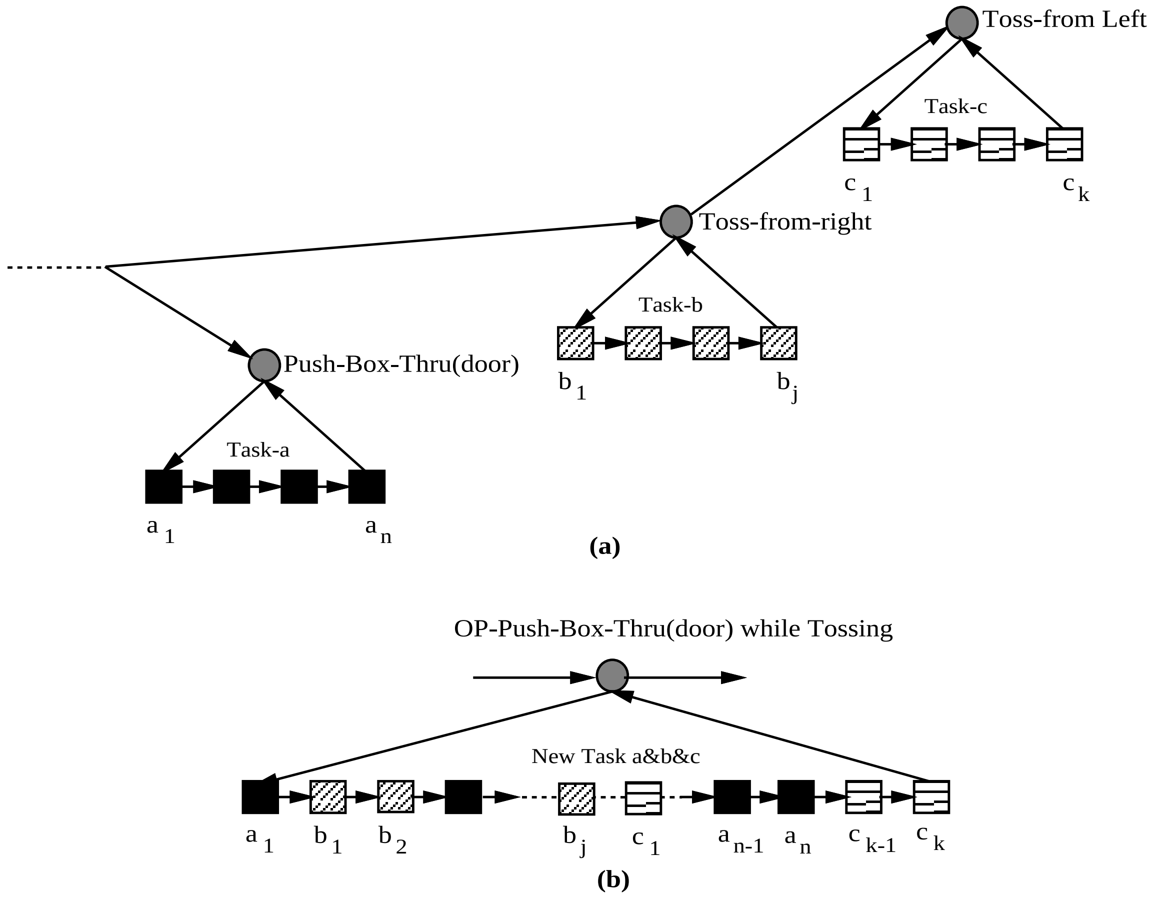

Figure 20: Merging of PSCM operators[7]

to be requested it is heuristically evaluated along with all the other available actions to decide the next action for the merge operator to request. In our case, we can arbitrarily set the heuristic evaluation function to prefer moving the robot over tossing a bean bag. Then all the actions for push-box-thru(Door) will be preferred over toss (right) actions. But since push-box-thru(Door) has some slack time, its actions are not always available, allowing the actions associated with toss (right) to be requested in what would have been the slack time of the push-box-thru(Door) operator.

If a new task, represented by an operator, becomes available during the merging process then this new task can be added to the set of tasks being merged and its actions can also be considered and requested. This happens in Figure 20 when the toss (left) operator becomes available for selection. The toss (right) operator has completed and the push-box-thru(door) operator has not completed. How the new task will affect the current combined task is checked before the new task is allowed to join the merging process. Checking uses the same model used to determine that the original tasks should be merged. The merge operator is completely applied when all of its sub-operators have completely applied.

Figure 20 shows that what is really happening is the actual operators (that do the task) are being interleaved in the subgoal. This interleaving is done under the control of the planning mechanism. Chunking is converting this interleaving into parallel applications, when possible.

[7] This is a two task modification of Figure 6 in Covrigaru [Covrigaru, 1992] where the toss (left) operator becomes available for merging when the toss (right) operator has completed.

Knowledge-lean combining

An alternative knowledge-lean method is to start combining operators when slack time is noticed. Slack time causes an impasse to occur because the definition of slack time includes a lack of knowledge at the PSCM. This lack of knowledge can be handled in the same manner as all other lacks of knowledge in the PSCM, namely by the creation of a problem space to address the lack of knowledge. The task of this problem space is to find a PSCM operation to do. If operators from a problem space are eligible for selection then posing the task that allows these operators to be selected is one of the ways of finding a PSCM operation to do during the slack time. Unfortunately this method of combination causes the function implemented by the C-operator to become ill-defined, having side-effects that could not be expected when the operator is selected or terminated. It also terminates the combination process when the C-operator terminates, causing the fanatical assumption to be violated for the other operators in the combination process.

Combining operators at slack time causes the operator functions to become ill-defined because it overloads the meaning of the slack time impasse. When an impasse occurs in the PSCM because an operator cannot complete, the task that is formulated for that impasse and the knowledge learned are supposed to resolve the lack of knowledge that caused the impasse. In the case of slack time, a lack of knowledge exists, but it is a lack of knowledge as to what to do while waiting for a response from the external world. In the past we have selected checking progress on the original task as what to do. But in a multiple task model, we want to consider working on another task as a more reasonable alternative. The C-operator in this method is the original operator that had slack time. However, if we select operators for other tasks to run during the slack time of the C-operator, knowledge about those operators is learned in the context of the C-operator. This knowledge is independent of the function the C-operator is implementing and can be applicable in other situations. Thus, when the C-operator applies in the future, it changes the state according to its original definition, but it also might make other changes to the state that are not really part of its original definition. These extra changes might even cause the original definition to fail. This would mean that the C-operator and the knowledge learned from the sub-operator interfered with each other.

Changing the function that a C-operator implements can make previously learned control knowledge incorrect. If the control knowledge that guides operator selection is the same for all situations then changing the function implemented will work fine, because the new function additions to the C-operator will apply only when the other operator is available for selection. However, in general control knowledge responds to different situations with different operator selections. Thus, even though the C-operator is correctly selected, the auxiliary changes it makes may not be desirable. This is not a problem for Covigaru because the C-operator is always a new operator that is dependent upon a tie from the smaller operators. Thus new control information has to be learned for the new operator.

Operator termination has a similar problem. In this method, unlike Covrigaru's combination process, the operators being combined are not treated equally. In particular, the sub-operator is terminated when the C-operator is completed, possibly before the sub-operator has completed. Unless this non-completion is handled, the fanatical operator application assumption will be violated. If the terminations only happen at slack time, then we can use the re-planning method of Section 3.4 to recover and complete the sub-operator. However, terminations can occur at other times in Soar and the PSCM places no restrictions on these terminations. Thus, using this method of combining makes implementing effective functions via operators impossible to guarantee.

This method of combining operators violates too many of our assumptions as to how PSCM and Soar systems should work without offsetting benefits to merit further serious consideration. We included it in this paper because it is an obvious method for combining operators afforded by the Soar architecture, and we wanted to describe the problems with it.

5 Discussion

We have introduced a number of key ideas related to the persistence of objectives. The first is that the fanatical completion assumption entails situations in which the persistence of an operator's objective exceeds the desired persistence of the operator itself on the goal stack. When an objective's persistence exceeds the desired persistence of its operator, we split the operator.[8] The split is accomplished by a dynamic redefinition of the operator's termination knowledge which may, of necessity, be overgeneral. This overgenerality has two consequences: it may prematurely terminate the operator under inappropriate conditions in the future, and it may cause a significant increase in the number of operators required to achieve an objective. Both these consequences are ameliorated somewhat by Soar's delay of termination until quiescence – if the world reacts within the bounds of the decision cycle, a split operator effectively recombines. Of course the process of splitting requires a concomitant process for continuing work on the objective at an appropriate time in the future. Specifically, it requires a potentially difficult context restoration process and consideration of the indexing involved in invoking the restoration. Although we have given some ideas for how to handle indexing and restoration by taking advantage of the context that is still available during the slack time that leads to splitting, this is clearly an area requiring further research.

Neither the persistence of an objective, nor the context restoration process has support from the Soar architecture at this point in time. However, a relevant proposal has been made, however, in the form of maintaining several complete contexts on the goal stack [Rosenbloom, 1993]. Consider what multiple contexts would mean for the cyclic toss example. If we had multiple contexts on the goal stack, then we could maintain the context of the cyclic toss indefinitely while doing other processing. When the bean bag got to the hand, the cyclic-toss operator would simply continue. Thus, it seems as if the cyclic-toss operator would never have to split, and no context restoration knowledge would be required. Unfortunately, the problem of terminating a cyclic-toss operator that will never complete remains. This is the case of tossing the helium balloon. The termination knowledge in this case is still likely to be overgeneral. Once it has been created, either the cyclic toss operator will be lost or some sort of recovery process will have to occur. Thus multiple contexts give us some architectural support but don't solve the fundamental problems for splitting and recovery.

Indeed, it may be premature to consider architectural support because the issues outlined here have been evoked by considering only a narrow range of the pertinent phenomena. The examples given here can be thought of as a mismatch of persistence in the small. Yet, if we consider the broad range of objectives held by an agent that acts over extended periods of time, it seems clear that most of those objectives persist long beyond the duration of the individual operators intended to achieve them. Put another way, it is not the norm that an agent has the opportunity to carry out a sequence of operators to achieve an objective with timely response from the external environment. Instead, our days are punctuated by the need to organize our resources to compensate for the delays between when we form an intention to achieve an objective and when the world presents the opportunity to pursue the next intentional step. We might think of this as a mismatch of persistence in the large. Before architectural change can come, we must first understand how the

[8] We can also create examples where the persistence of an operator's objective is shorter than the operator's persistence, but these cases are typically not problematic. An example problem of this sort of persistence mismatch, is shown in [Akyurek, 1992].

notions of splitting and recovery can be spread across the current architectural mechanisms for persistence in a way that makes some uniform sense for mismatches in the small and in the large.

References

[Akyurek, 1992] A. Akyurek. Means-ends planning: An example soar system. In J.A. Michon and A. Akyurek, editors, *Soar: A Cognitive Architecture in Perspective. A Tribute to Allen Newell*, pages 109–167. Kluwer, Dordrecht, The Netherlands, 1992.

[Chapman, 1987] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32, 1987.

[Covrigaru, 1992] A. Covrigaru. *Emergence of Meta-Level Control in Multi-Tasking Autonomous Agents.* PhD thesis, University of Michigan, 1992. CSE-TR-138-92.

[Fikes and Nilsson, 1971] R.E. Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, winter 1971.

[Laird and Huffman, 1992] J.E. Laird and S.B. Huffman. Chunking over supergoal changes: A proposed solution. Artificial Intelligence Laboratory, University of Michigan, January, 1992, Unpublished, January 1992.

[Laird and Rosenbloom, 1987] A. Laird, J.E. Newell and P.S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33(1):1–64, 1987.

[Mitchell, 1990] T. Mitchell. Learning stimulus response rules. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1051–1058. Morgan Kaufmann, July 1990.

[Newell *et al.*, 1991] A. Newell, G.R. Yost, J.E. Laird, P.S. Rosenbloom, and E. Altmann. Formulating the problem space computational model. In R.F. Rashid, editor, *Carnegie Mellon Computer Science: A 25-Year Commemorative*, pages 255–293. ACM-Press: Addison-Wesley, Reading, PA, 1991.

[Newell, 1990] A. Newell. *Unified Theories of Cognition.* Harvard University Press, Cambridge, Massachusetts, 1990.

[Rosenbloom, 1993] P.S. Rosenbloom. A modified psl proposal (version 3.1). Personal Communication, October 1993.

[Tambe and Rosenbloom, 1993] M. Tambe and P.S. Rosenbloom. On the masking effect. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 1993. Soar #93.12.