Compilation of Non-Contemporaneous Constraints

Robert E. Wray, III and **John E. Laird** and **Randolph M. Jones**
Artificial Intelligence Laboratory
The University of Michigan
1101 Beal Ave.
Ann Arbor, MI 48109-2110
{wrayre,laird,rjones}@umich.edu

Abstract

Hierarchical execution of domain knowledge is a useful approach for intelligent, real-time systems in complex domains. In addition, well-known techniques for knowledge compilation allow the reorganization of knowledge hierarchies into more efficient forms. However, these techniques have been developed in the context of systems that work in static domains. Our investigations indicate that it is not straightforward to apply knowledge compilation methods for hierarchical knowledge to systems that generate behavior in dynamic environments. One particular problem involves the compilation of non-contemporaneous constraints. This problem arises when a training instance dynamically changes during execution. After defining the problem, we analyze several theoretical approaches that address non-contemporaneous constraints. We have implemented the most promising of these alternatives within Soar, a software architecture for performance and learning. Our results demonstrate that the proposed solutions eliminate the problem in some situations and suggest that knowledge compilation methods are appropriate for interactive environments.

Introduction

Complex domains requiring real-time performance remain a significant challenge to researchers in artificial intelligence. One successful approach has been to build intelligent systems that dynamically decompose goals into subgoals based on the current situation (Georgeff & Lansky 1987; Laird & Rosenbloom 1990). Such systems structure procedural knowledge hierarchically according to a task decomposition. A hierarchical representation allows the performance system to react within the context of intermediate goals, using domain theories appropriate to each goal. Sub-tasks are dynamically combined and decomposed in response to the current situation and higher-level tasks, until the system generates the desired level of behavior. A hierarchical decomposition can thus generate appropriate complex behavior while maintaining an economical knowledge representation. In contrast, a flat, fully reactive knowledge base implementing the same behavior would explicitly represent all the possible combinations of subtasks that arise through dynamic composition within a hierarchy.

Unfortunately, performing step by step decomposition and processing at each level of a hierarchy takes time. This problem is exacerbated when the world is changing while execution takes place. For time-critical behavior, such decomposition may not be feasible. One possible solution is to have a system that supports both hierarchical knowledge and flat reactive rules. Reactive rules apply when possible. Otherwise, reasoning reverts to goals and subgoals. Many multi-level real-time systems approximate this general framework.

Critical questions concern which reactive knowledge should be included and where such knowledge would come from. A promising answer is the dynamic compilation of reactive knowledge from the system's hierarchical knowledge. To date, knowledge compilation algorithms such as explanation-based learning (EBL) (DeJong & Mooney 1986) have been used to compile the results of *off-line planning* activities into control knowledge for faster execution at runtime. However, except in limited cases (Bresina, Drummond, & Kedar 1993; Mitchell 1990; Pearson *et al.* 1993), EBL and knowledge compilation techniques have not been used to compile *hierarchical execution* systems into reactive systems *concurrent* with execution. Such an approach would provide an additional form of speedup for hierarchical execution systems.

In this paper, we study the issues that arise when hierarchical execution systems are dynamically compiled into reactive systems. We assume the hierarchical system represents its knowledge as goals and operators in domain theories, while the reactive system uses rules. Rule-based systems are an attractive representation for reactive knowledge because it is possible to build new rules incrementally and add them to very large rule bases as the system is running (Doorenbos 1994). Given this representation, we consider how a

knowledge compilation process could work using EBL-like techniques. What we find is that a direct application of EBL to a straightforward domain theory can lead to complications in a dynamic domain. Specifically, the learning mechanism may create rules with conditions that test for the existence of features that can never co-occur in the environment. We call such conditions *non-contemporaneous constraints.*

This problem represents a severe stumbling block in applying knowledge compilation methods not only in interactive environments, but for hierarchically decomposed tasks in general. The aim of this paper is to convey an understanding of the problem and to investigate the space of potential solutions. Results from implementations of the most promising solutions demonstrate how the problem can be avoided. These results suggest that knowledge compilation mechanisms may apply to a much larger population of environments than previous work has suggested.

An Example Domain

To illustrate our discussion, we will present examples for a system that controls an airplane in a realistic flight simulator. This task can naturally be decomposed into a hierarchy of domain theories. At the highest level, the agent may generate decisions to fly particular flight plans. Execution of flight plans can be achieved by creating subgoals to arrive at certain locations at certain times. Further decomposition is possible until execution reaches the level of primitive commands. At this level, the flight controller issues commands that can be directly executed by the flight simulator, such as "push the throttle forward 4 units" or "move the stick left 10 units".

This decomposition is not a pure abstraction hierarchy as employed by abstraction planners (Sacerdoti 1974; Knoblock 1994) but a hierarchy of domain theories over features that can be aggregates and not just abstractions of features at lower levels. Higher levels can also include intermediate data structures that are not used at the primitive levels. This representation is very similar to that used in HTN planners (Erol, Hendler, & Nau 1994). The difference is that in this approach all the necessary control knowledge is part of the domain theory. The hierarchy of goals develops dynamically, in response to goals higher in the hierarchy as well as to changes in the environment, but the task is executed without search.

This style of knowledge representation has been used for both stick-level control of a simulated aircraft (Pearson *et al.* 1993) and higher-level control of a tactical flight simulator (Tambe *et al.* 1995) using 400 to 3500 rules and from five to ten levels of goals. We will focus on a single task from the flight domain: achieving a new heading. This task uses commands that control the roll, pitch, yaw, and speed of the aircraft. Table 1 contains a subset of a domain theory for this simplified flight controller.

```
1. IF  NotEqual(Heading, GoalHeading)
   THEN  CreateGoal(ACHIEVE-HEADING)
2. IF  Goal(ACHIEVE-HEADING)  AND
       LeftOf(Heading, GoalHeading)
   THEN  Execute(ROLL(right))
3. IF  Goal(ACHIEVE-HEADING)  AND
       RightOf(Heading, GoalHeading)
   THEN  Execute(ROLL(left))
4. IF  Goal(ACHIEVE-HEADING)  AND
       NotEqual(Roll, level)  AND
       Equal(Heading, GoalHeading)
   THEN  Execute(ROLL(level))
5. IF  Goal(ACHIEVE-HEADING)  AND
       Equal(Roll, level)     AND
       Equal(Heading, GoalHeading)
   THEN  DeleteGoal(ACHIEVE-HEADING)
```

Table 1: Simplified domain theory for achieving a goal heading.

Assume that the controller is flying due north and has just made a decision to turn due east, represented as a new goal heading. The difference between the current and goal headings leads to the creation of an `ACHIEVE-HEADING` goal by Rule 1. The execution of the `ROLL(right)` command then follows from Rule 2. With the plane in a roll, it begins turning, reflected by changing values for the current heading arriving through the input system. The turn continues until the current heading and goal heading are the same. Because the plane is in a roll when the goal heading is achieved, the agent cannot simply delete the `ACHIEVE-HEADING` goal. Instead, Rule 4 generates a command to return the aircraft to level flight. Once the aircraft has leveled off, Rule 5 terminates the `ACHIEVE-HEADING` goal. In this example, the knowledge for attaining a new heading under different conditions has been distributed across a number of subtasks in the hierarchy. Individual rules consist of conditions that are simple, easy to understand, and may combine in different ways to get different behaviors. For instance, `ROLL` commands can be invoked from other contexts, and the `ACHIEVE-HEADING` goal can be created for conditions other than the simple one here.

Compilation of Hierarchical Knowledge

Now we consider one knowledge compilation technique, EBL, for compiling hierarchical knowledge. EBL uses a *domain theory* to generate an *explanation* of why a *training instance* is an example of a *goal concept* (Mitchell, Kellar, & Kedar-Cabelli 1986). It then com-

piles the explanation into a representation that satisfies appropriate *operationality criteria*. In our case, an explanation is a trace of the reasoning generated using the domain theory for a given level in the hierarchy. The training instance is a set of features describing the state of the world; however, these features may change because the domain is dynamic. The goal concepts are any features or actions that are created for use in solving a higher-level problem. In our example, this includes just the primitive **ROLL** actions sent to the external environment (assumed to represent the highest level of the hierarchy). Finally, the operationality criterion specifies that the features included in a compiled rule should come from the representation used in the higher-level problem. In the example, everything but the **ACHIEVE-HEADING** goal is operational.

EBL compiles the reasoning within a goal by regressing from the generated action through the trace of reasoning, collecting operational tests. The collected tests and the action combine into a rule that can be used in place of a similar chain of reasoning in the future. Because a given goal can incrementally generate results for higher levels (such as multiple primitive control actions), multiple explanations and multiple rules can be constructed from a single goal.

Non-Contemporaneous Constraints

The rules in Table 1 represent a natural decomposition of the turning task. However, compiling this reasoning proves to be problematic. Difficulty arises because the current heading changes while the **ACHIEVE-HEADING** goal is active, leading it to be used for two different purposes over the course of a turn. The heading is used to initiate the turn, when it is not equal to the goal heading. Then, when the heading and goal heading are equal, the system ends the turn by rolling the aircraft back to level flight. Thus, the direct approach to generating an explanation for the final **ROLL(level)** command tests *two different* values of the **Heading**, one tested by Rule 1 to initiate the goal, and one by Rule 4 to level the plane:

```
IF    NotEqual(Heading, GoalHeading)  AND
      NotEqual(Roll, level)           AND
      Equal(Heading, GoalHeading)
THEN  Execute(ROLL(level))
```

These conditions are clearly inconsistent with each other. This is not a problem just because **Heading** changed. The value of **Roll** also changed (from being level to banking right), but because it was only tested at one point (by Rule 4), it did not cause a problem.

One way to describe the problem is that a persistent goal has been created based on transitory features in the environment. The **ACHIEVE-HEADING** goal persists across changes in heading. If a feature that leads to a goal changes over the course of the goal, and a new feature is later tested, then those features might enter into the compiled rule as non-contemporaneous constraints. This problem can occur whenever the performance system creates a persistent data structure based on features that change during the subgoal.

When EBL has been used to compile control knowledge within planning systems, even for dynamic domains, non-contemporaneous constraints have not arisen because the training instance is static during plan generation. For instance, both Chien et al (1991) and DeJong and Bennett (1995) describe approaches to planning and execution in which there is no interaction with the environment during planning and learning. However, when plan execution and EBL both occur while interacting with the environment, non-contemporaneous constraints can result.

Possible Approaches

Rather than simply dismiss explanation-based learning as inadequate, we now investigate possible ways to address non-contemporaneous constraints.

A-1: Include All Tested Features in the Training Instance. The simplest approach is to include in the training instance every feature that was tested during reasoning, regardless of whether it exists at learning time. In some cases, a useful rule may be learned. If the features are non-contemporaneous, however, the system will learn a rule containing non-contemporaneous constraints. This rule will not cause incorrect behavior, but it will never match. Additionally, a particular rule may be created repeatedly, wasting more time. Perhaps most importantly, an opportunity to learn something useful is lost.

A-2: Prune Rules with Non-contemporaneous Constraints. Another obvious approach is to forgo learning when a training instance includes features that are not present when the primitive is generated. This will avoid the detrimental effects of learning rules that contain non-contemporaneous constraints. However, it will also fail to learn useful rules when the missing features are not actually non-contemporaneous. A refinement of this approach is to delete rules with non-contemporaneous constraints. However, the recognition of non-contemporaneous constraints requires domain-dependent knowledge. For example, it may be physically impossible for a given plane to fly at a specific climb rate and pitch, although it can achieve both of them independently. This knowledge may not even be implicit in the system's knowledge base, so that additional knowledge is required for correct learning, but

not for correct performance.[1]

A-3: Restrict Training Instances to Contemporaneous Features. A similar approach is to base explanations only on items that exist at some single point in time. For instance, a system could be designed to include in the training instance only those features that are true in the environment when the primitive is issued (even if other contributing features, no longer present, were tested as well). This guarantees that non-contemporaneous constraints will not appear in the final rule. However, there is no guarantee that the resulting rule will be correct. If the final result is dependent upon the change of a value from an earlier one, then including only the final value in the learned rule will make it over-general. For example, consider the flight controller. If the system used only contemporaneous features at the time it generated `ROLL(level)`, the rule would look like this:

```
IF    NotEqual(Roll, level)  AND
      Equal(Heading, GoalHeading)
THEN  Execute(ROLL(level))
```

This rule could be over-general, because there may be times when the system should not level off. Bresina et al. (1993) describe an approach to avoiding over-general rules when the training instance is based upon features that existed at the time a chain of reasoning was initiated. Their approach is based upon a specific representational scheme that requires knowledge of temporal relationships in the domain.

A-4: Freeze Input During Execution. Another approach is to not allow the training instance to change over the course of generating a primitive, thus avoiding reasoning that would incorporate non-contemporaneous constraints. Because the training instance no longer changes with time, any learned rules will be guaranteed to be contemporaneous. However, this approach forces a loss in reactivity because the system will be unable to respond to changes during its reasoning, even if those changes are very important (for example, if a wind shear causes the aircraft to experience a sudden loss of altitude).

A-5: Deactivate Goals Following Any Change. A seemingly drastic approach is to force the system to start with a new training instance by removing all persistent goals every time there is a change. This guarantees that the execution will not use any goals that depend on outdated features of the environment. However, this means that reasoning must be restarted every time the external environment changes. This can be time consuming, although as more reactive rules are learned, the need to generate goals decreases. It is an empirical question whether costs of regeneration will be balanced by improvements from learning.

This approach may also require some extensions to the original domain theory to work properly. For instance, in the air controller example, the original representation contains a single rule for establishing the `ACHIEVE-HEADING` goal. After the system has begun the turn, the aircraft's changing heading will cause the goal to disappear. However, the goal will continue to regenerate through the application of Rule 1 until the plane's heading matches the goal heading. At this point, Rule 1 will no longer apply, the `ACHIEVE-HEADING` goal will not be created, and Rule 4 cannot fire to level the airplane. This exposes a gap in the system's domain theory, which can be patched with the following rule:

```
IF    Equal(Heading, GoalHeading)  AND
      NotEqual(Roll, level)
THEN  CreateGoal(ACHIEVE-HEADING)
```

This rule is not just a special case to eliminate non-contemporaneous constraints. Rather, it is real flight domain knowledge that was missing from the original formulation of the task. The original representation assumed one would always roll back to level flight at the end of an `ACHIEVE-HEADING` goal, so this knowledge was not needed explicitly. However, this fails to take into account the possibility of a new goal arising while the old goal is being achieved. Suppose the plane is in a roll at a particular heading and higher level knowledge determines that the aircraft should now maintain that current heading (e.g., due to some emergency condition or a change in overall plan). It then becomes necessary to level off from the turn. In the new hierarchy, this is accomplished as an implementation of the `ACHIEVE-HEADING` goal. Thus, the knowledge that must be added is a beneficial refinement of the system's domain theory.

A-6: Deactivate Dependent Goal Structure Following Any Change. A refinement of the previous approach is to deactivate the structures in a goal selectively, based on knowledge dependencies between intermediate results and the current external situation. In this approach, the goals are continually adjusted so they are consistent with the current training instance. As the training instance changes, intermediate results are deactivated, and new ones, consistent with current training instance and domain theory, are generated. Referring to our flight example, `ACHIEVE-HEADING` would only be deleted once, when the conditions of Rule 1 no longer hold in the environ-

[1] Some systems make such relationships explicit. For example, the ERE architecture (Bresina, Drummond, & Kedar 1993) includes *domain constraints*, which specify all possible co-occurrences.

ment. As with A-5, this approach may require additions to the domain theory.

This alternative presents a subtle complication. A domain theory may create persistent internal features as well as persistent goals. The rules that create goals will, by definition, test features that are higher in the task hierarchy than the goals they create. Goals will always be compiled at learning time into their constituent conditions. Non-goal persistent features, however, will be generated by rules that may contain goal tests (thus requiring further compilation), tests of other persistent features (possibly leading to non-contemporaneous constraints), and tests of non-persistent features (which should not be problematic). Thus, a complete knowledge-dependence mechanism must keep track of dependencies for both goals and other persistent features. The expectation is that this alternative will most intelligently use the architecture to track dependencies, but may also incur a large overhead in keeping track of all the relevant (and only the relevant) dependencies for goals and persistent features. This overhead may be severe enough to impact the reactivity of the system significantly.

A-7: Eliminate the Persistence of Goals and Features. Another approach is to eliminate persistence in the performance system altogether. Goals (and other features) will remain active only as long as the rules that generate them match. This makes non-contemporaneous constraints impossible, because there will never be any goals in the current chain of reasoning that depend on features that are no longer true. This requires no further overhead for tracking knowledge dependencies than an execution system would already use for matching rules. This approach has been demonstrated in Theo-Agent (Mitchell 1990). However, eliminating all persistence means that the intelligent agent can have no memory. Suppose, for example, that the flight controller received a radio message to come to a particular heading. The system would forget the new heading as soon as the radio message disappeared from the input system because there would be no way to store the information persistently.

A-8: Extend Training Instances (and Domain Theories) To Include Temporal Dependencies. A final approach is to supplement our domain theory representation with causal knowledge or knowledge of temporal dependencies. Explanation-based learning algorithms assume that a training instance includes all the features that are potentially relevant to generating the goal concept. In dynamic environments, this may include the temporal dependencies among the changes to the state used to generate the goal. If true, domain theories need to be extended to create and test temporal relationships and dependencies (such as Rule 4 testing that the `Heading` is now equal to the `Desired-Heading`, but it previously had some other value). This requires a history of previous events and would lead to explanations that explicitly test temporal dependencies. The result of knowledge compilation would be rules that are non-contemporaneous, but could still match because a memory of earlier behavior would always be available. A number of planning and learning approaches use representations that include temporal constructs (Bresina, Drummond, & Kedar 1993; DeJong & Bennett 1995; Gervasio 1994). However, because these relations must be represented explicitly (even when behavior could be generated without the extensions), the type of domains in which such approaches are applicable may be limited, as in A-7.

Approaches in Task Knowledge. Each of the proposed changes to the performance system imposes some constraints on how knowledge must be represented. In addition, given a general enough performance system, many of the proposed alternatives can be realized by adding general rules to the domain theory rather than actually changing the performance system. Thus, for comparison purposes, we propose alternatives TK-4, TK-5, TK-6, and TK-7 as knowledge-based implementations of alternatives A-4, A-5, A-6, and A-7, respectively. Besides the possible requirement for some new domain-general rules, the TK alternatives are conventions for representing the domain theory, whereas the A alternatives *require* similar changes to the domain theory. However, it is also possible that some of the TK alternatives will incur less overhead than the corresponding A alternatives because they can be tailored to particular domains and do not have to provide general solutions. A-8 is already an approach that is dependent on changing the domain theory in addition to the execution architecture.

Evaluating the Alternatives

We could explore all of these alternatives in depth, but a qualitative analysis indicates that some of the approaches are more promising than others. Thus, we have identified a few alternatives to implement based upon a number of priorities, presented in Table 2. For instance, because A-1, A-2 and A-3 do not lead to unproblematic learning, they have least priority for implementation. On the other hand, because A-5 and TK-5 (the "deactivation approaches") meet all our priorities, they were implemented first. Space prevents us from justifying each of these priorities in this paper. However, our analysis has shown they successfully identify the more promising alternatives.

Priority	A-1	A-2	A-3	A-4	A-5	A-6	A-7	A-8	TK-4	TK-5	TK-6	TK-7
1. Learns Correctly	X	X	X	√	√	√	√	√	√	√	√	√
2. Maintains Reactivity	-	-	-	X	√	√	√	√	X	√	√	√
3. Appears Tractable	-	-	-	-	√	√	√	√	-	√	√	√
4. Does Not Weaken Performance System	-	-	-	-	√	√	X	√	-	√	√	X
5. Does Not Require Domain Theory Reformulation	-	-	-	-	√	√	-	X	-	√	√	-
6. Minimizes Complexity	-	-	-	-	√	X	-	-	-	√	X	-

Table 2: Prioritization of the Non-Contemporaneous Constraints Alternatives

Comparing A-5 and TK-5 requires quantitative analysis. This analysis requires both a performance system amenable to the implementation of the solutions and an appropriate suite of tasks. Our execution strategy demands a performance system with these features: 1) interacts responsively to an external environment; 2) represents and executes hierarchical knowledge; 3) operationalizes experience using a knowledge compilation mechanism. The Soar architecture (Laird, Newell, & Rosenbloom 1987) meets these demands. Specifically, Soar has been applied to a number of different external domains using hierarchical, procedural knowledge (Laird & Rosenbloom 1990; Pearson *et al.* 1993; Tambe *et al.* 1995) and Soar's learning mechanism, chunking, has been described as a form of explanation-based generalization (Rosenbloom & Laird 1986).

For a task environment, we have developed the *dynamic blocks world*, a test bed (in the sense of Hanks, Pollack & Cohen 1993) that facilitates controlled experimentation while posing tasks that distill important properties of domains like flight control. Tasks in the test bed are similar to blocks world problems familiar in the planning literature. However, there are two key differences. First, actions are not internal. The agent generates primitive commands (e.g., "open the gripper", "move-up 2 steps"), which are then executed by a simulator. The agent's knowledge of a primitive action is thus separate from the actual implementation of the action. Second, the domain is dynamic. Actions take time to execute, there is simulated gravity, and exogenous events can be scheduled to move blocks, knock towers over, etc. Our goals in developing this test bed are both to compare solution alternatives under controlled, experimental conditions and to further understanding of the capabilities necessary for interaction in complex and dynamic environments.

Our quantitative results thus far are based on a simple tower-building task. When executing this task, the simplest approach supported by the Soar architecture learns rules with non-contemporaneous constraints (A-1). We use the performance of this system as a baseline to compare the other approaches. Both deactivation approaches have been implemented and applied to this task. Evaluation of the alternative approaches relies on the following three criteria.

Executes Task and Learns Correctly: Each of the deactivation alternatives successfully executes the tower-building task and learns rules without non-contemporaneous constraints. Our formulation of the domain theory for these problems was intended to cause the non-contemporaneous problem whenever possible by creating relatively deep hierarchies. Thus, although a relatively simple task, this result is a significant validation of the approaches.

Prefer Less Knowledge: Less knowledge is preferred because it reduces knowledge engineering demands. Acquiring this knowledge, regardless of the technique chosen, will take longer for greater knowledge requirements.

Three different types of knowledge are required in the two approaches. First is the domain theory itself. All approaches, including the baseline approach, require this knowledge. However, domain theories may need further refinement under the deactivation approaches, pointing out incompleteness in the domain theory. This proved true for the tower-building task, for which 10 additional rules were added to the baseline domain theory of 124 rules. Domain-independent knowledge of the approach is also required. In A-5, this knowledge is incorporated in the architecture itself. In TK-5, this knowledge must be added to the knowledge base. This task required the addition of 2 task-independent rules.

The third type of necessary knowledge is domain-dependent knowledge of the approach. Once again, in A-5 the solution is implemented in the architecture. Thus, goal deactivation can be performed independent of the domain. TK-5, on the other hand, required 7 domain-specific rules for deactivating goals. These rules are significant because they are the most difficult to engineer and/or acquire.

Based on this analysis, A-5 clearly requires less knowledge in comparison to TK-5. Questions for future work are to determine if the additional knowledge

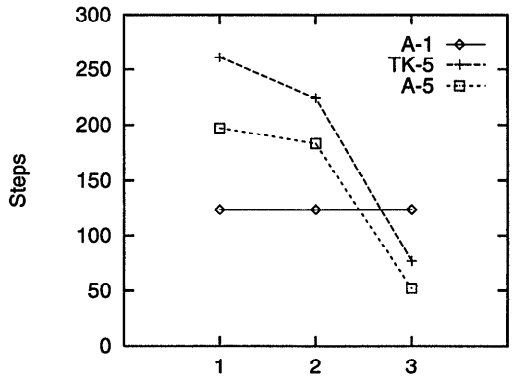

Figure 1: Execution steps of alternatives before (1), during (2), and after (3) learning.

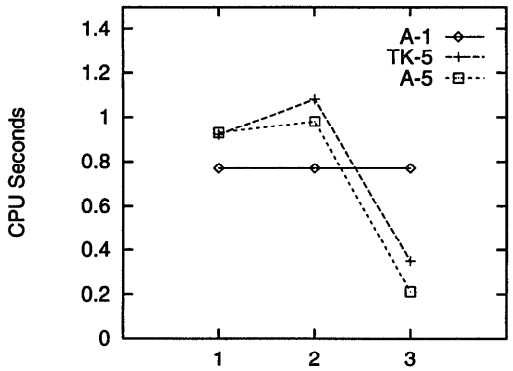

Figure 2: Total CPU time of alternatives before (1), during (2), and after (3) learning.

required by TK-5 is constant over the domain and how the additional knowledge requirements scale with an increasingly complex domain theory.

Task Performance: We measure performance according to three key characteristics: the number of reasoning steps required for executing a task, the performance speedup with learning, and CPU time.

Figure 1 plots the number of execution steps for the baseline, A-5, and TK-5 approaches as they change with learning. The deactivation approaches initially required about twice the number of execution steps as the baseline. However, after learning the solutions performed much better than the baseline with an average speedup factor of more than three. Additionally, the architectural approach was consistently better than the knowledge-based approach, although this was not unexpected. For the same type of approach, the architecture must bring the knowledge of the technique to bear in a TK variant while in the architectural variant this knowledge is embedded within the architecture and does not require additional execution steps.

One advantage of implementing the different alternatives in the same basic architecture is that CPU times can be readily compared. Figure 2 shows the total CPU times for the block stacking task. Although the effects of learning in this diagram and Figure 1 are similar, the increase in CPU time is proportionally less than the increase in execution steps, meaning that average cycle time (time per execution step) is reduced in the deactivation approaches. We are still investigating this effect and do not have the space to consider the issues here. However, this result does suggest that the total performance cost of the deactivation approaches may be less than that indicated by the increase in the number of execution steps.

Final Discussion

These results demonstrate that the implemented approaches are sufficient for overcoming the non-contemporaneous constraints problem for at least one simple task. Furthermore, A-5 executed the task and learned correctly while requiring only minimal additions to the domain theory and no domain-independent rules. Although performance before learning required more execution steps, A-5's performance improved significantly with learning. TK-5 required more execution steps than A-5 both before and after learning. These results have led us to consider A-5 as the primary solution approach.

In addition to our studies with A-5 and TK-5, a version of A-6 for a flight simulation domain has been implemented. We have not implemented a version of A-6 in the dynamic blocks world because the original architectural modifications were made to a now-defunct version of Soar. The A-6 approach turned out to be very difficult to implement, and led to a dramatically increased cycle time, due to the overhead of maintaining many knowledge dependencies. This experience combined with the implementation results for A-5 have led us to generate one additional alternative, which we plan to explore in the near future, as we briefly discuss here.

A-5 requires increased execution steps before learning has a chance to operationalize the domain theory, because it deactivates the entire goal structure any time there is a change in input. A-6 is the "smart" alternative, which retracts only the goals and persistent features that depend on the changing input, but this alternative is computationally expensive. A compromise approach is to track the dependencies of goals on changing input, but not expend the extensive effort required to track the dependencies of persistent features

that are not goals. This approach allows the architecture to eliminate non-contemporaneous constraints that arise from goals, but requires a domain-knowledge convention (as in TK-5) for other persistent features. The simplified tracking of dependencies through goals should decrease the computational overhead incurred by A-6, but should be less subject to environmental change than A-5. The final evaluation awaits implementation and empirical tests, but our experiences with implementations of A-5, A-6, and TK-5, suggest that this might be a worthwhile pursuit.

Although the alternative approaches have so far been applied mostly to simple tasks, the tasks have been designed within the test bed to capture the important properties of interaction in external domains. The results of our evaluation suggest that the problem of non-contemporaneous constraints, arising from the use of explanation-based knowledge compilation in external environments, while serious, is not debilitating. Further, several of the approaches presented appear to be appropriate strategies for performance and speed-up learning in external environments.

Acknowledgements

This research was supported under contract N00014-92-K-2015 from the Advanced Systems Technology Office of the Advanced Research Projects Agency and the Naval Research Laboratory, and contract N66001-95-C-6013 from the Advanced Systems Technology Office of ARPA and the Naval Command and Ocean Surveillance Center, RDT&E division.

References

Bresina, J.; Drummond, M.; and Kedar, S. 1993. Reactive, integrated systems pose new problems for machine learning. In Minton, S., ed., *Machine Learning Methods for Planning*. Morgan Kaufmann. chapter 6, 159–195.

Chien, S. A.; Gervasio, M. T.; and DeJong, G. F. 1991. On becoming decreasingly reactive: Learning to deliberate minimally. In *Proceedings of the Eighth International Workshop on Machine Learning*, 288–292.

DeJong, G., and Bennett, S. 1995. Extending classical planning to real-world execution with machine learning. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, 1153–1159.

DeJong, G., and Mooney, R. 1986. Explantion-based learning: An alternative view. *Machine Learning* 1(2):145–176.

Doorenbos, R. 1994. Combining left and right unlinking for matching a large number of learned rules. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*.

Erol, K.; Hendler, J.; and Nau, D. S. 1994. HTN planning: Complexity and expressivity. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, 1123–1128.

Georgeff, M., and Lansky, A. L. 1987. Reactive reasoning and planning. In *Proceedings of the National Conference on Artificial Intelligence*, 677–682.

Gervasio, M. T. 1994. An incremental learning approach for completeable planning. In *Proceedings of the Eleventh International Conference on Machine Learning*, 78–86.

Hanks, S.; Pollack, M.; and Cohen, P. R. 1993. Benchmarks, test beds, controlled experimentation and the design of agent architectures. *AI Magazine* 14:17–42.

Knoblock, C. A. 1994. Automatically generating abstractions for planning. *Artificial Intelligence* 68:243–302.

Laird, J. E., and Rosenbloom, P. S. 1990. Integrating execution, planning, and learning in Soar for external environments. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, 1022–1029.

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

Mitchell, T. M.; Kellar, R. M.; and Kedar-Cabelli, S. T. 1986. Explantion-based generalization: A unifying view. *Machine Learning* 1(1):47–80.

Mitchell, T. M. 1990. Becoming increasingly reactive. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, 1051–1058.

Pearson, D. J.; Huffman, S. B.; Willis, M. B.; Laird, J. E.; and Jones, R. M. 1993. A symbolic solution to intelligent real-time control. *Robotics and Autonomous Systems* 11:279–291.

Rosenbloom, P., and Laird, J. 1986. Mapping explanation-based generalization onto Soar. In *Proceedings of the National Conference on Artificial Intelligence*, 561–567.

Sacerdoti, E. D. 1974. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence* 5:115–135.

Tambe, M.; Johnson, W. L.; Jones, R. M.; Koss, F.; Laird, J. E.; Rosenbloom, P. S.; and Schwamb, K. 1995. Intelligent agents for interactive simulation environments. *AI Magazine* 16(1):15–39.