Ensuring Reasoning Consistency in Hierarchical Architectures

by

Robert E. Wray, III

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1998

ABSTRACT

Ensuring Reasoning Consistency in Hierarchical Architectures

by
Robert E. Wray, III

Chair: John E. Laird

Agents often dynamically decompose a task into a hierarchy of subtasks. Hierarchical task decomposition reduces the cost of knowledge design in comparison to non-hierarchical knowledge because knowledge is simplified and can be shared across multiple tasks. However, hierarchical decomposition does have limitations. In particular, hierarchical decomposition can lead to inconsistency in agent processing, resulting potentially in irrational behavior. Further, an agent must generate the decomposition hierarchy before reacting to an external stimulus. Thus, decomposition can also reduce agent responsiveness.

This thesis describes ways in which the limitations of hierarchical decomposition can be circumvented while maintaining inexpensive knowledge design and efficient agent processing. We introduce Goal-Oriented Heuristic Hierarchical Consistency (GOHHC), which eliminates across-level inconsistency. GOHHC computes logical dependencies in asserted knowledge between goals rather than individual assertions, thus avoiding the computational expense of maintaining dependencies for all assertions. Although this goal-oriented heuristic ensures consistency, it does sometime lead to unnecessary repetition in reasoning and delay in task execution. We show empirically that these drawbacks are inconsequential in execution domains. Thus, GOHHC provides an efficient guarantee of processing consistency in hierarchical architectures. Ensuring consistency also provides an architectural framework for unproblematic knowledge compilation in dynamic domains. Knowledge compilation can be used to cache hierarchical reasoning and thus avoid the delay in reaction necessitated by decomposition. An empirical investigation of compilation in hierarchical agents shows that compilation can lead to improvement in both overall performance and responsiveness, while maintaining the low design cost of hierarchical knowledge.

To Erma Hawkins ...
and promises made in childhood

Acknowledgements

I would like to thank John Laird for his wise guidance, patience, and great enthusiasm during my time at Michigan. John was always ready with encouragement when code suddenly failed to work, offered incisive ideas in those dark days of meandering about for a thesis topic, and never demanded more of me than he gave himself. I can only hope that I can apply all I have learned from John about being an effective mentor in my future work. Randy Jones was also an important mentor and friend. Randy provided a critical but supportive perspective on my research. He would gently remind me of the greater world when my work became too narrowly focused but also expressed faith in my ability to widen my perspective. I thank my fellow graduate students in the Soar research group at Michigan, including Mazin As-Sanie, Sayan Bhattacharyya, Ron Chong, Scott Huffman, Doug Pearson, Joe Phillips, Seth Rogers, Mike van Lent, Scott Wallace, and Chris Waterson. Our work together was always cooperative and collegial.

I also thank the other members of my committee, Ed Durfee and Bob Lindsay, for their challenge to consider the larger frameworks of my research. Any failure to meet that challenge is my own, but I greatly appreciate the advice and time they gave me and know that my work improved through their participation. Dan Koditschek also discussed the project with me on a number of occasions and I thank him for his participation.

This work would not have been possible without the people who helped in concretely in the development and evaluation of Goal-Oriented Heuristic Hierarchical Consistency. Scott Huffman, John Laird and Mark Portelli worked on the Soar implementation of Assumption Justification. Ron Chong implemented a precursor to GOHHC and unselfishly provided his advice, knowledge, and time throughout the course of the development of GOHHC. His help was indispensable. I also want to thank those who designed and implemented agents which I used for evaluation. Randy Jones, John Laird, and Frank Koss all contributed to μTacAir-Soar. Frank was especially patient and thorough in his help with the ModSAF simulator and the SMI. Sayan Bhattacharyya, Randy Jones, Doug Pearson, Peter Wiemer-Hastings, and other members of the Soar group at Michigan contributed to the development of the Dynamic Blocks World simulator.

I would also like to thank the staff of the Advanced Technologies Laboratory, who always made me feel like a partner in the lab rather than a lowly graduate student. Karen Coulter, Mike Hucka, Jeff Kopmanis, and Frank Koss were all keen providers of technical help throughout my time in the ATL and I'm certain I would not yet be graduating without their suggestions and solutions.

Finally, I want to thank all the people who helped make Ann Arbor a fun place to live as well as to work. Patrick Kenny coached the lab softball team and Pete Wurman organized the lab's racketball players. My office mates, Clare, Chris, Ella, Pete and Mike, provided many hours of conversation, diversion, laughter as well as their great intellects for the hard problems. Chad, Catherine and Christine reminded me often that music and friendship both endure, while grad school would someday pass away. And Karen's friendship, love and support provided balance, perspective and peace.

aspirat primo Fortuna labori.

Table of Contents

List of Tables

List of Figures

List of Appendices

Chapter 1

Introduction

Managers are not confronted with problems that are independent of each other, but with dynamic situations that consist of changing problems that interact with each other. I call such situations messes... Managers do not solve problems; they manage messes.

– R. Ackoff "The Future of Operations Research is Past," *Journal of the Operations Research Society*, 1979.

Henry has a problem. Worse, he doesn't even know he has a problem. Henry is a middle-manager at the ACME corporation. Profits are up and last week Henry's boss told him he could spend an extra $100K this fiscal year on computer upgrades. The deadline for fiscal year purchases is tomorrow so Henry has been working almost constantly since his boss told him about the windfall. Henry distributed some of the work to his own subordinates, indicating to each what they should spend of the allotment. Unbeknown to Henry, his boss has already decided to spend half of the $100K on another project. Henry's boss forgot to tell him that the amount of money he could spend had been halved. Henry is working hard on the task he was given, but the real task has changed. If tomorrow, he executes the purchase order, Henry will be in big trouble – or, at the very least, $50K over-budget.

Like human organizations, artificially intelligent computer agents are often organized via hierarchies. The complexity of running a big corporation is distributed among managers and workers with circumscribed responsibilities and specific expertise. Similarly, agents, autonomous computational artifacts that interact with external systems, can distribute their work among different processes or subtasks. An office robot might divide its tasks into processes concerned with physically moving in the environment and sensing the environment. Little interaction between the two processes is necessary except in particular circumstances, like obstacle avoidance. Importantly, this division of labor simplifies the design of agents, thus reducing their cost.

This thesis describes the ways in which communication between different subtasks in an agent hierarchy can breakdown, similar to the dilemma faced by Henry. Is it Henry's responsibility to ask his boss if the money is still available, or the boss's responsibility to keep track of what he has told Henry and inform him of any change? A failure to realize a change in circumstance results in inconsistency in the agent's – or organization's – knowledge. For instance, if we viewed the information content of the ACME corporation as the sum of the knowledge of all its workers, we would discover an inconsistency: Henry's boss believes Henry's division has \$50K to spend, while Henry believes he has \$100K. Inconsistency causes problems when an agent – or organization – acts on inconsistent knowledge. Henry spends \$50K he actually does not have.

We increasingly use agents to perform tasks for us. Robots build cars and explore the Martian landscape. Software agents filter email and search the world-wide web. Virtual agents fly aircraft and fight forest fires in real-time simulations. However, in order to give real responsibility to agents, agents must be able to perform their tasks reliably. Inconsistency in agent processing is one source of unreliability in agents. This thesis identifies problems in consistency that can arise in hierarchically organized agents, proposes some possible solutions to those problems that provide *guarantees* of consistency in processing, and empirically evaluates the most promising solutions. A guarantee of consistency makes agents more reliable, even in situations for which they were not specifically designed. Because design costs should be minimized, we prefer solutions that reduce development expense. Similarly, because agents face time pressure just like Henry, we prefer solutions that can guarantee consistency *efficiently*. In the following sections, we introduce the motivations and goals of this research in more detail.

1.1 Motivation: Limitations of Hierarchical Decomposition

The process of executing a task by dividing it into a series of different subtasks is called *hierarchical task decomposition*. Hierarchical task decomposition helps control environmental complexity. For example, an agent may consider high-level tasks such as "find a power source" or "fly to Miami" independent of low-level subtasks such as "go east 10 meters" or "turn to heading 135." The low-level tasks are chosen dynamically based on the currently chosen high level tasks and the current situation; thus the high-level task is progressively decomposed into smaller subtasks. Hierarchical decomposition can make it much easier to build agents for execution tasks because the current, active hierarchy provides a *context* that identifies important, relevant features of the external situation for subtasks in lower levels of the hierarchy.[1] However, hierarchical decomposition also has two important limitations: 1) an agent needs to ensure consistency in its hierarchical processing and 2) an agent must execute its tasks efficiently.

1.1.1 Inconsistency in Hierarchical Reasoning

Without careful design, it can be difficult to ensure consistent and efficient reasoning in agents employing hierarchical task decompositions. Inconsistency in the reasoning process can lead to irrationality in the agent's behavior. For example, imagine an exploration robot that makes a decision to travel to some distant destination based, in part, on its power reserves. Suppose in some different level of the hierarchy, the agent notices that its power reserves have failed. If this change is not communicated to the place in the hierarchy where the travel decision is made, the agent will continue to act as if its full power reserves were still available. Thus, the agent would be acting irrationally, in direct analogy with

[1]Throughout this thesis, we assume that the hierarchy grows from top to bottom; thus low-level tasks are in the lower levels of the hierarchy.

the introductory vignette of the ACME corporation. We have identified two specific sources of inconsistency that arise when hierarchical context changes as decomposition progresses: 1) failing to react to changes in the hierarchical context and 2) reacting overly aggressively; that is, reacting before a change in the hierarchy is fully communicated to all levels of the hierarchy.

Failure to React to Context Changes due to Persistence

An agent can fail to react to a change in the hierarchical context, leaving a local level "unsituated" with respect to the higher context. This failure may happen when the agent retains some previously derived assertion that depended upon assertions of knowledge no longer in the context. For example, as we described above, an agent might generate an assumption about a future state in a subtask (`future power: high`) that depends indirectly upon conditions in the current context (`battery: normal`). If the dependent assertion changes (`battery: failure`), then the agent must recognize that its assumption of future power depended on this value. Otherwise, its reasoning can become inconsistent. In the example, the agent might decide that its planned future state is achievable. However, it may make this decision based on the assumption that its power supply will be high at some later time, even though that assumption is no longer supported in the current state due to the battery failure.

Failure to react to a context change arises when the agent asserts knowledge in a local state that is *persistent.* A persistent assertion is one that is maintained in memory independent of the conditions that led to its creation (its justification). Persistence is necessary because it provides an agent with the ability to maintain internal state independent of an agent's perception of the current external situation. Thus, hierarchical architectures need to ensure reasoning consistency while also allowing persistent, internal state.

Overly Aggressive Reaction to Context Changes due to Multiple, Simultaneous Threads of Reasoning

An agent can also be too reactive, taking an external act or making an internal derivation before the context higher in the hierarchy is fully elaborated. For example, an agent might generate an irreversible motor command (`launch-rocket`) in a subtask while knowledge in the higher context was being applied that terminated the subtask (`abort-launch`). In this situation, the agent's assertions of knowledge in the local level can be inconsistent with the larger context, potentially leading to irrational behavior. An agent is overly reactive to a context change when the agent responds before all the ramifications of context change have been determined.

Overly aggressive reaction to a context change arises when the agent can pursue different threads of reasoning in different levels of the hierarchy simultaneously, without enforcing an ordering on the knowledge. This capability is important because it gives an agent the ability to bring all knowledge relevant to a given situation to bear simultaneously. When only a single thread of reasoning is supported, the results of applying an initial assertion may deactivate other applicable knowledge, and the agent is prevented from considering those additional choices. The agent must then depend upon arbitrary conflict resolution mechanisms for choosing one piece of knowledge over another. Multiple threads of reasoning allows flexibility in making a decision among mutually exclusive choices. The challenge for hierarchical architectures is to support multiple threads of reasoning while also ensuring reasoning consistency.

Previous Solutions

Previous methods for maintaining consistency across the hierarchy generally require additional agent knowledge. This knowledge is not necessarily required by the task but is necessary to manage consistency. For example, the example robot agent we mentioned above would need specific knowledge that told it to remove persistent assumptions about power when the battery failed. A knowledge designer must identify and formulate this *consistency knowledge* for all situations in which inconsistency could arise. This knowledge is specific to interactions between the information asserted in different subtasks, rather than simply concerned with executing the task. Thus, consistency knowledge can be very difficult to create, increasing the expense of agent development. Further, without a guarantee of consistency, these knowledge-based solutions can be brittle. In novel situations, unanticipated by the designer, or when using new architectural capabilities for which the agent knowledge was not specifically developed, incompleteness in knowledge-based solutions can lead to irrational behavior and agent failure.

1.1.2 Performance Degradation in Hierarchical Reasoning

The second important limitation of hierarchical decomposition is that the decomposition itself takes time to compute, potentially making the agent less efficient and responsive. Agents should be *efficient*, able to bring knowledge to bear on a complex problem using tractable processes, and *responsive*, able to react quickly to changes in the environment. However, in hierarchical task decomposition, the knowledge for performing a task has been distributed over the hierarchy. This expansion makes knowledge design easier and less costly, but may also lead to slower reaction times. If we compare an agent using a hierarchical decomposition to a hypothetical agent that performed the same task using a non-hierarchical or "flat" representation, the flat agent would most likely react faster to the environment because its knowledge is more specific to the external environment and avoids the overhead necessary for decomposition.[2] While it may be possible to re-engineer the hierarchical agent's knowledge to improve its performance, this re-design takes more effort, resulting in increased cost.

1.2 Goals and Organization of the Thesis

The overall goal of this thesis is to understand the possible limitations of hierarchical decomposition and to explore and evaluate some potential alternatives to these limitations while preserving the advantages of hierarchical knowledge. As introduced above, agents can use specific domain knowledge to avoid inconsistency. We hypothesize that

[2]This supposition assumes the cost of matching individual pieces of knowledge is the same in both systems. We assume that the procedure for matching automatically organizes flat knowledge for efficient matching, even if the flat agent requires considerably more knowledge (e.g., as measured in rules) than the hierarchical agent. We will explore this assumption in more detail in Chapter 6.

this knowledge can be replaced by procedures in the agent's processing substrate, its *architecture*. Wholly architectural solutions preserve inexpensive knowledge design and guarantee consistency even in situations not anticipated by the knowledge designer. Thus, a more specific goal of this research is to solve the consistency problems architecturally, guaranteeing appropriate reactivity within a level of the hierarchy and thus ensuring reasoning consistency.

In Chapter 2, we examine the advantages of hierarchical decomposition for execution environments. We first introduce an agent framework, then describe how hierarchical decomposition makes agent design less costly along some dimensions than the design of a non-hierarchical representation. We also provide specific examples of the limitations of hierarchical task decomposition outlined above. Chapter 2 emphasizes the assumptions underlying the research and problems motivating it.

Chapter 3 describes our approach to inconsistency due to persistence. We argue that previous approaches all fail along at least one of the key dimensions we have outlined here: knowledge design cost, efficiency, and responsiveness. Our approach extends truth maintenance techniques (Doyle, 1979) to capture the logical dependencies between local knowledge and the higher level context. However, efficiency concerns lead us to consider heuristic solutions to the determination of dependencies. Our heuristic guarantees reasoning consistency but may retract some assertions unnecessarily.

Chapter 4 focuses on our solution to inconsistency arising from multiple, simultaneous threads of reasoning. Our solution recognizes that a potential logical dependence exists between local threads of reasoning and threads of reasoning higher in the hierarchy. Our solutions prohibits local reasoning when the hierarchical context is changing. Again, we adopt a heuristic solution to simplify the computation of dependencies between reasoning in different levels of the hierarchy. Again, too, the solution ensures consistency but sometimes sequences reasoning unnecessarily.

In Chapter 5, we evaluate *Goal-Oriented Heuristic Hierarchical Consistency* (GOHHC), our total solution to the inconsistency problems. Having added this solution to the Soar architecture (Laird et al., 1987), we evaluate our solution along two critical dimensions. First, we compare the knowledge requirements for GOHHC agents to the knowledge requirements for agents using knowledge-based solutions for consistency. We expect that the knowledge representation requirements will be reduced in the new system, because the consistency knowledge has been incorporated in the architecture's processing. Second, because our particular architectural solution is heuristic in nature, and is potentially inefficient (in comparison to more domain-specific approaches), we compare the performance of GOHHC agents to agents using knowledge-based solutions. Results show that knowledge requirements are reduced using the architectural approach. Further, overall performance often improves under Goal-Oriented Heuristic Hierarchical Consistency because embedding general consistency knowledge in the architecture is less expensive than applying task-specific consistency knowledge.

We also hypothesize that agent performance can improve with experience by using knowledge compilation (Goel, 1991). Knowledge compilation has been used successfully in static domains and in dynamic domains in which the learning occurs "off-line" from the execution. For example, STRIPS (Fikes et al., 1972) compiled macro-operators over a static planning space even though these operators were then used to direct a robot in the external world. A few systems with on-line, analytic learning have been developed; however, these have been dependent upon specific representation schemes to avoid problems resulting from using knowledge compilation in a dynamic environment. These problems include generating rules that include features that never co-occur (the non-contemporaneous constraints problem (Wray et al., 1996)) and conflicts between compiled and original task knowledge (the knowledge contention problem). In Chapter 6, we show that Goal-Oriented Heuristic Hierarchical Consistency provides solutions to these learning problems, leading to agents whose performance improves with domain experience. Also, and as importantly, little or no modification to the agent's knowledge base is necessary with the addition of the compilation capability. Thus, knowledge engineering cost does not increase with the addition of compilation.

Finally, Chapter 7 summarizes the results of this work and outlines some potential directions for future research. Because the Soar architecture has also been used as a model of human cognition (Newell, 1990), we also briefly survey the impact Goal-Oriented Heuristic Hierarchical Consistency may have on this aspect of Soar.

1.3 Contributions

The primary contributions of this thesis are to:

- Provide an understanding of how inconsistency can arise due to persistence in hierarchical architectures. We introduce two new solutions, *Assumption Justification* and *Dynamic Hierarchical Justification* to avoid across-subtask inconsistency arising from persistence.

- Provide an understanding of how inconsistency can arise due to multiple threads of reasoning in a hierarchical architecture. We describe a solution, *Subtask-limited Reasoning*, that avoids inconsistency arising from multiple threads of reasoning in different levels of the hierarchy.

- Provide an empirical analysis of Goal-Oriented Heuristic Hierarchical Consistency, a combination of Dynamic Hierarchical Justification and Subtask-limited Reasoning.

- Introduce a new methodology allowing comparison of a new architecture to a baseline architecture by comparing relative behavior of agents implemented in the architectures.

- Provide an understanding of how inconsistency in an agent's processing can lead to specific problems in compilation. We show that Goal-Oriented Heuristic Hierarchical Consistency provides a guarantee of consistency sufficient for on-line compilation of subtask processing in the hierarchy.

Chapter 2

Hierarchical Decomposition for Execution Tasks

... thus is the poor agent despised.
– William Shakespeare

In the previous chapter, we outlined some of the advantages and limitations of hierarchical task decomposition for use in agent-based systems. In this chapter, we examine these advantages and limitations in greater detail. We focus in particular on the assumptions driving our approach to addressing the limitations of hierarchical task decomposition in execution domains.

2.1 A Framework for Agent Architecture

In this section, we introduce an initial framework for discussing agent architectures. There are currently many different implemented agent architectures and certainly more unexplored ways of building them. The 3T architecture is one instance of these many architectures, having been used primarily in mobile robot domains (Bonasso et al., 1997). The 3T architecture consists of three layers or "tiers," with a specific planner (Elsaesser and Slack, 1994), scheduler (Firby, 1987), and controller (Yu et al., 1994) comprising each tier. However, the tiers of the 3T architecture have also been discussed as a general functional decomposition of the capabilities necessary for building interactive agents, independent of the specific choice of implementation in each tier (Pryor, 1996). Throughout the dissertation, we adopt this abstract notion of an agent with three layers of processing as a general framework and explore the specific 3T agent framework below.

2.1.1 Three-tiered Agents

Figure 2.1 illustrates the three levels of processing in the 3T agent framework. In the lowest tier, the agent uses reactive skills to act in the external environment. In a physical system, such as a mobile robot, the reactive skills may be implemented as a collection of controllers for specific skills. These skills need not be decomposed into finer-grained skills nor composed with others at that level. For example, for a robot working in a business office environment, these skills might include obstacle avoidance, recognizing objects, and grasping some subset of those objects (e.g., recyclable cans) with a robotic arm. In a simulated system, the reactive layer simply executes a procedure that simulates

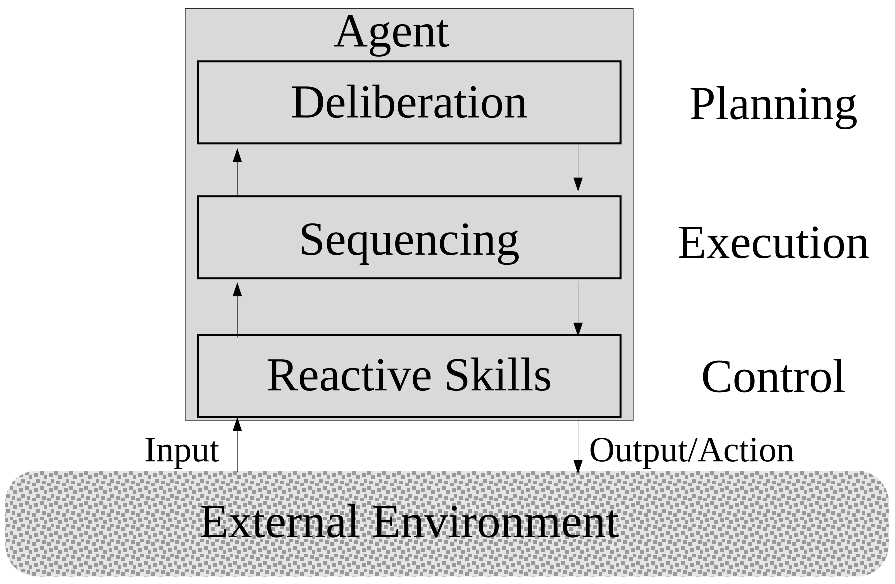


Figure 2.1: A basic, three-tiered, agent architecture. Adapted from (Bonasso et al., 1997).

the execution of the skill with an appropriate degree of fidelity.

The execution layer executes routine tasks by composing the skills in the reactive layer. We will call these routine tasks "behaviors."[1] For example, for a can collecting behavior, a robot will use many different reactive skills: recognizing a can, reaching and grasping, etc. Determining which of the reactive skills should be active (i.e., are salient to the current task) is the responsibility of the execution layer. The execution layer sequences the execution of the reactive skills by determining which should be active at any point in time. Rather than reason about the specific details of a skill like grasping, it can simply treat grasping as a discrete step, and then activate the appropriate skills, parameterized with task-relevant features (e.g., "grasp the can at (x, y)").

The execution layer simplifies the design of the control layer because interactions between the skills can be minimized. If we assume that the robot is always stationary when grasping a can, then the obstacle avoidance skill and grasping skills need never be active simultaneously. Therefore, any potential interaction between these skills in the design of the respective controllers for each skill can be ignored.

When some part of a task is no longer routine, perhaps due to a resource constraint ("collect all cans in the conference room before the executive board meeting at 2 P.M."), the deliberation or planning layer can be invoked to determine how to achieve the task. Just as the execution layer abstracted from the specifics of the control procedures to abstract steps like "grasping," the planning layer can reason abstractly about behaviors implemented in the execution layer. Rather than reason about the specifics of grasping

[1]We provide no formal distinction between a skill and behavior and there is none specified in the literature. In general, we use "skill" to represent procedures that lack agent goal context. For instance, balancing a bicycle is a skill because the activity can be performed with no reference to the agent's goal context. On the other hand, we would call an activity like "ride the bicycle to work" a behavior because it makes explicit reference to a goal context ("get to work").

cans or moving between rooms, the deliberative layer can regard `collect-cans`[2] as an atomic action which can be achieved by the execution layer. It can then develop a plan that ensures that the can collecting task for the conference room is completed by 2 P.M. This abstraction greatly simplifies the planning process (Sacerdoti, 1974; Knoblock, 1991).

2.2 Skill Activation in the Execution Level: Hierarchical Task Decomposition

Consider again our robot, on its task of collecting cans in the conference room. Assume that the robot has received an appropriately parameterized plan in which `collect-cans` is an implementable directive from the planning layer. The reactive skills that the agent will activate include grasping, reaching, obstacle avoidance, etc. How should the agent execute the can-collecting task in terms of these "primitives?" A great deal of reasoning must still be accomplished in the execution level, even if no deliberate planning is necessary. How will the robot go about searching the room for cans? Will it search while moving? Will it build a map of the room as it finds cans, to determine an optimal collection route, or simply opportunistically collect each individual can when it finds one? In this section, we introduce dynamic hierarchical task decomposition, which helps both a knowledge engineer and an agent manage the complexity of execution tasks.

Hierarchical task decomposition simplifies agent design by breaking a task down into smaller abstract tasks. The execution layer dynamically decomposes a high-level task like `collect-cans` into progressively smaller abstract tasks until a it can initiate a primitive action, such as "grasp" can be initiated. Because the agent decomposes the task dynamically, the specific decomposition can be tailored to the specific external situation. For example, rather than relying on a pre-encoded procedure for collecting cans, the dynamic decomposition allows the robot to pursue a variety of different "search" and "collect" strategies, determining which is most appropriate as execution progresses. Hierarchical task decomposition has been used in a great number of execution systems, including the Adaptive Intelligent Systems architecture (Hayes-Roth, 1990), ATLANTIS (Gat, 1991), Cypress (Wilkins et al., 1995), the Entropy Reduction Engine (Bresina et al., 1993), the New Millennium Remote Agent (Pell et al., 1996), the Procedural Reasoning System (Georgeff and Lansky, 1987; Lee et al., 1994), RAPS (Firby, 1987), Soar (Laird et al., 1987; Laird and Rosenbloom, 1990), and Theo (Mitchell, 1990; Mitchell et al., 1991), among others. We will examine some of the advantages of hierarchical task decomposition and a more concrete example in the following section.

As we discussed in Chapter 1, the goal in this research is to address the limitations of hierarchical task decomposition for the execution layer. However, two methodological problems immediately present themselves when considering how to approach an exploration of hierarchical decomposition for execution systems. First, although many architectures use hierarchical decomposition for plan execution, it may be difficult to develop a characterization that applies to all architectures. Each architecture supports decomposition in different ways. What is important in one architecture's implementation may be a trivial or even absent part of another. Second, agent design remains largely an engineering process, with significant effort involved not only in constructing theoretically-

[2] Appendix A provides an overview of the typographical conventions used throughout the dissertation.

motivated components but also solving technical problems in interfaces and control. We take these problems as constraints on the scope of our exploration and adopt the following methodological assumptions:[3]

Assumption 1: Experiment in simulation domains. The difficulty inherent in building physical agents has the potential to distract from the goal of addressing the limitations of hierarchical decomposition for plan execution systems. Further, the complexity of both domains and tasks make complete formalization of the characteristics of hierarchical decomposition intractable. Therefore, we will pursue an empirical characterization of hierarchical decomposition and limit these investigations to simulation domains where the engineering effort for the interface is more easily controlled. Support for this assumption includes (Hanks et al., 1993), which suggests that simulated "test beds" are a good choice for empirical studies because the experimenter has control over the underlying domain. As a further control, we will use simulation tasks designed independently of this research.[4]

Assumption 2: Focus on the execution level. The complexity of agent design for interactive, real domains makes simplification necessary. For our empirical investigations, we assume that agents have all the execution knowledge necessary for their tasks, and thus no planning layer is necessary. A number of successful execution systems have been built without a planning component (Bonasso et al., 1997; Georgeff and Lansky, 1987; Pearson et al., 1993; Tambe et al., 1995). We also assume that our agents can treat reactive skills as primitives by making them directly executable in simulation. Thus, no control layer will be used. We will be careful to point out when the specifics of some technique impacts processing in either the planning or the control layer but we will ignore these layers for the most part in our analysis and experiments.

Assumption 3: Focus on a particular implementation. Although we are interested in the general characteristics of hierarchical task decomposition, we will focus our empirical experimentation in one plan execution system: the Soar architecture (Laird et al., 1987). We originally became interested in the characteristics and problems of hierarchical task decomposition through the use of Soar in a number of different plan execution environments constrained by **Assumption 2** (Laird and Rosenbloom, 1990; Pearson et al., 1993; Tambe et al., 1995)). Given the complexity inherent in both domain and architecture, we decided to limit our actual implementations to this single architecture. Where possible, we will point out how some particular analysis applies to other architectures but we will not attempt to implement our approaches in different architectures.

We will consider further assumptions in later sections of this chapter. However, now we introduce a specific example of hierarchical decomposition.

[3]We will introduce additional assumptions later in this chapter. Appendix B provides a summary of the assumptions underlying this research.

[4]The specific methodology used in the thesis is discussed in Chapter 5.

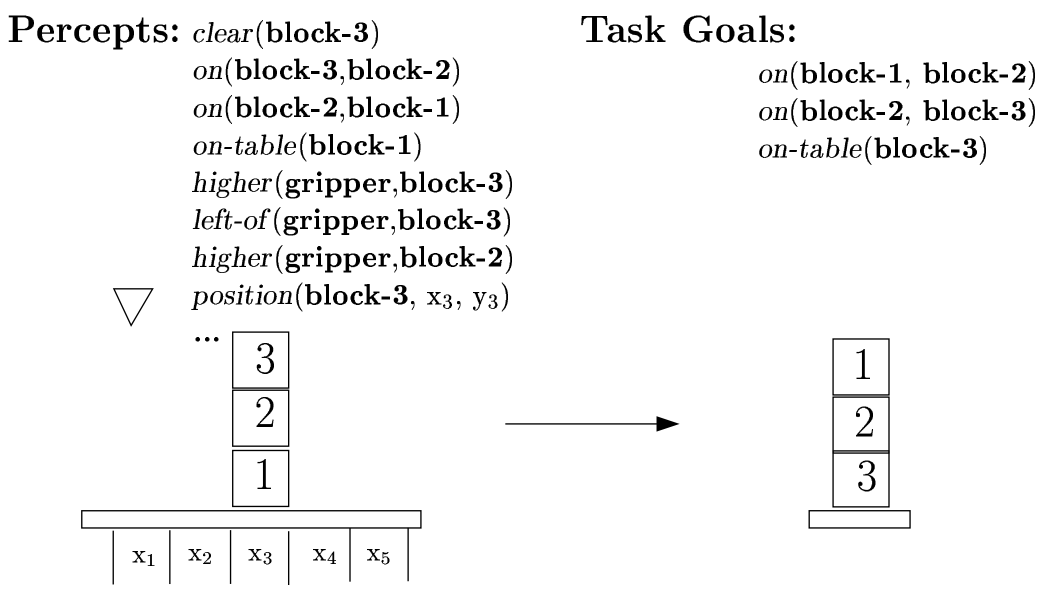


Figure 2.2: Building a tower in the Blocks World.

2.2.1 A Blocks World Example

We now turn to a specific example to illustrate hierarchical task decomposition. This example is based on the Blocks World domain familiar from planning (Chapman, 1987).[5] Rather than form a plan about how blocks should be moved, in this example an agent must actually move blocks in a simulated world to build towers, walls, etc. Figure 2.2 shows the agent's input as perceptual features of the world (position of blocks, relations between blocks, etc.). Also shown are the agent's task goals. In this case, the task goal is to build a particular tower (i.e., **block-1** on **block-2** on **block-3** on the table). Primitive actions are moving the gripper unit steps, and opening or closing the gripper.

Figure 2.3 shows the hierarchical decomposition we will use to illustrate the tower building task.[6] The agent decomposes the task of building a tower into two operators, one for putting blocks on the table, the other for stacking blocks. Each of these operators is further decomposed into operators for picking up and putting down a block. Both operators are generally necessary for either `put-on-table` or `stack` so lines extend from both of these highest level operators to each operator in the second level.

The operators `pick-up` and `put-down` both involve moving the gripper. Therefore, in the third level, the primitive operators for moving blocks are connected to both operators

[5]Throughout this dissertation, we will use the Blocks World for illustrative purposes, both because of its simplicity and the frequency of its use in illustration in artificial intelligence. We will also empirically test our solutions in a simulated Blocks World domain slightly more complex than the one presented in this chapter. However, because we are ultimately interested in more complex domains, we emphasize that the Blocks World is used primarily for illustration and simple, test bed comparisons. The primary empirical validation of our approaches will be made in a more dynamic and complex domain.

[6]Note that this is one of many possible ways in which this knowledge could have been formulated. It may be a worthwhile research question to develop criteria for developing good or "best" formulations of some task decomposition. While not a goal of this work, our approach to improving hierarchical decomposition in Chapter 3 will lead us to consider some decompositions as "better" than others.

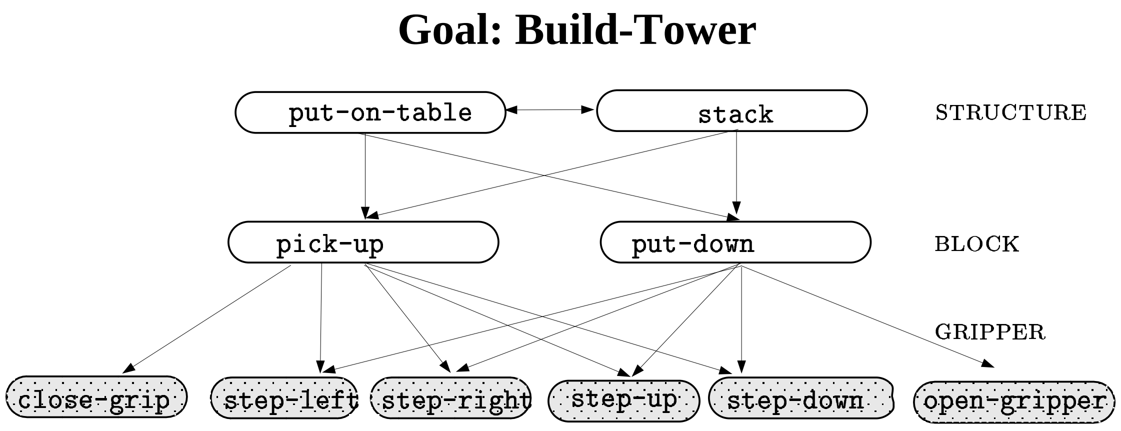


Figure 2.3: A simple hierarchical decomposition of the Blocks World domain. Primitive operators are shaded. Problem spaces are identified to the right.

in the second level. However, `pick-up` requires that the gripper actually grab the block; its decomposition thus includes the primitive operator `close-gripper`. On the other hand, putting down a block requires opening the gripper so `put-down` includes this primitive in its decomposition.

Figure 2.3 illustrates all the relationships between all the operators in the hierarchy. However, the actual decomposition at run-time is dynamic. Each subtask is chosen based on the current situation. As the external situation changes, either through the action of the agent (endogenous change) or through actions not directly caused by the agent (exogenous change), the choices of the current operators can be revised and updated. For example, when the agent is attempting to put a block on the table and the gripper does not currently hold the block, the `pick-up` operator will be activated. Once the gripper holds the block, `pick-up` is achieved. `Put-down` will then be activated to initiate the next step in the task.

Table 2.1 provides a few examples that illustrate how the external situation drives the choice of the current operators in the decomposition. We use abstract rules to illustrate the knowledge because it is easy to understand and similar to the actual implemented representation that we will use in later experiments. However, other representations, such as LISP procedures or PROLOG Horn clauses could be used as well. Rule 1 simply recognizes that the first step in building a tower is placing the bottom-most block so it creates `put-on-table`. In Rule 2, `pick-up` is created as an implementation of `put-on-table` if the block to move is *clear*. Finally, Rule 3 executes the `step-right` primitive for `pick-up` when the gripper is already higher than and to the left of the block to be moved. In the situation in Figure 2.2, Rule 1 would fire with **block-3** bound to x. Then, Rule 2 would fire, instantiating an operator to pick up **block-3**. Rule 3 would fire to step the gripper to the right. After the move terminated, Rule 3 would then fire again, because the conditions of this rule would be satisfied again. At this point, other rules would fire to close the gripper over the block and execution of the task would continue.

Each level of the decomposition in a hierarchical system corresponds to a problem space (Newell, 1980). A problem space is simply a space that the agent creates to search for a solution. Three different problem spaces are represented in this example. Each problem space represents a body of knowledge insulated from the others. For example,

```
1. IF     Task-Goal(Tower(x,y,z))
          Not(On-Table(x))
   THEN   CreateGoal(Put-On-Table(x))

2. IF     Goal(Put-On-Table(x))
          Clear(x)
   THEN   CreateGoal(Pick-Up(x))

3. IF     Goal(Pick-Up(x))
          Left-Of(Gripper, x)
          Higher(Gripper, x)
   THEN   Execute(Step(right, Gripper))
```

Table 2.1: Hierarchical Knowledge for Executing the Task in Figure 2.2

the possible operations in the STRUCTURE problem space are `put-on-table` and `stack`. Intelligent applications of these two operators will lead to the accomplishment of the task goal. However, *how* the agent should accomplish these tasks is left for other levels in the hierarchy. Similarly, the GRIPPER problem space concerns only the actions of the gripper – moving, opening and closing. Knowledge in the hierarchy above influences which action is chosen but the knowledge within the gripper space is largely independent of the goals higher in the hierarchy. At the level of maneuvering the gripper, the agent generally need not be concerned if it's working with blocks, balls, or an assembly line.

2.2.2 Hierarchical Architectures

In the simple rules in Table 2.1, the agent creates new subtask operators just like other facts in the knowledge base except that the subtask inferences are "goals." Some hierarchical systems treat goals as a distinct memory, and have special processing for the creation of goals (selection of operators), as opposed to just simple inference. One of the reasons for this special memory is that inferences can be associated with a goal. For example, if an agent were in the process of putting down a block on the table, it might need to remember the empty space on the table, where it plans to place the block.[7] The calculation of the empty space is part of the implementation of the `put-down` operator. Therefore, the agent stores this inference with the `put-down` goal. Conceptually, the agent's memory can be divided into a number of distinct pieces, each representing a currently active operator in the hierarchy, as illustrated in Figure 2.4. Thus, in the rules of Table 2.1, the CreateGoal action in the rules does not simply create a new symbol in memory, but rather a distinct memory.

One of the advantages of this memory structure is that the inferences are localized to dependent goals explicitly. In Figure 2.4, we assume the agent's architecture includes a set of processes for managing this hierarchical memory. When a goal is achieved (or abandoned), this hierarchy management module can efficiently remove all the inferences specific to the removed goal. Thus, if the `put-down` operator were interrupted, the empty

[7]We consider this computation in more detail in Section 2.2.4 and in Figure 2.6, page 19.

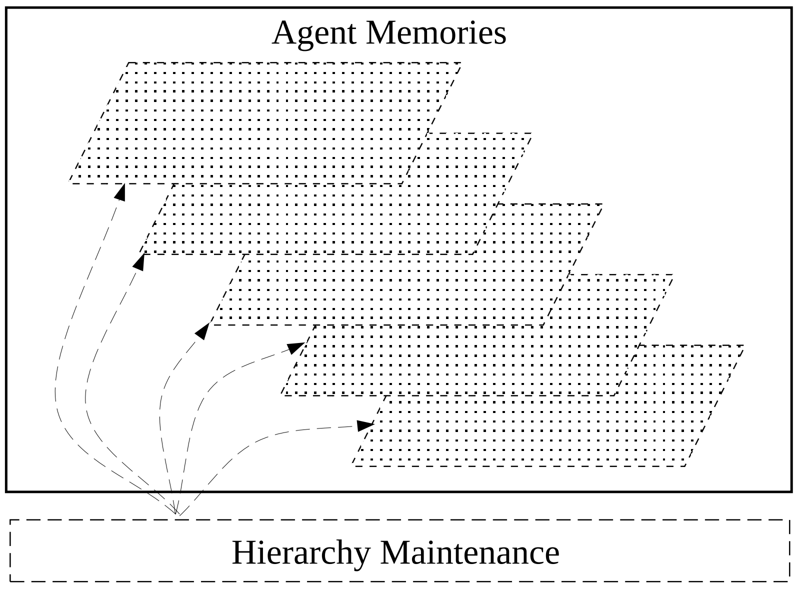


Figure 2.4: Hierarchical architectures provide special processes to manage hierarchical memories.

calculation could be automatically removed as well, without the overhead of a dependency calculation such as those used in truth maintenance systems (Doyle, 1979). We will have more to say about the capabilities and limitations of these processes in later chapters. However, for now, we define an *hierarchical architecture* as an agent architecture that divides the agent's memory into hierarchically-organized units and includes processes for managing the creation, revision, and removal of the individual memories.

2.2.3 Advantages of Hierarchical Task Decomposition for Execution

Our Blocks World example is simple enough that hierarchical decomposition, and special agent processes to manage the hierarchy, may seem unnecessary. Indeed, it would be relatively easy to design an agent for the Blocks World that used a non-hierarchical representation. For example, we could design an agent that simply mapped external world states to actions, in the spirit of a number of *reflexive* architectures (Agre and Chapman, 1987; Brooks, 1986; Schoppers, 1986). While later in this research we will consider tasks that would be difficult or impossible to implement in a reflexive agent architecture, we will use these non-hierarchical or "flat" agents for comparison purposes for two important reasons. First, flat agents will not experience the same problems in reasoning consistency that hierarchical agents do. The inconsistencies in hierarchical agents arise due to incongruities between the hierarchical memories, as we will explore below. Because the flat agents lack these hierarchical memories, they will not suffer from these sources of inconsistency. Second, a flat agent is generally able to respond more quickly to the external situation than a hierarchical agent, thus providing (potentially) better performance. Based on these characteristics, we will use these non-hierarchical agents as a potential alternative to hierarchical agents.

Hierarchical decomposition reduces the complexity of planning from exponential to linear in the size of the problem (Korf, 1987). This advantage has made hierarchical decomposition a well-researched and frequently-used technique in classical planning (Sacerdoti, 1974; Tate, 1976; Erol et al., 1994). We argue in this section that hierarchical task decomposition simplifies knowledge design in execution domains, in comparison to a flat representation. For both planning and execution, the gains provided by hierarchical decomposition are based on the following assumption.

Assumption 4: Focus on tasks that are (nearly) decomposable. Hierarchical task decomposition is based on the premise that tasks can be broken down into discrete units that have little interaction with other units. Simon (1969) argues that hierarchic structure is a natural consequence of evolutionary processes; a hierarchy simply provides stable, intermediate structure during the design process, offsetting increasing complexity. This intermediate structure gives hierarchical decomposition the advantages for execution we describe below. In this research, we assume that the tasks we will be addressing are *nearly decomposable.* A fully decomposable system has no interaction with other units in the system while a "nearly decomposable" system has limited interaction among different units. For example, in the Blocks World, the tasks of picking up a block and putting a block down have little interaction; each can be executed with no reference to the other. On the other hand, the `put-down` operator will need to know where to place the block, leading potentially to some interaction with an operator that determines the right space. The prevalence of planning and execution systems suggests this assumption is a reasonable one, although it does suggest that our methods and results will not apply to non-decomposable tasks.

For execution, rather than planning, the agent already has the knowledge needed to execute its tasks. It need not search through hypothetical states in search of the right sequence of actions as in planning. Instead, its chief responsibility is responding to the current situation by activating the knowledge most appropriate to the current situation. In other words, the execution system attempts to optimize *knowledge search* rather than *problem search* (Newell, 1990, pp. 98). Thus, the motivations for using hierarchical task decomposition in the execution layer are different than for using it in the planning layer. We outline some of these motivations below.

Natural Representation

A hierarchical decomposition of knowledge is a very natural way to represent the task. Even though the robot in the Blocks World cannot directly implement operators like `stack`, we can think of this goal/operator as a natural subtask in a tower-building task. Obviously, when the representation is strongly matched to the task, then the knowledge design will be easier than compared to a non-natural representation. In this latter case, representation requires a non-trivial transformation from the task.

Knowledge Sharing

With hierarchical decomposition, the same knowledge can be used for many different tasks. For example, consider the Blocks World we introduced in Figure 2.3. The knowledge specific to picking up and putting down blocks can be used for many different goals.

In this specific task, these operators are used for both the `stack` and `put-on-table` operators. However, the operators in the BLOCK problem space can be shared among any higher level operators in STRUCTURE or something else, as long as the task concerns moving blocks. Because the BLOCK operators are largely independent of these higher level tasks, the same knowledge for moving the block can be used for the different goals. Thus, sharing makes knowledge design easier because lower-level knowledge needs to be designed only once, rather than re-implemented for each higher-level goal.

Simplification of Knowledge

Hierarchical decomposition can also simplify the representation of individual pieces of knowledge. The currently active operators provide a local context for determining which knowledge to apply. For example, Rule 3 in Table 2.1 does not need to test that the block it is moving toward is *clear*. That test is implicit in its test of `pick-up`. Therefore, for an appropriately constructed decomposition, the knowledge specific to a subtask like "moving" need not examine the entire state space to determine what action should be taken.

Modularity

As we mentioned previously, decomposition provides a way of insulating different aspects of knowledge from each other. In the Blocks World, knowledge in the STRUCTURE problem space can be designed and implemented with little consideration as to how the operators in the lower levels would be realized. This *modularity* provides an advantage in knowledge design because if some aspect of the task changes, the change may affect only a small part of the task knowledge. For instance, suppose a new technology allowed grippers to move directly to an (x,y) coordinate rather than simply stepping in a single direction. In this case, every rule that generates a movement command (e.g., Rule 3 in Table 2.1) must be replaced. However, because movement is confined to the GRIPPER problem space, only knowledge in this problem space needs to be modified. A flat representation for the same task could require significantly more modification.

Another advantage of modularity is that the knowledge in different problem spaces can be developed independently. Because the knowledge is localized, many developers can be working on different parts of an agent's knowledge base simultaneously. Further, modularity facilitates re-use in different systems and replacement. Modularity is an important aspect of object-oriented methodologies for these reasons (Cox, 1986; Martin, 1993).

However, the modularity of the task knowledge is not complete. A dependence exists between different levels of the hierarchy. In particular, the higher levels of the hierarchy provide a context for lower-level problem-solving. If `pick-up` is chosen as the operator in the BLOCK problem space, then the implementation of `pick-up` in GRIPPER is impacted. The number of relevant operators in GRIPPER is reduced and the information associated with the instantiated `pick-up` operator provides parameterization to the operators in GRIPPER. The lack of complete modularity will be critical to our later work because the dependencies between levels will lead to potential inconsistency in the agent's knowledge base.

The result of these advantages is that hierarchical decomposition makes it much easier to build agents in comparison to agents with flat representations, especially as task

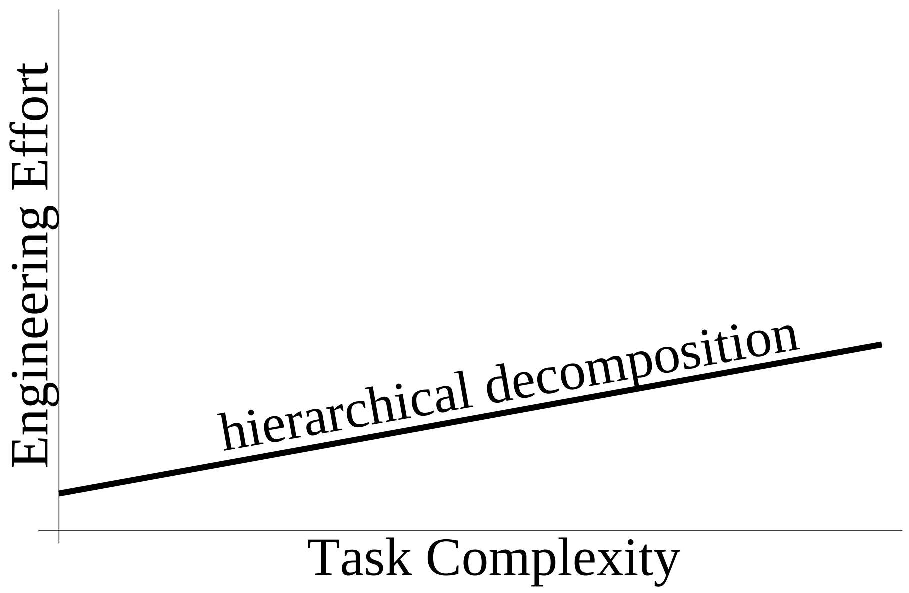


Figure 2.5: Hypothetical relationship between task complexity and engineering effort for hierarchically-decomposed tasks.

complexity increases. We use "task complexity" as a shorthand for a number of factors including the steps required to complete a task (number of primitive actions), the dynamic nature of a task, uncertainty in executing the task, resource bounds, and the capabilities necessary for a task (coordination, communication, fault-tolerance and meta-reasoning, etc.), among others. The Blocks World task we presented above can be considered relatively simple because it requires few steps, has no exogenous components, includes no explicit resources bounds, etc. However, we could also argue that the three-block Blocks World was more complex than a two-block world because it requires more steps, on average, to build the larger tower and all the other properties are the same between the two domains. The formal characterization of domain properties for software systems is a nascent and growing field of research in computer science (Prieto-Diaz and Arango, 1991). However, currently there is no method to quantitatively describe the complexity of domain and task; we therefore rely on relative, qualitative comparisons.

We hypothesize that the effort to build an hierarchical agent is bounded by a linear function of task complexity, as illustrated in Figure 2.5. This curve is based solely on intuitions provided by the advantages we outlined above and we will not try to empirically validate this hypothesis in the dissertation. In general, metrics used in software engineering provide only rough approximations of engineering effort, (e.g., person hours, source lines of code (Brooks, 1995)) and the complexity of tasks usually varies along many dimensions, as we described above. However, the important point from this diagram is that as the complexity of a task increases, so does the engineering effort necessary to implement a task. Further, because effort is closely correlated to cost, we assume that a driving motivation in agent design is to minimize effort and thus keep costs low. This observation leads us to another assumption:

Assumption 5: Engineering effort should be minimized. An important evaluation criterion in our work will be to ask how an approach or solution impacts the knowl-

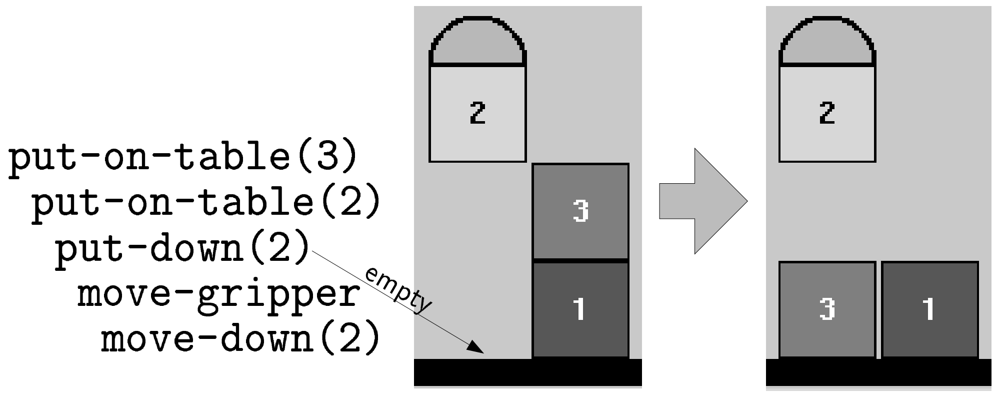


Figure 2.6: Knowledge-based methods can be brittle when the dynamics of the environment change.

edge design or engineering effort associated with building an agent for a particular task. Solutions that minimize effort with acceptable task performance will be preferred over solutions that require more engineering effort. Because engineering effort is difficult to measure precisely, our experimental methodology will rely on relative measures of engineering effort rather than absolute ones. For instance, we will measure the number of rules required for particular tasks and compare this measure of effort to those of other approaches for the same task. However, we will not compare the number of rules for one task versus another because rules may not be a good measure of effort across different tasks.

2.2.4 Limitations of Hierarchical Task Decomposition

Hierarchical decomposition also exhibits some disadvantages. Our perspective is to view these disadvantages as limitations, and then develop approaches that will allow us to minimize the impact of the limitations. In this section, we elaborate the two limitations outlined in Chapter 1. In the following section, we then present our approach to addressing the limitations.

Agents Require Knowledge to Maintain Consistency

A hierarchical decomposition provides a context for local decision-making that makes knowledge design simpler because the knowledge at a specific point in the hierarchy can ignore many of the specific details of the higher level hierarchy. However, this simplification also means that with respect to any level in the hierarchy, there are two contexts: the one represented by the external state, and the one represented internally that includes all previous knowledge in different levels of the hierarchy (active assertions). Because the agent relies on the internal context to simplify its knowledge, the internal and external states must be kept consistent or the agent can behave irrationally.

Consider, for example, the Blocks World situation shown in Figure 2.6. As we mentioned previously, in each hierarchical level, knowledge for the local goal can be asserted. In the diagram, the agent is placing **block-2** on the table, in order to later put **block-3** on the table and begin the goal tower. The `put-on-table(2)` operator does not concern itself with where **block-2** is placed on the table. However, `put-down`, which is currently implementing part of the `put-on-table` process, does need to make sure the space in which it is putting the block is actually empty. We assume it has computed that the space underneath the gripper is empty. This computation may not be directly observable – it may need to be derived from a number of other facts in the domain and stored in memory. Now, assume that some other agent suddenly places **block-3** underneath **block-2**, as shown in the figure. If the `put-down` operator has already determined that the space below is empty, then there is an inconsistency between the hierarchical context (which says that $(x, y) = (2, 1)$ is a good place to put a block) and the external world (which has changed such that $(2, 1)$ is no longer a valid place to place the block).

If the agent fails to recognize that **block-3** has moved, it will behave irrationally, attempting to put **block-2** into the same location occupied by **block-3**. This behavior is irrational, or not consistent with the agent's goals and knowledge, because we assume the agent has knowledge that indicates that blocks should not be placed in positions already occupied by other blocks. The inconsistency arises because the agent has failed to recognize the previously-derived assertion (empty) is no longer supported in the current situation. Failure to react to a context change arises when the agent makes assertions in a local state that are necessarily ungrounded. Such *persistence* is necessary because it provides the agent the ability to maintain internal state independent of an agent's current external situation (e.g., to maintain a sensor reading when the sensor is temporarily occluded) and to modify that internal representation non-monotonically.

Other sources of inconsistency include multiple, simultaneous threads of reasoning, or the ability to pursue many different directions of reasoning simultaneously. Suppose in our example that **block-3** is moved into its new position while the agent prepares to execute the command to move **block-2** into that space. Suppose also that the agent knows to remove the empty assertion from memory. In this example, the agent will be pursuing one thread of reasoning in the `put-down` subtask that will lead eventually to the recognition that the desired location is not empty while another thread of reasoning in a lower level of the hierarchy uses the existing empty assumption to initiate an action that will lead to failure. The inconsistency arises because the second (lower) thread of reasoning is actually dependent on the results of the first (higher) thread of reasoning.

In both of these examples, the inconsistency can be avoided by adding additional knowledge to the agent. For instance, to avoid inconsistency due to persistence, knowledge such as "when a block moves, recalculate the empty space and remove assertions that depend on the current empty space" would avoid the problem. Similarly, to avoid inconsistency from multiple threads of reasoning, knowledge could be added that recognized the dependence between two seemingly independent threads of reasoning ("only move a block into an empty space when certain that the empty calculation is not being updated."). In these examples, inconsistency is avoided by adding agent knowledge that manages the interactions between changes in the context.

However, this additional knowledge can be difficult (and thus expensive) to develop. First, this knowledge is not a natural part of the task representation. For example, this knowledge is motivated by the frame problem (McCarthy and Hayes, 1969). That is, it describes how to change previous assumptions as the world changes. In general, this knowledge is necessary in both flat and hierarchical systems. However, in the hierarchical system, this consistency knowledge is not limited to a single level of the hierarchy; instead, it must examine multiple levels of the hierarchy and determine what types of potential

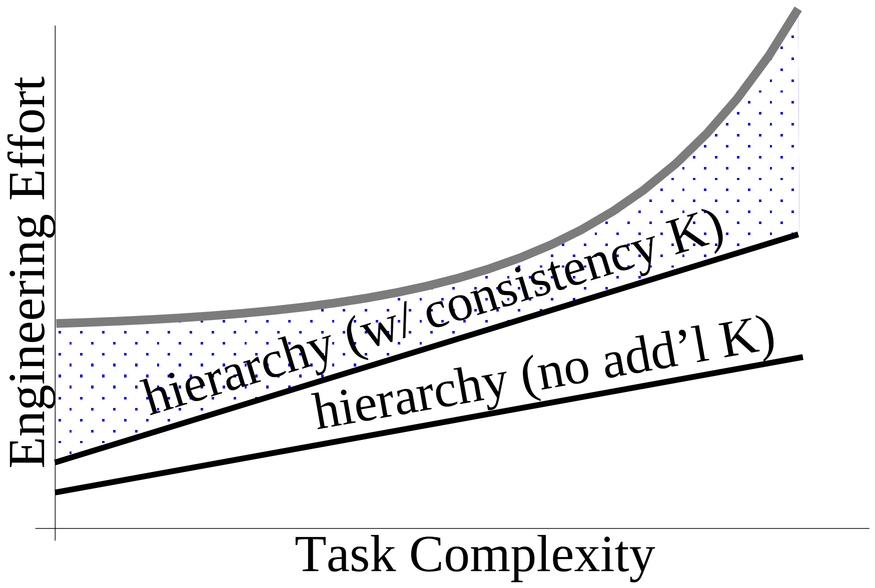


Figure 2.7: Relationship between task complexity and engineering effort.

interactions are present.

Knowledge that crosses levels of the hierarchy can defeat the advantages we presented earlier for hierarchical decomposition. Most importantly, this knowledge is not confined to a single level of the hierarchy and is thus not modular. Instead, the knowledge must consider the currently asserted knowledge in multiple levels of the hierarchy. Consistency knowledge makes the knowledge base less hierarchical and more "flat" because the agent has to know and reason about the interactions between different levels in the hierarchy. A lack of modularity makes the maintainability of software more difficult (McGregor and Sykes, 1992). Indeed, a number of presentations at a recent meeting of researchers using the Soar architecture (Schwamb, 1998) stressed that, for large systems, maintainability and re-use were difficult to achieve, even though Soar systems are organized around modular problem spaces. One source of this lack of maintainability is the consistency knowledge necessary in current version of the architecture.

Figure 2.7 represents the hypothesized effect this knowledge has on the engineering effort associated with employing an agent in increasingly complex domains. While we expect the knowledge effort to always be greater than that of an agent that did not need this knowledge, if the interactions between the different levels are combinatorial, the effort could increase substantially, perhaps even exponentially. Thus, one of the primary advantages of the hierarchical task decomposition is potentially defeated by the necessity of consistency knowledge.

Another disadvantage of representing consistency knowledge explicitly in the agent is that it provides no guarantees. An agent can fail if it does not have consistency knowledge for some situation. For instance, in the Blocks World example, it is reasonable to assume that blocks do not spontaneously move. Under this assumption, the knowledge designer probably would not have included consistency knowledge to handle this specific circumstance. This lack of a guarantee is a drawback because knowledge is often created with certain characteristics of the domain in mind. If these characteristics change, or turn out to be different than expected, the knowledge can easily fail. If blocks can move in the blocks domain, the agent with the knowledge presented here can no longer perform its task without error. Thus, an agent's robustness in a new situation is limited by the situations the knowledge designer explicitly anticipated.

Hierarchical Task Decomposition Incurs A Performance Cost

Another potential limitation of hierarchical execution is that the decomposition from the original task to primitive actions requires time. For example, the reflexive, flat agent we introduced earlier can generate a primitive action every reasoning cycle. However, in the hierarchical approach, each action is generated only after the decomposition is completed. In the worst case, an action at depth n in the hierarchy is generated only after n cycles, or $O(1)$ in the reflexive system *vs.* $O(n)$ in the hierarchical system. For time-critical behavior in complex environments, where the depth of the hierarchy could be arbitrarily large, such decomposition may not be feasible, especially as the speed with which the environment demands action approaches the cycle time of the underlying system.

2.3 Addressing the Limitations of Hierarchical Architectures

Can these limitations of hierarchical decomposition be circumvented? This dissertation describes efficient, general mechanisms that obviate the need for explicit, across-level, consistency knowledge in the agent. Our solution to the consistency problems, in turn, makes unproblematic compilation over the decomposition possible. We use compilation to improve performance when an agent encounters similar execution tasks in the future. The following sections outline these hypotheses in more detail. Chapter 3 describes our approach to the inconsistency due to persistence. Chapter 4 focuses on our solution to inconsistency arising from multiple threads of reasoning. Chapter 6 then explores the use of compilation to improve performance with experience.

2.3.1 Ensuring Consistency

We hypothesize that the "consistency knowledge" we described above can be embedded in an agent's underlying processes, rather than having to be included in the agent's domain knowledge. The resulting knowledge engineering costs will decrease for agent development in a specific domain, providing a much closer approximation to ideal engineering costs we described in Figure 2.5. The challenge in developing these solutions will be to find domain-independent approaches that are also efficient and thus do not degrade performance. We provide details of the specific methodological approaches in Chapters 3 and 4. Chapter 5 summarizes our empirical evaluation of these solutions.

Assumption 6: Development cost can be amortized over many applications.
An architectural solution presents a dilemma because it is generally easier to develop a knowledge-based solution for a specific task than it is to develop a general solution applicable to all tasks. Thus, this approach appears to be in conflict with **Assumption 5**. However, we assume that there will be many agents developed

using the architectural solution. Thus, the effort necessary for developing a general solution can be amortized over the development of all the agents using the architectural solution.

2.3.2 Improving Performance Through Compilation

We hypothesize that compilation can be used to improve agents that use hierarchical decomposition. Compilation is a speed-up learning mechanism that caches the results of reasoning (Anderson, 1987; Goel, 1991; Laird et al., 1986b). In an execution system, when a primitive skill is activated, the reasoning that led to the activation of the primitive may be compiled, resulting in a new piece of knowledge in the agent's knowledge base. In future similar situations, the newly compiled, more reactive knowledge recognizes the task situation as the one that previously led to the generation of a specific primitive and generates the primitive immediately, making further decomposition unnecessary. Thus, the agent pays the time-price for the decomposition to a primitive in a specific circumstance only one time.

Assumption 7: The world has regularities that make learning useful. We assume that the world has goal-relevant regularities (Laird et al., 1996). A block stacking robot regularly encounters situations in which it must stack blocks. The can-collecting robot regularly encounters situations in which it must move, search, and grasp cans. Regularities in the task domain provide the impetus to learn from experience because it is reasonable to assume similar situations will be encountered again.

The knowledge base remains manageable in this approach as well. The agent begins with a hierarchy of problem spaces, as introduced in the previous section. As the agent acts in its world, new rules are added to the knowledge base incrementally, based on the problem situation and the primitive action that was generated. Thus, actual experience guides which of a possibly combinatorial number of rules is compiled. Furthermore, when the agent lacks a reactive rule for a particular situation, it can always fall back to the hierarchical knowledge, generate a primitive, and then compile that experience to generate another new, reactive rule.

There are a number of issues that must be solved before compilation can be used in dynamic domains. We show in Chapter 6 that the solutions to the consistency problems we develop in Chapters 3 & 4 lead also to solutions to the non-contemporaneous constraints and knowledge contention problems for compilation. With these solutions, we provide empirical evidence in a number of task domains that shows compilation improves performance when an agent repeats a (similar) task following compilation. Significantly, this compilation occurs in a completely dynamic domain, and requires little additional knowledge engineering effort to enable the learning.

Chapter 3

Failing to React to Context Changes Due to Persistence

Do I contradict myself?
Very well then I contradict myself. – Walt Whitman

In previous chapters, we introduced two ways in which inconsistency could arise in a hierarchical system. In this chapter, we explore the first of these problems: what happens when an agent fails to respond to a change in its hierarchical context. Because our goal is to embed consistency knowledge in the agent's processing, we begin by discussing truth maintenance systems (TMS), which are used to maintain consistency in non-hierarchical architectures. We then introduce several examples to illustrate the limitations of standard truth maintenance techniques in hierarchical systems.

These limitations lead us to propose two new approaches.[1] *Assumption Justification* is the more conservative approach, ensuring minimal backtracking in execution, but is computationally expensive. On the other hand, *Dynamic Hierarchical Justification* is much less expensive to implement, but sometimes has to regenerate reasoning unnecessarily. In other words, Assumption Justification can lead to inefficiency in the architectural processing, while Dynamic Hierarchical Justification can cause inefficiency in the execution of a task. We examine these costs in detail and argue that Dynamic Hierarchical Justification is potentially a more efficient solution in execution domains, provided the implemented decompositions are nearly decomposable. Chapter 5 provides empirical evidence that shows dynamic hierarchical justification contributes to a reduction in engineering effort and improvements in performance in comparison to knowledge-based techniques.

[1]These solutions are also described in (Wray and Laird, 1998).

Inference Rules	Prior Assertions		New Assertion(s)
$A \wedge B \to D$	A, B	D	no contradiction (consistent)
	A, C	$\neg D$	no contradiction (consistent)
$C \to \neg D$	B, C	$\neg D$	no contradiction (consistent)
	A, B, C	$\neg D, D$	contradiction (inconsistent)

Table 3.1: A simple example of how inconsistency can arise in a knowledge base

3.1 Reasoning Consistency in Non-hierarchical Systems: Truth Maintenance

Suppose an agent has in its knowledge base the two simple inference rules represented in the left column of Table 3.1. The result of an inference is an *assertion*: a new fact that the agent can treat a being true in the world. Based on the contents of memory, the prior assertions, the agent can infer D, $\neg D$, both, or neither. As long as only two of the prior assertions are in memory, the resulting assertion added to the knowledge base does not contain a contradiction and the knowledge base remains consistent. However, if all three of the prior assertions are expressed simultaneously, both inference rules will apply, and the resulting knowledge base will contain a contradiction, as shown in the last row of Table 3.1. However, these rules are not necessarily inconsistent. For example, A and C may never occur simultaneously in the domain.

Inconsistency need not arise directly from inconsistencies in the asserted knowledge. Consider an agent that never asserts A, B, C simultaneously. Inconsistency in this agent's knowledge base can still arise through the agent's processing. For example, bringing knowledge to bear for execution (or problem solving, in general) takes time. What is true at one point in time may not be true in another. Suppose at some time t_1, the prior assertions are A, B. The first rule in Table 3.1 then applies, resulting in the data base of assertions: A, B, D. Now, suppose that at some later time, t_2, the world changes so that the A is retracted and C is asserted. The prior assertions are now B, C. The second rule in the table asserts, $\neg D$. The data base of assertions should now include only $B, C, \neg D$. The problem for the agent is to recognize that the conditions for assertion D are no longer supported, and withdraw that assertion from the data base.

Truth maintenance systems (TMS) (Doyle, 1979; McDermott, 1991; Forbus and deKleer, 1993) were developed to solve this problem in non-hierarchical systems. A simple representation of an agent utilizing a truth maintenance system is illustrated in Figure 3.1. Domain knowledge is used to create *assumptions*. An assumption is simply a fact that the inference engine wants to treat as being true in the world, even though such a belief is not necessarily entailed by the current state. For example, inputs are assumptions because the agent cannot usually determine their basis. Another example of an assumption is a hypothetical assertion. In the Blocks World, suppose that the agent's perception is limited to the column over which the gripper currently resides. When attempting to put a block into an empty space on the table, the agent might assume that a particular space, unsensed at the moment, was empty. Because it cannot derive this property in the current situation, it assumes it and acts as if it were true. The agent *enables* assumptions by informing the TMS that it should treat the assumption as held belief.

Importantly, in order to maintain reasoning consistency, domain knowledge must be formulated such that no enabled assumptions are contradictory. This requirement is the

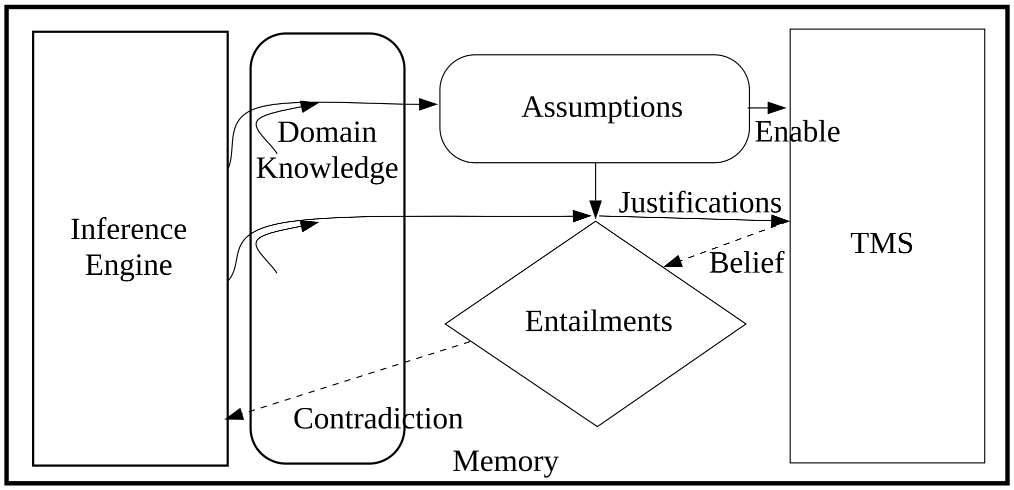


Figure 3.1: A non-hierarchical agent employing a truth maintenance system.

same one we reviewed in the previous chapter. If some assumption in the knowledge base can become inconsistent with the current input, then knowledge must recognize that situation and remove the assumption. The difference in a non-hierarchical agent is that there is only a single data base of assertions; the knowledge is not divided among different hierarchical memories. Thus, a flat agent requires no across-level consistency knowledge because there are no hierarchical levels for knowledge to cross.

The problem we introduced above is solved through a second class of assertions, *entailments*. The inference engine also uses its domain knowledge and assumptions to make these additional inferences. Entailments differ from assumptions because they can be justified by prior assertions (assumptions and entailments). In the Table 3.1 example, when D is added to the data base at t_1, it can be justified by A, B. The inference engine communicates justifications to the TMS. The agent continues making inferences and may non-monotonically change its assumptions. For instance, the world can change (now C is true) or an assumption can be disabled (e.g., if the gripper moved to a hypothetical empty space and discovered a block there). The TMS recognizes when an entailment is no longer justified, and the entailment is removed from the set of currently believed facts. Because D is justified by A, B, when A is removed, the TMS also removes D.[2] Or, when the agent removes the assumption that the Blocks World space was empty, entailments that suggested that space would be a good candidate location for a block are retracted. Thus, the agent relies on its processing via the TMS to maintain the consistency of entailments while the agent uses explicit domain knowledge to ensure consistency among the unjustified assumptions.

[2]Conceptually, D is removed. In many TMS implementations, a removed assertion is retained in memory but labeled as "OUT." Assertions labeled "IN" are currently justified, while those labeled "OUT" are not justified by the current assumptions.

3.1.1 The Necessity of Persistence

Naively, it may seem the consistency limitations we previously introduced can be solved solely through entailment and justification using a TMS. Indeed, Theo (Mitchell et al., 1991), a hierarchical plan execution system, does not experience the problem we describe in this chapter. However, Theo can only reason and act about what it can directly sense. The persistence of its internal inferences is limited by its sensors. If Theo needed to put out a fire in the kitchen and went into the basement to get a fire extinguisher, it would forget about the fire because it no longer sensed the fire.

Although there has been some research in structuring the external environment to provide persistent memory (Agre and Horswill, 1997), internal, persistent memory is necessary for many agent domains. Consider three reasons. First, as we saw above, agents sometimes need to "remember" an external situation or stimulus, even when that perception is no longer available. Second, some assertions may need to reflect hypothetical states. Such assertions are assumptions because a hypothetical inference cannot always be grounded in the current context. We saw an example of such an assertion in the previous section, when the agent assumed an *empty* space when unable to sense one directly. Third, sometimes the result of an inference changes one of the inputs to the inference (i.e., as in non-monotonic reasoning). As an example, consider the task of counting. Each newly counted item replaces the old value of the count. In many cases, entailment is sufficient for counting. However, as we will see in an example in the next section, when the determination of which items to count is itself a complex procedure involving many steps, persistence is necessary for this task.

Assumptions are the persistent features in an agent utilizing truth maintenance methods for consistency. Although most often assumptions reflect hypothetical reasoning about the world (hence "assumptions"), assumptions can also be utilized for all the functions we described above. We will see further use of assumptions as persistent memory in the following section, where we begin to explore the problems arising in reasoning when persistent assumptions are utilized in a hierarchical system.

3.2 Failing to Respond to Changes in Context

In the Blocks World example we presented in Chapter 2, we described inconsistency arising when a block moved into a space the agent has formerly calculated to be empty. In this case, if the relation empty needed to be calculated as an assumption, then the system would experience the problem as we described. However, one could also potentially design an agent such that the calculation of empty relation could be made an entailment. Truth maintenance systems are sufficient for maintaining the consistency of all entailments, and thus making this change would solve the problem in the Blocks World. However, as we discussed above, many inferences cannot be fully justified. In this section, we describe the role of truth maintenance in hierarchical agents. Problems occur in these agents when the agent fails to respond to a change in the hierarchical context that leaves a local, persistent assumption inconsistent with the hierarchical context. In order to illustrate the problem more concretely, we also consider an example from a more complex domain.

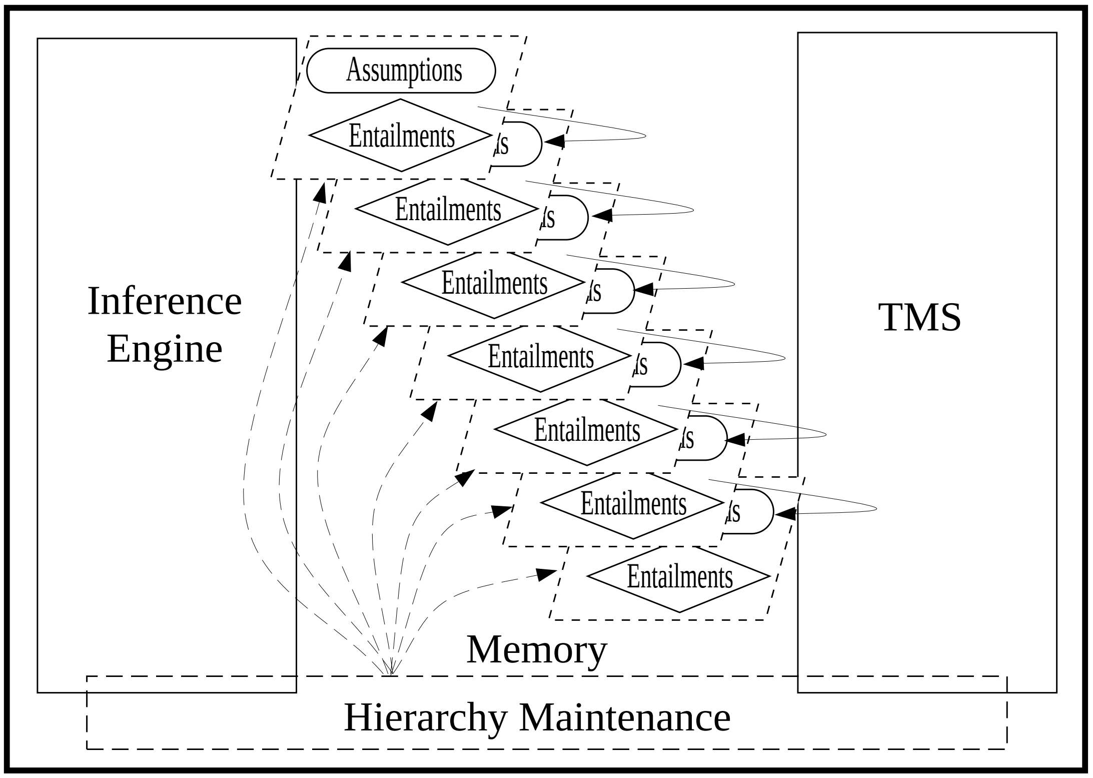


Figure 3.2: A hierarchical agent. Assumptions and entailments are localized to subtasks in the hierarchy. The hierarchy maintenance module determines when new subtasks should be created and when old subtasks should be retracted/deleted.

3.2.1 Assumptions in Hierarchical Agents

The basic agent framework we introduced in Figure 3.1 can be extended to hierarchical architectures, as shown in Figure 3.2. In such an agent, the inference engine and TMS are chiefly identical to the non-hierarchical agent. For convenience, the specific actions of the inference engine and TMS are not represented, although the functions are the same as in Figure 3.1.

When the agent initiates a new subtask, it creates a new level that will contain assumptions and entailments local to the subtask, just as we described in Figure 2.4 in Chapter 2. Figure 3.2 includes a new component, "hierarchy maintenance," which is responsible for creating and retracting levels when subtasks begin and terminate. The diagram represents hierarchy maintenance by the fan of dotted, arrowed lines. Implementations differ as to whether the maintenance is mediated by domain knowledge, TMS-like mechanisms, or other agent processes.

Within a specific level, reason maintenance can go on as before. The agent makes and deletes assumptions using domain knowledge. The TMS asserts and retracts entailments, based on the current assumptions. However, the hierarchical structure adds a significant complication to the creation of assumptions. Assumptions in a flat system are not usually dependent on other assertions. However, in a hierarchical system, the assumptions at one level can be dependent on the entailments and assumptions in higher levels of the

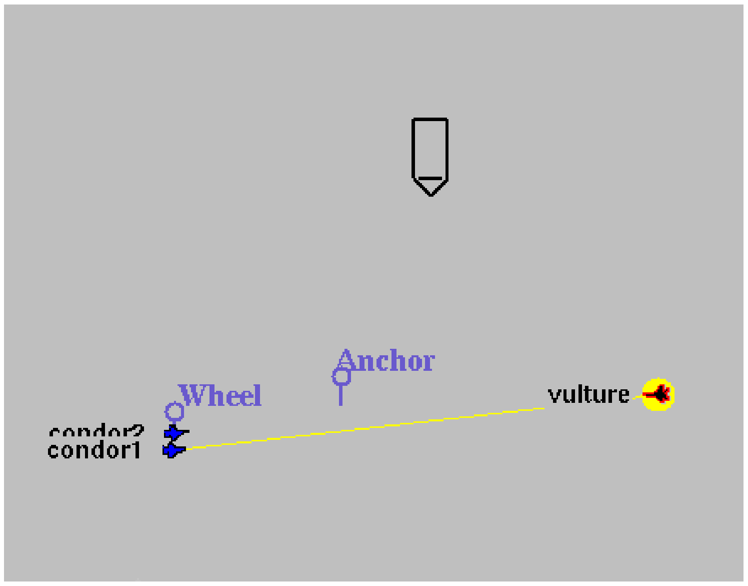


Figure 3.3: Intercepting an enemy plane in the TacAir-Soar simulator. The two planes to the left (Condor 1 and Condor 2) are intercepting the enemy plane on the right (Vulture). The line extending from Condor 1 to Vulture represents the path of a missile launched against the enemy aircraft. The circle around Vulture shows a successful missile hit. After completing this intercept, the Condor agents will return to a patrol anchored on the point labeled *Wheel* in the diagram.

hierarchy. The higher levels of the hierarchy form a “context” for problem solving in the local subtask and the subtask can rely on previous computations in this context rather than re-deriving the computations locally. For instance, when searching for an empty space in the Blocks World, the agent can use the current position of the gripper (computed in a higher level of the hierarchy) as a starting point for the search. The information dependence is illustrated in Figure 3.2 by the solid, arrowed lines extending from one level to the assumptions of the next level.

Changes in higher levels of the hierarchy may invalidate the assumptions of lower levels. Further, for execution agents embedded in dynamic, exogenous domains, the context will potentially change almost continuously. The changing context is not problematic for entailments because TMS will retract any entailment not justified in the new context. However, the assumptions of one level must be managed to ensure that they are consistent with the current situation as defined by the assumptions, entailments, and sensor data of higher levels. If they are allowed to persist independently of context changes, the reasoning in a subtask can become irrelevant to the actual tasks being pursued, leading to potentially irrational behavior. When an agent retains a local assumption even though the assertions in the levels above the local level (including perceptual input) are no longer consistent with that assumption, we say that an the agent has failed to respond to changes in its context.

- Patrol
  - Intercept enemy plane(s)
    - Attack (defensive)
      - Achieve proximity
        - Turn to heading

Figure 3.4: Decomposition of behavior into subtasks.

3.2.2 Example: TacAir-Soar

In order to understand how inconsistency arises due to persistence in hierarchical architectures, we now introduce TacAir-Soar (Tambe et al., 1995), an execution system that pilots virtual military aircraft in a real-time computer simulation of tactical combat. TacAir-Soar is a complex system, using over 450 operators in a hierarchical task decomposition that sometimes reaches a depth of greater than 10. In TacAir-Soar, each agent can have one of several different mission roles, among them attacking a ground target, flying a patrol mission, and acting as a partner or "wing" to some other agent's "lead."

We will concentrate on a pair of planes on patrol, which have been given specific directions to engage enemy aircraft entering their patrol area. Figure 3.3 shows an example from a simulation in which two agents (Condor 1 and Condor 2) have successfully attacked an enemy agent (Vulture). The line extending from Condor 1 to Vulture represents a missile launched by Condor 1. Having destroyed the enemy agent, the agents will return to their patrol around the point labeled "Wheel" in the diagram.

When enemy aircraft enter the patrol area, the lead agent decides to initiate an `intercept` of the enemy aircraft. The agent makes a number of calculations to determine the best course to take in attacking the enemy agent(s). For example, the simplified decomposition shown in Figure 3.4 shows that the complex task of task of intercepting the enemy aircraft has been decomposed into a relatively simple decision to turn the agent's aircraft to a specific heading. The agent is turning to this heading in order to get close enough to the enemy agent (`achieve-proximity`) to launch an attack.

In this example, we assume that there are three different kinds of attack that can be chosen for an `intercept`. The first tactic (scare) is to engage and attempt to scare away the enemy planes without using deadly force. This tactic is selected independently of the number of planes when the rules of engagement specify that deadly force should not be used. The second tactic (offensive attack) can be used when deadly force is allowed. It is appropriate when friendly planes outnumber or equal enemy planes. The third tactic (defensive attack) is used when enemy planes outnumber friendly planes and deadly force is permitted.

The determination of which tactic to pursue requires counting the current aircraft in the area when deadly force is permitted. Figure 3.5 shows the subtasks necessary for this count.[3] The agent must count the relevant enemy and friendly planes. We assume that determining a plane's "side" and its relevance to the count is sufficiently complex that entailment of the count is not possible. Thus, counting is necessarily non-monotonic, as we described above. The agent determines that enemy planes outnumber friendly ones

[3]This task could be decomposed many different ways. We chose this specific decomposition to illustrate issues in reasoning maintenance.

Patol
    Intercept enemy plane(s)
        Count enemy planes
        Count friendly planes
        Attack (defensive)
            Achieve proximity
                ....

Figure 3.5: Trace of behavior leading to intercept tactic in TacAir-Soar.

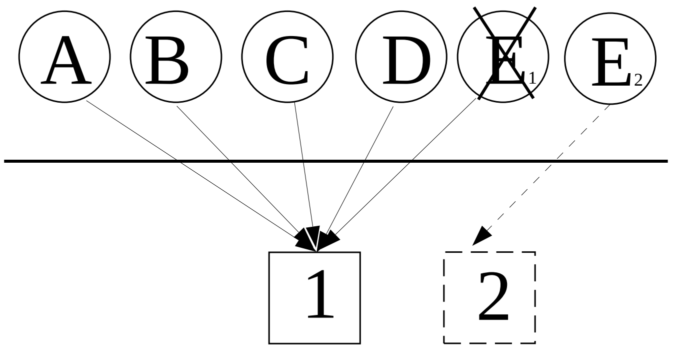


Figure 3.6: Inconsistency due to persistence. Assumptions are represented as squares, entailments as circles. The horizontal line represents a hierarchical task/subtask relationship between the assertions above the line and the ones below.

and the agent then selects `defensive-attack`, leading to further decomposition.

Given this scenario, what happens if an enemy plane suddenly turns to flee, thus reducing the actual count of relevant enemy planes by one? The count maintained by the agent is now invalid. Standard TMS reasoning is insufficient to retract the count, because the count assertion is an assumption, not an entailment. If the actual number of enemy and friendly planes is now equal, then the agent should switch its tactic to offensive attack. Continuing the defensive attack is not consistent with the agent's knowledge. Additionally, other "friendly" agents participating in the attack may base their behavior on the expectation that the agent is pursuing an offensive attack. Thus the agent needs to recognize the inconsistency and remove the existing count.

We can now present an abstract definition of the problem, as shown in Figure 3.6. In the diagram, assumptions are represented as squares, entailments as circles. The horizontal line represents a hierarchical relationship between the assertions above the line and the ones below. Thus, in the diagram, **A**, **B**, **C**, **D**, and $\mathbf{E}_1$ represent the hierarchical context for processing in the lower subtask. The arrowed lines represent logical dependence in the creation of an assertion or entailment. Thus, in this diagram, assumption **1** was created from entailments **A**, **B**, **C**, **D**, and $\mathbf{E}_1$.

Now, suppose the world changes so that $\mathbf{E}_1$ is retracted from memory and $\mathbf{E}_2$ is asserted. **1** remains in memory because it is an unjustified assumption, as we described above. If $\mathbf{E}_2$ would not also lead to **1** (e.g., it could lead to some new assumption **2**, as shown by the dotted line and box), then the subtask in which **1** exists is no longer consistent with the higher level context. Whether or not this memory inconsistency results in a behavior inconsistency is dependent on the use of assumption **1** in later reasoning.

3.3 Potential Solutions

How can an agent utilize persistent assumptions and simultaneously avoid inconsistencies that lead to irrational behavior? In this section, we review a number of potential approaches. This review is not intended to represent all possible approaches but rather a survey of different points in the space of all solutions.

How should we compare and evaluate different approaches? For now, we consider four qualitative evaluation criteria. In later experiments, some of these criteria will be expanded to allow us to make quantitative comparisons among different approaches.

Removes "cross-level" consistency knowledge. Assumption 5, in Chapter 2, described our desire to reduce engineering cost in knowledge design by eliminating the need for consistency knowledge that "crossed" hierarchical levels. Therefore, solutions that eliminate the need for this class of agent knowledge are preferred in order to reduce the overall cost of building agents.

Solves problem efficiently. Performance is a critical dimension of agent behavior. As we consider new additions to the agent's architectural processing, we will evaluate the impact these new processes have on the agent's performance.

Remains responsive to the external world. In addition to overall efficiency, an agent's performance is also impacted by how responsive it is to the external environment.

Retains nonmonotonic reasoning in subtasks. As we saw above, nonmonotonic reasoning is an important capability for agents acting in dynamic domains. Nonmonotonic reasoning should occur throughout the hierarchy, in subtasks appropriate to the reasoning.

3.3.1 Knowledge-based Assumption Consistency

An agent with a flat representation requires domain knowledge to ensure that its assumptions remain consistent. The same approach can be used to ensure reasoning consistency in a hierarchical agent by formulating explicit domain knowledge that recognizes potential inconsistencies and responds by removing assumptions. We call this approach *knowledge-based assumption consistency* (KBAC). For example, the Entropy Reduction Engine (ERE) (Bresina et al., 1993) relies on knowledge-based assumption consistency. ERE requires *domain constraints*, or knowledge that describes the physics of the task domain. Domain constraints identify impossible conditions. For instance, a domain constraint could indicate that an agent cannot occupy two different physical locations simultaneously.

In ERE, domain constraints are specifically used "to maintain consistency in the current world model state during execution" (Bresina et al., 1993, pp. 166). However, we believe that many other architectures use KBAC as well (either solely, or in conjunction with other methods we describe below). The use of KBAC explains why this problem has not been previously examined. In one sense, the knowledge is simply knowledge that must be added to the system to achieve behavior. Except in cases in which the knowledge itself was the focus of the research, researchers have generally ignored the specifics of their agent's knowledge (ERE being a notable exception).

KBAC will always be necessary to maintain consistency among the assumptions within a level of the hierarchy. However, as we suggested in the previous chapter, the requirement to represent explicitly the interactions leading to inconsistency *throughout the hierarchy* can add significant cost to agent development. With KBAC, a knowledge engineer must not only specify the conditions under which an assumption is asserted but also *all* the conditions under which it must be removed. In the TacAir-Soar interception example, the agent requires knowledge that disables all assumptions that depend upon the number of enemy airplanes when the enemy plane flees. To be a complete solution, an agent must have knowledge of the potential dependencies between assumptions in a local level and any higher levels in the hierarchy. Similarly, in the Blocks World example from the previous chapter (Figure 2.6), the agent must have knowledge that recognizes any condition, in any goal, that should cause the local removal of the `empty` assertion. Thus, this knowledge must "cross" levels. As we saw in Chapter 2, knowledge that crosses levels can greatly lessen the advantages of hierarchical task decompositions by making knowledge less modular and maintainable.

3.3.2 Disallowing Context Change

The problem we have described only arises when the hierarchical context changes. We assume that most change occurs in the context due to changes in the outside world. The block is knocked over; an enemy plane retreats. The agent responds to these changes by updating assertions in the hierarchy. If these new assertions conflict with unchanged assumptions, the inconsistency problem results. Therefore, one way to avoid the problem is to disallow changes in the context. In effect, while problem solving, the agent metaphorically "closes its eyes" while reasoning progresses in lower levels of the hierarchy. The perception of the world is not changing, and thus the agent has no reason to update its assumptions in lower levels. Each time the agent "opens its eyes" and perceives a new world state, it regenerates its subtasks again, thus guaranteeing the new hierarchy is consistent with the new input state. In the Figure 2.6 example, regardless of the actual world state, the agent will not "see" that a block has moved. Further, each time it perceives the world, it will regenerate the `put-on-table` task hierarchy.

Although this approach ensures that the agent will update its reasoning with the new world state, it makes the agent both less responsive and less efficient. The world will be changing even when the agent is not sensing these changes (assuming exogenous domains). Thus, the agent can be internally consistent in its reasoning, yet be working on a problem no longer relevant to the world state. Inefficiency is caused by the need to regenerate reasoning. For instance, in the Blocks World, moving a block onto the table requires executing a series of steps in the GRIPPER problem space, all as part of the implementation of a single `put-down` operator. Assume the world is sensed after each primitive step in the Blocks World. After each step, the agent completely regenerates the hierarchy, as we described above. Thus, the knowledge to activate the `put-down` operator must be activated for each step in the solution of the problem, even though that knowledge really only needs to be activated once. These drawbacks make this solution less desirable in highly dynamic, exogenous domains, although the solution may be acceptable in primarily endogenous domains or ones in with slow time constants of change, relative to the rate of the generation of a primitive operator.

3.3.3 Aggressive Response to Context Change

In contrast to the previous solution, another solution would be allow the input to the world to occur with regular frequency, but to respond to changes in the world by automatically retracting the hierarchy. In this solution, the agent responds very aggressively, by restarting any reasoning that is potentially inconsistent. This approach is inexpensive (i.e., it requires no KBAC) while retaining full responsiveness to the external world. However, this approach will oftentimes be overly aggressive and retract the hierarchy in response to a change not relevant to the the agent's reasoning. For example, if block color was an input to the agent's state, and the color of some block changed, the agent's hierarchy of assertions would be retracted, even though color is not relevant to the tasks we have been considering.

This approach, like the previous one, is appropriate primarily for endogenous domains or exogenous domains with a slow time constant relative to the rate of the generation of primitive operators. For instance, this approach can be readily applied to agents for the Blocks World. However, unlike the previous approach, this approach may also be appropriate even for fast time constant domains, if changes can be filtered to limit the changes the agent perceives to only goal-relevant changes. For example, if the agent used a pre-processor that determined that "color" was not relevant to the current task in the previous example, then the agent could avoid unnecessary regeneration. The AIS architecture (Hayes-Roth et al., 1995), for example, uses a sophisticated mechanism for input pre-processing to the cognitive architecture that is equivalent to another instantiation of the architecture, but one with knowledge and processing specific for perception. This "input architecture" filters, integrates, and modulates the data passed to the task architecture. Such a scheme makes this alternative viable in many domains.

3.3.4 Limited Assumptions

Another way to avoid inconsistency is to limit the use of assumptions such that only monotonic assertions are created in subtasks. In this case, regular truth maintenance can provide reasoning consistency in the hierarchy because assumptions are restricted or limited. There are a number of different ways the assumptions could be limited. We examine a couple below.

Theo, as we described previously, is an extreme example of this approach. The only assumptions in Theo are the inputs. All reasoning is derived from the entailments of these sensors. Thus, Theo cannot reason nonmonotonically about any *particular* world state; only the world can change nonmonotonically.

Another restriction would limit assumptions to a single, separate memory, or, equivalently, allow assumptions only in the top level of the hierarchy. This solution ensures that the hierarchical context is always consistent because the assertions in any subtask are simply entailed from the assumptions and the higher context. Nonmonotonic reasoning in a single world state is possible. However, this approach is equivalent to the KBAC approach. Knowledge does not cross levels but only because all assumptions are grouped together in one level or a separate memory. All the knowledge necessary to maintain consistency among all assumptions in KBAC is still necessary in this approach. In the Figure 2.6 example, the agent records the assumption that the space above the gripper is empty in the top goal, rather than within the `put-down` subtask. However, the agent obviously still requires knowledge that recognizes when the `empty` assumption should be removed.

In addition to the drawbacks of KBAC, moreover, this approach also removes the explicit relationship between assumptions and the subtasks for which they were created. The "hierarchy management" module we illustrated in Figure 3.2 can remove assumptions for a subtask when that subtask is no longer active. For example, if the agent decided that putting down a block was no longer a relevant task, the `empty` assumption could be removed along with the `put-down` goal. In most architectures, this removal is automatic, and thus requires no additional task knowledge. This new approach, however, requires explicit "clean up" knowledge, because the assumptions are divorced from their subtask. If `put-down` is interrupted, the agent needs knowledge to remove the `empty` assumption.

3.3.5 Fixed Hierarchical Justification

Another alternative concentrates on ensuring that the reasons for initiating a level are still valid throughout the execution of the subtask, as illustrated in Figure 3.7. When each new level of the hierarchy is created, the architecture identifies assertions at each of the higher levels in the hierarchy that led to the creation of the new level. These assertions together form what we will call a "subtask support set" (or just "support set"). In Figure 3.7, assertions a_{14}, a_{15}, e_{14}, and e_{19} are the support set for $Level_2$ while a_{12}, a_{22}, e_{22} "support" $Level_3$. These support sets, in effect, form justifications for levels in the hierarchy. When an assertion in a support set is removed (e.g., a_{22}), the agent responds by removing the level ($Level_3$).

Architectures such as the Procedural Reasoning System (PRS) (Georgeff and Lansky, 1987) and Soar (Laird et al., 1987) use architectural mechanisms to retract complete levels of the hierarchy when the support set no longer holds. PRS checks the continuing applicability of each active subtask ("knowledge area") prior to continuing reasoning in that level. Soar, on the other hand, determines if any architecture-generated subgoals have been resolved before installing a new level in the hierarchy. Although the details are different, in both cases, the architectures ensure that some initial, activation conditions still hold before further processing. Conceptually, those activating conditions make up the "support set" for each subtask.

A significant disadvantage of the support set is that it is *fixed.* The support set is computed for the initiation of the subtask but is not updated to reflect reasoning that occurs within the subtask. For example, in Figure 3.7, suppose that assumption a_{34} depends upon assumptions a_{22} and a_{21} (represented in the figure by the dotted, arrowed lines). The support set does not include a_{21} (a_{21} may not have even been present when $Level_3$ was created.). When a local assumption depends upon an assertion not in the support set, then a change in that assertion will not directly lead to the retraction of the assumption or the level. Thus, approaches using *fixed hierarchical justification* (FHJ) may require knowledge-based assumption consistency for assumptions in the subtask.

Another way to ensure consistency using Fixed Hierarchical Justification would be to limit the reasoning in the subtask so that it tested only those features in memory that

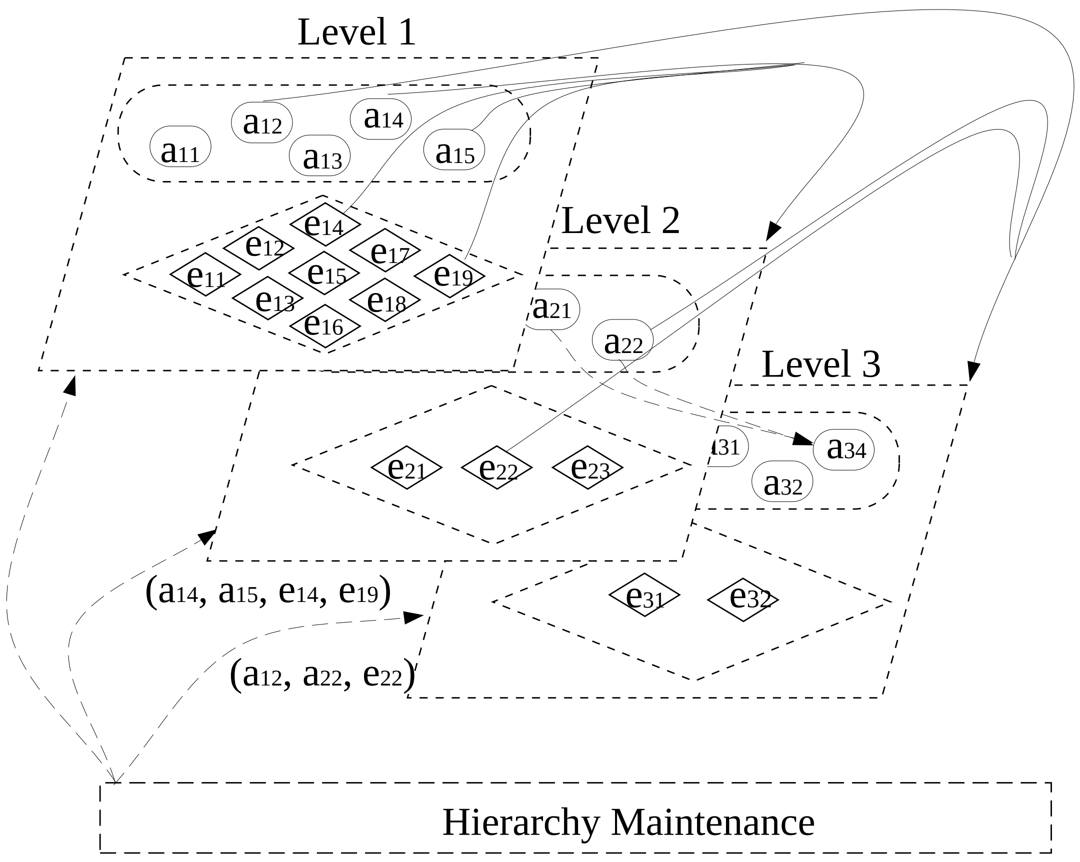


Figure 3.7: Fixed Hierarchical Justification. Assumptions are labeled with an "a," entailments with an "e." The first subscript denotes the level and the second an identification number within the level. Support sets for each level (see text) are represented as a list of assertions associated with the hierarchy maintenance for each level.

	Evaluation Criteria			
Approaches	Additional cost (KBAC)	Reduced Efficiency	Reduced Responsiveness	Monotonic reasoning only in subtask
Knowledge-based Assumption Consistency	Yes	No	No	No
Disallowing Context Change	No	No	Yes	No
Aggressive Response	No	Yes	No	No
Limited Assumptions	Yes	No	No	Yes
Fixed Hierarchical Justification	Yes	No	No	No

Table 3.2: Summary of potential approaches, based on the evaluation criteria described in Section 3.3.

were also members of the support set for the local subtask. In Figure 3.7, a_{21} would be included in the support set of $Level_3$ because the calculation a_{34} will depend on a_{21}.[4] This approach is the one used by PRS and provides an assurance of consistency. If we applied this solution to the TacAir-Soar example, `intercept` would now require the number of enemy planes as a precondition of the subtasks because the reasoning for the subtask will depend on this number. Similarly, in Figure 2.6, the support set for the `put-down` subtask would include the empty space. Thus, although this solution does not require the explicit knowledge we described for KBAC, it does require that the knowledge designer identify all the potentially relevant features that may used in the processing of the subtask. Additionally, the resulting system may be overly sensitive to the features in the support set if those features only rarely impact reasoning.

Fixed Hierarchical Justification does require less explicit reasoning consistency knowledge than the previous solutions but still requires it if access to the whole task hierarchy is possible (as it is in Soar). Thus, an agent's ability to make subtask-specific reactions to unexpected changes in the environment is limited by the knowledge designer's ability to anticipate and explicitly encode the consequences of those changes.

3.3.6 Summary of Potential Approaches

Table 3.2 presents a summary of the approaches we have considered thus far, evaluated along the dimensions we introduced earlier. Knowledge-based assumption consistency, assumptions limited to a single memory, and fixed hierarchical justification all require some form of consistency knowledge represented in the agent's domain knowledge, thus increasing the cost of building agents using these solutions. In addition, the limited assumptions approach also does not allow nonmonotonic reasoning within the subtask. On the other hand, additional knowledge is not required in the approach that disallows changes to the context nor in the aggressive response solution. However, each of these

[4]Alternatively, a_{34} could also computed in a separate subtask, with a_{21} a member of that subtask's support set.

approaches is limited in other ways. Responsiveness is potentially poor when using the first approach, while the efficiency is potentially compromised due to regeneration in the second. The table shows that no approach we have examined thus far does not impact one of the indicated evaluation criteria. In the following section, we will search for approaches that do not impact these factors.

3.4 Extensions to Truth Maintenance

All the previous solutions failed along at least one dimension of our evaluation. We now present two new solutions for reasoning maintenance in hierarchical systems that will require no additional cross-level assumption consistency knowledge, allow locally unjustified, non-monotonic assumptions, and architecturally resolve potential inconsistencies. We will also examine the efficiency of each solution. Both of these new solutions are based on the idea that the local processing can be justified with respect to the higher level context; in this sense, they are both extensions to the non-hierarchical truth maintenance approaches we reviewed earlier. Our extensions assume that an agent can compute the higher level dependencies of every assumption in a subtask, similar to the the non-hierarchical agent's computation of direct dependencies for entailments. The new computations require that the inference engine record every variable test made during the course of processing. In Soar, these calculations are available from its production rule matcher. However, these calculations may not be supported in other architectures, requiring modifications to the underlying inference engine.

Although we will explore these solutions in some depth, they represent only two points in the space of all possible solutions. Other approaches (including combinations of the new approaches) could be investigated as well. We were led to the first approach, Assumption Justification, by a desire for fine-grain reasoning maintenance where each assumption could be individually maintained or retracted as appropriate. We were led to the second approach when we discovered that fine-grain maintenance came at a significant computational overhead, and when we observed the structure of problem solving for well-decomposed tasks.

3.4.1 Assumption Justification

The first new approach is based on the idea that a local assumption can be justified with respect to assertions in higher levels of the hierarchy. *Assumption Justification* treats each assumption in the hierarchy as if it were an entailment with respect to assertions higher in the hierarchy. Locally, an assumption is handled exactly like an assumption in a non-hierarchical system. However, each assumption is justified with respect to dependent assertions in higher levels. When this justification is no longer supported, the architecture retracts the assumption. Assumption justification thus requires only local consistency knowledge for assumptions, which was true in the non-hierarchical truth maintenance approaches as well, while no additional knowledge is needed to reason about the interaction between the local assumptions and other goals.

For an example of how Assumption Justification works, refer again to Figure 3.7. Assumption a_{34} depends on a_{22} and a_{21}. We assume that a_{22} remains a member of the fixed support set of $Level_3$. Thus, if the architecture retracts a_{22}, it also removes $Level_3$

```
C := Set of assertions used to create assumption A
$make\_assumption\_justification\_for\_assertion(C, A)$

PROC $make\_assumption\_justification\_for\_assertion(dependencies\ C,\ assumption\ A)$
    FOR $c_x$ := Each assertion in $C$
①       IF $c_x \rightarrow level$ higher $A \rightarrow level$
            $add\_c_x\_to\_assumption\_justification(c_x, A)$
②       ELSE IF $(c_x \rightarrow level == A \rightarrow level)$
            $just_{c_x}$ := assumption justification of $c_x$
            $add\_just_{c_x}\_to\_assumption\_justification(c_x, A)$
        END(IF)
    END(FOR)
END(PROC)
```

Table 3.3: A procedure for building assumption justifications.

(and thus a_{34}), as in Fixed Hierarchical Justification. However, now when the agent asserts a_{34}, the architecture builds a justification for the assumption that includes a_{22} and a_{21}. Now if the agent retracts a_{21}, the justification for a_{34} is no longer supported and the architecture also retracts a_{34}. In the Figure 2.6 example, when the agent creates the empty assumption, the architecture computes the assertions on which it depends, including any input assertions. When the world changes, these higher level assertions change and the agent retracts the empty assumption. The architecture ensures reasoning consistency across hierarchy levels because the assumption never persists any longer than the higher level assertions used in its creation.

Assumption Justification is another type of limitation on assumptions (i.e., as we described in Section 3.3.4). In this approach, the hierarchical context limits the persistence of local assumptions: when the context changes dependently, the architecture removes the assumption. However, because the assumptions are unjustified in the local subtask, the architecture still supports local nonmonotonic reasoning. Thus, assumption justification appears to meet all our functional criteria. However, in order to assess its impact on the efficiency of the system, we must consider some implementation details.

Implementing Assumption Justification

Creating assumption justifications requires computing dependencies for each assumption. The justification procedure must examine all the local assertions that contributed to the situation that created the assumption because assumption dependencies can be indirect. For example, the count in the TacAir-Soar example depends on the previous count, so the dependencies of the new count must include the dependencies of the prior count.

We consider two ways in which to compute the assumption justification. The first is simply to compute the justification "on demand"; when the agent creates an assumption, the architecture computes the dependencies in the higher context by tracing back through the reasoning in the local goal. The second approach computes assumption justifications for every assertion in the local goal, even the local assertions. In effect, the architecture caches the higher level dependencies for each local assertion when the assertion is created. The advantage of this caching is that the architecture can create a justification for any

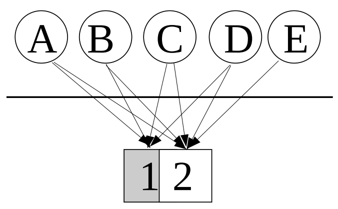


Figure 3.8: An assumption can replace another assumption nonmonotonically.

particular assumption by simply (but uniquely) concatenating the justifications of the assertions directly contributing to the creation of the assumption. We chose this second option for computing assumption justifications because the dependencies for any assertion need to be computed only once, even if the assertion contributes to the creation of many local assumptions.

Table 3.3 shows a pseudocode procedure for computing the assumption justification. The procedure builds an assumption justification for some assumption **A** by collecting any non-local assertions that contributed to the creation of **A** (Condition #1) and the assumption justifications of any local assertions (Condition #2). It is necessary to add the justification of the local assertion c_x, rather than the individual assertion itself, because the architecture can retract a local assertion for reasons other than a change in the hierarchical context.

Unlike the addition of a non-local assertion (which can be accomplished with a single pointer check), $add\_just_{c_x}\_to\_assumption\_justification(c_x, A)$ may be non-trivial because the procedure must uniquely add each dependent assertion to the assumption justification of **A**. This procedure call determines the complexity of the overall procedure. Assume that each assertion is created by a unique instantiation of knowledge (true in the worst case). In this case, there are at most $(n-1)$ local instantiations whenever the nth assertion is created. Thus, the assumption justification procedure needs to call $add\_just_{c_x}\_to\_assumption\_justification(c_x, A)$ no more than n times for any call to the assumption justification procedure. This limit provides an upper bound of $O(n)$ on the complexity of the assumption justification procedure. Thus, the cost of building an individual assumption justification is linear in the number of assertions, n, in the level. However, the architecture executes the assumption justification procedure for every assertion in the level. Thus, the worst case cost for building all the justifications in a particular level is $O(n^2)$.

A complication arises when an assumption non-monotonically replaces another assumption. The architecture must disable the initial assumption, rather than permanently delete it from memory, because the architecture must restore the initial assumption if the second assumption is retracted. In Soar, this requires that an assumption be removed from the agent's primary or "working" memory but retained in a secondary memory. For example, in Figure 3.8, when the agent asserts **E**, the agent also asserts **2** in the local level, replacing **1**. If the agent removes **E**, assumption justification will retract **2**, as desired, but it must also re-enable **1**, adding it back to the agent's working memory. Thus assumption **1** must remain in memory, although disabled. More concretely, consider again the TacAir-Soar example. When the enemy plane departs, the agent removes the current count. However, the agent must also re-enable the previous count, because each iteration of the count non-monotonically replaced the previous one. For the prior count to be im-

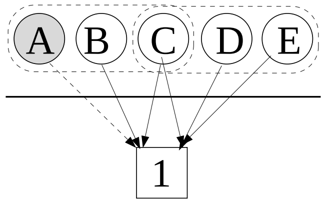


Figure 3.9: Multiple assumption justifications for the same assumption.

mediately available, the agent must store this prior assumption in memory. Assumption justification thus requires the storage of every subtask assumption while the subtask is active, even when the assumptions have been nonmonotonically replaced. Thus, the n we discussed above is bound not by the number of "active" assertions but by the total number of assertions the agent makes while the subtask is instantiated. As a result, the longer a subtask is active, the more likely Assumption Justification will begin to impact overall performance severely due its polynomial complexity.

Figure 3.9 illustrates a second difficult problem. An assumption can have multiple assumption justifications and these justifications can change as problem solving progresses. Assumption **1** initially depends on assertions **A**, **B**, and **C** in higher levels. Now assume that later in the processing, the agent removes **A**, which normally would result in the retraction of **1**. However, in the meantime, the context has changed such that **1** is now also justified by $\{$**C**, **D**, **E**$\}$. Now when the agent removes **A**, the architecture cannot immediately retract **1** but must determine if **1** is justified from other sources. Thus, as problem solving progresses within a level, the architecture must recognize and create new justifications of the local assumptions as they occur.

This work provides a theoretical explanation for the costs associated with Assumption Justification. An actual implementation of assumption support in Soar was completed by members of the Soar research group at the University of Michigan other than the author (Laird, 1998). Experiments using assumption justification in Soar, using Air-Soar, a flight simulator domain (Pearson et al., 1993), showed the overhead of maintaining all prior assumptions in a level negatively impacted agent performance, a result not surprising given the analysis presented here. In this domain, assumption justification had significant computational cost, requiring 50% more time than the Fixed Hierarchical Justification version of Soar for the same task. In addition, the number of assumption justifications maintained within a level continued to grow during execution, for the reasons we explained above. In Air-Soar, some levels could persist for many minutes as the plane performed a maneuver, leading to a continual increase in the amount of memory required. Thus, assumption justification fails to meet the efficiency criteria we described above on both theoretical and empirical grounds. These results were strong enough that we decided not to explore assumption justification empirically in this research. For the remainder of the dissertation, we will use assumption justification only for theoretical analysis and comparison.

3.4.2 Dynamic Hierarchical Justification

Our second solution provides a coarser-grain maintenance of assumptions in a level, finessing some of the complexities of Assumption Justification. Instead of maintaining

```
$C$ := Set of assertions used to create assumption $A$
$add\_dependencies\_to\_support\_set(C, A)$

PROC $add\_dependencies\_to\_support\_set(dependencies\ C,\ assumption\ A)$
    FOR $c_x$ := Each assertion in $C$
①       IF $\{c_x \rightarrow level$  higher  $A \rightarrow level\}$
            $add\_c_x\_to\_assumption\_justification(c_x, A)$
②       ELSEIF $\{(c_x \rightarrow level == A \rightarrow level)$ and
                $(c_x$ is an assumption$)\}$
            break
③       ELSEIF $\{(c_x \rightarrow level == A \rightarrow level)$ and
                $(c_x$ is not an assumption$)$ and
                $(c_{x_{inspected}})\}$
            break
④       ELSEIF $\{(c_x \rightarrow level == A \rightarrow level)$ and
                $(c_x$ is not an assumption$)$ and
                (not $c_{x_{inspected}})\}$
            $C_x$ :=  Set of assertions supporting entailment X
            $add\_dependencies\_to\_support\_set(C_x, X)$
            $c_{x_{inspected}} := TRUE$
        END(IF)
    END(FOR)
END(PROC)
```

Table 3.4: A procedure for dynamic hierarchical justification.

support information for each individual assumption, our second solution maintains support information for the complete subtask, similar to Fixed Hierarchical Justification. This simplification decreases the complexity and memory requirements for the support calculations, but means that the complete level retracts when the architecture retracts any dependent assertion. We call this solution *Dynamic Hierarchical Justification* (DHJ), because the support set grows dynamically as the agent makes assumptions for a subtask. When a DHJ agent asserts a_{34} in Figure 3.7, the architecture updates the support set for $Level_3$ to include a_{21}. Assumption a_{22} is already a member of the support set and does not need to be added again. When any member of the support set for $Level_3$ changes, the architecture will retract the entire subtask. In the Figure 2.6 example, the architecture adds the assertions that led to the creation of the `empty` assumption to support set of the `put-down` subtask. When the world changes such that the space is no longer empty, the agent retracts the `put-down` subtask, where Assumption Justification would retract only the `empty` space assumption. Thus Dynamic Hierarchical Justification enforces reasoning consistency across the hierarchy because a subtask in the hierarchy persists only as long as all higher-level dependent assertions.

Implementing Dynamic Hierarchical Justification

Whenever a new assumption is created, the architecture must determine the dependencies as for Assumption Justification but the dependencies are added now to the support set for the local subtask. Unlike Assumption Justification, in Dynamic Hierarchical Justification we use the "on-demand" model for computing justifications rather than the cache model. The on-demand computation is more appropriate than the cache model for DHJ because any individual assertion needs to be examined only once over the duration of the subtask. Thus, DHJ will not need to re-compute dependencies, which motivated the cached approach in assumption justification.

Table 3.4 shows the procedure for computing the support set in DHJ. As in Assumption Justification, the architecture can simply add dependencies in any higher level directly to the support set (Condition #1). When the architecture computes the dependencies for a local assertion (Condition #4), the assertion is flagged as having been inspected, as shown. In future calls to the DHJ procedure, these assertions can simply be ignored (Condition #3) because the architecture has already computed those dependencies. The architecture can also ignore dependent, local assumptions because the architecture has already added that assumption's dependencies to the support set (Condition #2).

The recursive call to *add_dependencies_to_support_set* (Condition #4) is the only non-trivial computation in the DHJ procedure. However, as in assumption justification, this procedure needs to be called only once for any assertion. Thus, the worst case complexity to compute the dependencies is linear in the number of assertions in the level, as in Assumption Justification. However, unlike Assumption Justification, DHJ requires at most one inspection of any individual assertion, rather than repeated inspections for each new assumption. Thus the architecture needs to call *add_dependencies_to_support_set* at most n times for any subtask consisting of n assertions and the worst case cost of computing the dependencies over all the assumptions in a level with DHJ remains $O(n)$. This reduction in the complexity of the consistency calculation potentially makes Dynamic Hierarchical Justification a much more efficient solution than Assumption Justification, especially as the number of local assertions grows.

Additionally, the two technical problems we outlined for assumption justification do not impact DHJ. DHJ never needs to restore a previous assumption. When a dependency changes, the architecture retracts the entire level. Thus, DHJ can immediately delete from memory non-monotonically replaced assumptions. Secondly, DHJ collects all dependencies for assumptions, so there is no need to switch from one justification to another. In Figure 3.9, dependencies **A**, **B**, **C**, **D**, and **E** are all added to the support set. These simplifications can make the support set overly specific but reduce the memory and computation overhead incurred by Assumption Justification.

However, DHJ will sometimes cause the "unnecessary" removal of prior reasoning, which may need to be regenerated. In the TacAir-Soar agent, for example, if the enemy plane fled as we described, DHJ would retract the entire level associated with the counting subtask. The count would then need to be re-started from the beginning. Assumption Justification, on the other hand, would retract only those assumptions that depended upon the presence of the departed plane. In the best case, if this plane was counted last, the architecture would need to retract only the the final count, and no new counting would be necessary. The cost incurred through the *regeneration* of previously-derived assertions is the primary drawback of Dynamic Hierarchical Justification. However, unlike the regeneration we discussed for previous approaches (Sections 3.3.2 and 3.3.3), DHJ

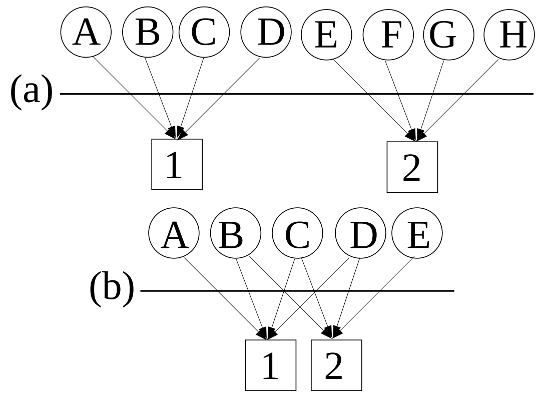


Figure 3.10: Examples of (a) disjoint dependencies and (b) intersecting assumption dependencies.

retracts a subtask only when continued reasoning in that subtask would be inconsistent with the higher context. Thus, regeneration should be less costly in this approach than in the previous ones.

In addition to implementing the Dynamic Hierarchical Justification procedure, we also made one change to Soar's procedure for firing rules. In Soar, as we will examine in more detail in the next chapter, all matching rules are fired in (simulated) parallel, including those that assert entailments and those that assert assumptions. In some cases, this parallelism leads to inconsistency because an assumption can be created while previous assumptions or inputs are still being entailed. With DHJ, this parallelism can be especially problematic because an assumption that is created "too fast" can lead to removal of the entire subtask. In order to address this problem, we separated the rule firings into distinct "entailment" and "assumption" phases. Assumptions are only created when all possible entailments have been applied. When a new assumption is asserted, additional entailments may be triggered, etc. Although this change did not require significant additional computation (the architecture already classifies each rule according to the type of computation it makes), it does decrease potential parallelism in architecture processing. We will explore further limitations on parallelism and discuss the impact of these limitations on execution in the next chapter.

3.5 The Influence of the Task Decomposition on Assumption Justification and Dynamic Hierarchical Justification

We were led to Dynamic Hierarchical Justification by our experience with Fixed Hierarchical Justification, in which entire sub-hierarchies of reasoning are automatically retracted as we outlined above. However, retracting subtasks, rather than individual assertions, is a heuristic simplification. Because DHJ can lead to the regeneration of retracted subtasks, we now briefly examine why this heuristic is useful for execution domains and what it can tell us about this potential limitation of DHJ. We will explore the actual empirical costs of Dynamic Hierarchical Justification in Chapter 5.

For agent execution domains, we have assumed that the agent has the knowledge necessary to execute its task (Assumption 2). Under this assumption, little deliberate search is necessary for execution. However, the inconsistency problem we have been examining in this chapter can be viewed as a failure to backtrack in the search for a primitive operator. The world changes, leading to changes in the agent hierarchy and the agent must retract some of the knowledge it has previously asserted; it backtracks to a knowledge state consistent with the world state. All the solutions we have examined can be described as different ways to achieve (or avoid) backtracking. For instance, KBAC is a knowledge-based backtracking scheme.

Assumption Justification is a realization of dependency-directed backtracking (Stallman and Sussman, 1977). In dependency-directed backtracking, regardless of the chronological order in which the architecture made assertions, the architecture can identify and retract only those assertions that are directly dependent on a failure in the search, and retain non-dependent assertions. Similarly, in Assumption Justification, the architecture retracts only those assumptions that are directly affected by a change in the context. Assumptions made later in the processing, not dependent on the change, are unaffected. Consider the examples in Figure 3.10. In (a), assumptions **1** and **2** each depend upon disjoint sets of assertions. With Assumption Justification, removal of any assertion in **1**'s justification will result in the retraction of **1**; **2** is unchanged, even if the architecture created **2** after **1**.

With Dynamic Hierarchical Justification, the agent retracts all reasoning in the dependent subtask (and all lower levels in the hierarchy). Therefore, some assertions not dependent on the change in the context will be withdrawn. DHJ is thus similar to *backjumping* (Gaschnig, 1979). Backjumping uses heuristics to determine to where in a search space a current search should backtrack or "backjump." The heuristics used by backjumping are based on syntactic features of the problem. For instance, in constraint satisfaction problems, the backjumping algorithm identifies which variable assignments are related to other variable assignments via the constraints specified in the problem definition. When a violation is discovered, the algorithm backtracks to the most recent, related variable (Dechter, 1990). Intervening variable assignments are discarded. In DHJ, when an assertion in the hierarchy changes, the system "backjumps" to the highest level in the hierarchy for which the changed assertion is not dependent. In Figure 3.10 (a), all the dependent assertions are collected in the support set for the subtask. Thus, the entire subtask is removed if any of the higher level assertions change. Further, assume the removal of the subtask was due to a change in **A**. If the same subtask is reinstated, assumption **2** may need to be regenerated. This regeneration is *unnecessary* because **2** did not need to be retracted to avoid inconsistency.

To further examine the nature of the heuristic used in DHJ, consider now the situation in in Figure 3.10 (b). The dependencies of assumptions **1** and **2** have a large intersection. If assertions **B**, **C**, or **D** change, then all the local assertions will be retracted, if we assume that everything in the local level is derived from **1** or **2**. In this situation, assumption justification pays a high overhead cost to track individual assumptions, when (most) everything in the local subtask is removed simultaneously. Because DHJ incurs no such overhead, DHJ is a better choice when the intersection between assumption dependencies is high. If we could expect tasks structured more like the situation in (b), rather than (a), we could expect that DHJ would infrequently regenerate reasoning *unnecessarily* retracted.

In arbitrary domains, of course, we expect both intersecting and disjoint dependencies

Approaches	Evaluation Criteria			
	Additional cost (KBAC)	Reduced Efficiency	Reduced Responsiveness	Monotonic reasoning only in subtask
Knowledge-based Assumption Consistency	Yes	No	No	No
Disallowing Context Change	No	No	Yes	No
Aggressive Response	No	Yes	No	No
Limited Assumptions	Yes	No	No	Yes
Fixed Hierarchical Justification	Yes	No	No	No
Assumption Justification	No	**Yes**	No	No
Dynamic Hierarchical Justification	No	**???**	No	No

Table 3.5: Qualitative evaluation of potential approaches including Assumption Justification and Dynamic Hierarchical Justification.

among different assumptions. However, in Chapter 2, we assumed that hierarchical decomposition was useful for tasks that were nearly decomposable (Assumption 4). Thus, the goal of a hierarchical decomposition is to separate independent subtasks from one another. A consequence of this separation is that the specific dependencies in the higher levels are more tightly coupled. We can argue that (b) represents a more nearly decomposed problem than (a). In (a), two independent assumptions are being pursued. These assumptions could potentially be computed by separate subtasks in the decomposition. In (b), on the other hand, the assumptions in the subtask are closely tied together in terms of their dependencies and represent problem solving more reasonably computed by the same subtask. In (b), the total number of dependent assertions does not necessarily grow as a function of the assumptions in the local level, while in (a) it does.

Assumption justification thus excels when there are many orthogonal dependencies in a subtask, which we could say represents a task that has been poorly decomposed. DHJ, on the other hand, excels when the dependencies for the assumptions in any individual subtasks are shared, resulting in a "good" decomposition. In this situation, DHJ suffers from few unnecessary regenerations while avoiding the overhead costs of assumption justification. As will will see Chapter 5, another benefit of DHJ is that it can point out problem areas in the decomposition, acting as an aid for the development of better decompositions.

3.6 Summary

In this chapter, we have shown that inconsistency arising from a failure of the agent to respond to context changes can be solved in a number of different ways. Our emphasis in exploring this problem was to find efficient, *process-based* solutions that would reduce the cost of knowledge design while retaining the agent's ability to make local, persistent, nonmonotonic assumptions. Table 3.5 summarizes the results of our analysis thus far.

Both of the new approaches we have introduced, Assumption Justification and Dynamic Hierarchical Justification, solve this inconsistency problem through architectural means. They require no cross-level consistency knowledge and do not compromise responsiveness or the agent's ability to make local, nonmonotonic assumptions. However, Assumption Justification proved to be prohibitively inefficient, and thus fails to meet our qualitative evaluation criteria. We argued that Dynamic Hierarchical Justification would not suffer from unnecessary regeneration and thus avoid some inefficiency. However, necessary retractions may also lead to regeneration as well. For instance, in the Blocks World example of inconsistency, when the `put-down` operator is retracted due to the change in the support set, the agent will still need to put the block onto the table, and reinitiate the `put-down` subtask. Thus, even if our argument in the previous section is correct, we still cannot know with certainty if Dynamic Hierarchical Justification will be acceptably efficient. In Chapter 5, we will empirically evaluate the efficiency of our architectural solutions to the inconsistency problems, after addressing inconsistency arising from overly aggressive response to context changes, the subject of the next chapter.

Chapter 4

Overly Aggressive Reaction to Context Changes Due to Multiple, Simultaneous Threads of Reasoning

A foolish consistency is the hobgoblin of little minds....
– Ralph Waldo Emerson

Inconsistency can arise in an agent's hierarchical context when an agent responds too quickly to change in the context. A response is too quick if it occurs before assertions higher in the hierarchy have had an opportunity to apply and those new assertions would have made the lower-level knowledge inapplicable. This inconsistency results from the agent's simultaneous pursuit of different threads of reasoning. Supporting multiple threads of reasoning is important because the different threads allow an agent to pursue independent lines of reasoning simultaneously. Across-level inconsistency arises when one thread of reasoning turns out to be dependent on another, on-going thread of reasoning higher in the hierarchy.

We introduce several potential solutions to inconsistency arising from multiple threads of reasoning, including Subtask-limited Reasoning (SLR). Like Dynamic Hierarchical Justification from the previous chapter, Subtask-limited Reasoning is a heuristic solution that guarantees consistency across the levels of the hierarchy, requires no additional agent knowledge, and can be computed efficiently. Rather than attempting to compute dependencies between assertions applied in different threads of reasoning, SLR simply delays reasoning in a subtask until threads of reasoning in higher levels of the hierarchy have been fully elaborated. Thus, although the Subtask-limited Reasoning computations can be computed efficiently, SLR can delay some reasoning in hierarchical agents. We argue that the delay introduced by SLR is not significant because most activity in the hierarchy is usually concentrated in the lowest levels of the hierarchy at any particular time. In Chapter 5, we provide empirical evidence that shows not only that the cost of SLR is insignificant, but that SLR contributes to an improvement in overall performance because it eliminates some unnecessary reasoning, which would have occurred in a system that allows all threads to be pursued simultaneously.

4.1 Overly Aggressive Reaction to Changes in Context

In the Blocks World example we presented in Chapter 2, we showed how inconsistency could arise if an agent was simultaneously pursuing one thread of reasoning while also following another line of reasoning in a lower level of the hierarchy. These different threads of reasoning are not different paths in an exponential search but rather different elaborations of the current situation using execution knowledge. Different threads of reasoning could potentially be executed independently of one another. For example, an agent with multiple processors might be able to commit one processor to one thread of reasoning and a different processor to another thread. As a shorthand, in this chapter, we call the simultaneous pursuit of different threads of reasoning *parallelism*. We will describe more concretely what we mean by parallelism in the following section. For now, we simply assume that an agent applies any knowledge that it can as soon as that knowledge becomes applicable, and it can apply multiple pieces of knowledge simultaneously, thus following different threads of reasoning simultaneously.

In the Blocks World example, Dynamic Hierarchical Justification, the solution we introduced in the previous chapter, would avoid inconsistency due to overly aggressive response in this specific instance. When the position of **block-3** changed, the architecture would remove the `put-down` subtask (the subtask containing the `empty` assertion), and the thread of reasoning leading to the initiation of the `move-down` operator would thus be interrupted. In general, however, inconsistency resulting from multiple threads of reasoning can arise from sources other than assumptions, and thus Dynamic Hierarchical Justification does not provide a complete solution to the problem. In order to illustrate the problem more completely, we now consider a specific example that is not solved by Dynamic Hierarchical Justification.

4.1.1 Example: TacAir-Soar

Suppose a TacAir-Soar agent has determined an enemy plane is hostile. In TacAir-Soar, the identification of hostile aircraft may be determined via an IFF sensor.[1] For the purposes of this discussion, we make a simplifying assumption that an agent must either receive a friendly IFF signal or radio confirmation of an aircraft's friendliness; otherwise, it assumes the enemy is hostile. In actuality, this classification is considerably more complicated.

The agent begins an intercept of the plane it has classified as hostile. The intercept continues until the agent comes within missile range. Suppose now that at the same time the agent achieves missile range, it also receives an incoming radio message, indicating that the hostile plane it is targeting is actually friendly. The agent can now pursue two different threads of reasoning: parsing the incoming radio message and firing the missile. There is no necessary dependence between these two tasks and we assume that the agent pursues them simultaneously. Although the transitive closure of the agent's knowledge would specify that the target is friendly and it should not fire the missile, it takes time to come to this conclusion in the reasoning thread related to the radio message. During this computation, at the lower levels of the hierarchy, the agent might still believe the agent is hostile and launch the missile.

[1]IFF: identify: friend-or-foe

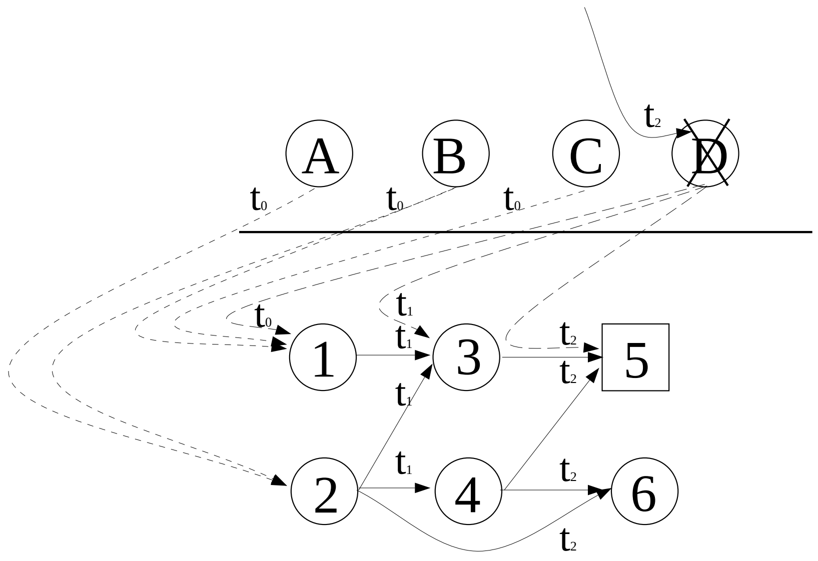


Figure 4.1: Inconsistency due to multiple threads of reasoning. Assumptions are represented as squares, entailments as circles. The horizontal line represents a hierarchical task/subtask relationship between the assertions above the line and the ones below.

In this example, the agent is responding too aggressively to a change in the hierarchical context that is known to be relevant (i.e., in missile range), before determining if other changes, like the radio message, impact the current reasoning as well. Inconsistency results because the agent has enough information to determine that the plane is friendly, but, because it takes time to compute this inference, the agent also acts as if the plane is hostile. Because an output action is generally nonmonotonic, the agent cannot readily "retract" the action it has initiated. Further, as is especially evident in this example, the inconsistency resulting from over-reaction can lead to irrational behavior on the part of an agent. In this case, the agent launches a missile against a plane known to be friendly.

Figure 4.1 provides an abstract illustration of inconsistency arising from multiple threads of reasoning. The conventions used in this figure are the same as those used in Figure 3.6. Entailments **1**, **2**, **3** and **4** are created before time t_2. At t_2, the architecture retracts entailment **D**, in the higher subtask, as part of an on-going thread of reasoning not explicitly shown in the figure. In most circumstances, **D** will have been an elaboration of input, and the architecture retracts it because the current input no longer supports this inference. Concurrent with the retraction of **D**, the architecture asserts **5** in one of the threads of reasoning in the local subtasks. **5** depends upon entailment **D**. The figure illustrates both a direct dependence (the direct line **D** to **5**) and indirect dependence (through **3** and also **1**). Although the architecture may be able to detect direct dependencies, indirect dependencies are much more difficult to recognize (and thus avoid).

Any resulting problem is dependent on the type of assertion **5** and the underlying architecture. If **5** is an entailment, then an agent with a truth maintenance component can recognize that **5** is no longer justified and withdraw it, as it will also do for **3** and **1**. If **5** is an assumption (as shown in the figure) and we are using Dynamic Hierarchi-

①	$A \wedge B \wedge C_1 \rightarrow C_2$
②	$A \wedge B \wedge C_1 \wedge D \rightarrow C_3$

Table 4.1: An example of the importance of parallelism in agent architectures.

cal Justification, then the entire subtask will be removed. This removal may cause the unnecessary regeneration of **2**, **4** and **6** because their dependencies have not changed. In both these cases, the problem leads to inefficiency – some knowledge is asserted and then almost immediately retracted – but not necessarily inconsistency. However, if **5** generates an output action, then the system is *acting* as if **D** is still believed, thus leading to potentially irrational behavior.

4.1.2 Supporting Multiple, Simultaneous Threads of Reasoning

In this section, we examine the way in which Soar supports multiple threads of reasoning using (simulated) parallelism. As we will see, Soar's current implementation may be overly parallel, which, in turn, allows inconsistency between multiple threads of reasoning.

Parallelism is used in Soar to allow an agent to pursue competing or mutually exclusive reasoning without committing to one of those courses arbitrarily. We call reasoning leading to a particular conclusion (such as an assumption or an output action) a *thread*. In order to illustrate how parallelism supports multiple threads of reasoning, consider the rules shown in Table 4.1. Both rules match in the same context, when memory contains **A**, **B**, $\mathbf{C}_1$ and **D**. However, the rules specify mutually exclusive actions. Rule 1 changes the value of $\mathbf{C}_1$ to $\mathbf{C}_2$ and Rule 2 changes it to $\mathbf{C}_3$. Without parallelism, when these rules match simultaneously, the agent will have to choose one of the two rules to apply. The process of deciding which one of many rules that can apply is known as *conflict resolution* (McDermott and Forgy, 1978). For example, in the OPS5 production system (Forgy, 1981), conflict resolution strategies include choosing the production that is most specific to the current situation, or choosing a production that matches against more recently instantiated elements, etc. In the example, if the most specific strategy had been chosen, Rule 2 would be selected and applied.

Conflict resolution is limited because the knowledge embedded in the process is local and fixed, what Newell (1990) calls a *trap-state mechanism.* Trap-state mechanisms are problematic because they do not allow an agent to bring all its knowledge to bear on a problem. For instance, for the rules in Table 4.1, assume another rule which indicates that $\mathbf{C}_2$ should be preferred over $\mathbf{C}_3$ in some specific circumstance. In an agent using fixed conflict resolution, Rule 1 may never get to fire. The result of Rule 2 leads to the removal of the Rule 1 instantiation because Rule 1 does not match once $\mathbf{C}_3$ is added to memory. Thus, the preference knowledge never applies either. In this example, the agent's choice of two mutually exclusive choices ($\mathbf{C}_2$ and $\mathbf{C}_3$) is made using a process that can only be indirectly influenced by the agent's knowledge.

In order to avoid using such a fixed procedure for conflict resolution, Soar uses simulated parallelism to allow all applicable knowledge to fire. In the example, both Rule 1 and Rule 2 to apply simultaneously. The agent must decide what to do about the two different choices represented by the different actions of the rules. In Soar, the agent represents the available choices explicitly in memory, thus allowing the agent to reason directly about the available choices using its knowledge. If the agent had knowledge to prefer $\mathbf{C}_2$, as we described above, then the agent could then apply this knowledge and finally choose $\mathbf{C}_2$. Parallelism in Soar thus allows the agent to avoid commitment to one of potentially many threads of reasoning until it has used all its available knowledge to consider to which choice it should actually commit. The agent, of course, does make a commitment to some particular choice sooner or later. The important capability is that the agent can use its knowledge to make these choices, rather than rely on a fixed, local procedure. Knowledge can be extended and customized for particular domains, whereas a fixed procedure that determines behavior is not readily extensible to new domains and tasks.

Soar thus uses parallelism to achieve knowledge-mediated conflict resolution. Importantly, the motivation for parallelization as a means to avoid arbitrary conflict resolution is local. A decision among several different choices is made within an individual subtask. However, Soar currently allows parallelism throughout the hierarchy. One benefit of such "unlimited parallelism" is that some performance improvement could be achieved on a parallel architecture. For example, an agent could elaborate one thread of reasoning in one subtask simultaneously with a separate line of reasoning in a different subtask. When these computations are independent, a truly parallel architecture will decrease the total time necessary for these computations by parallelizing them. However, as we further describe in Section 4.2.4, unlimited parallelism allows inconsistency because there is a potential logical dependence between threads of reasoning at different levels of the hierarchy. In particular, a higher level thread may lead to the conclusion that the lower subtask (which the lower thread is serving) is no longer a proper subtask to pursue. This distinction between local and unlimited parallelism will be important as we consider approaches to solving inconsistency arising from multiple threads of reasoning in the following sections because Soar uses a less restricted parallelism than is necessary to support knowledge-mediated conflict resolution.

4.2 Potential Solutions

How can an agent support multiple threads of reasoning and also avoid inconsistencies that lead to irrational behavior? In this section, we review a number of potential approaches, including a number of approaches from the previous chapter that also provide a solution to inconsistency resulting from multiple threads of reasoning. Again, as we suggested in the previous chapter, this review is not intended to represent all possible approaches but rather a survey of different points in the solution space.

Before reviewing the approaches, we again consider how to evaluate different approaches. The criteria are basically the same as those in the last chapter. Solutions should avoid the addition of new knowledge, maintain performance and responsiveness, and preserve the ability to pursue multiple threads of reasoning. We now consider each of these criteria in more detail.

Avoids additional "meta-level" consistency knowledge. Knowledge necessary to avoid inconsistency due to multiple threads of reasoning may have to reason about the interactions between different threads. We call this knowledge "meta-knowledge" because it is concerned with how the agent reasons, rather than how the agent should execute its task. We prefer solutions that eliminate the need for this meta-knowledge and thus reduce the overall cost of building agents.

Solves problem efficiently. New processes in the agent's architecture should not significantly impact the overall speed of the agent's processing. We prefer solutions that minimally impact the overall speed of an agent's execution of its task.

Remains responsive to the external world. We prefer solutions that do not compromise an agent's ability to respond quickly to new stimuli. This criterion is especially important for this particular problem because some solutions may slow response to ensure consistency.

Allows multiple threads of reasoning within a subtask. As we described above, the important advantage of multiple threads of reasoning is local to a subtask, providing a way of pursuing different lines of reasoning. Thus, we require multiple threads of reasoning within a subtask be maintained, in order to achieve this functional capability.

4.2.1 Meta-level Consistency Knowledge

As we described for inconsistency due to persistence, explicit domain knowledge that recognizes and resolves potential inconsistencies can be developed for the agent. The function of this knowledge is to recognize and interrupt reasoning that would lead to inconsistency. In the case of persistence, we assumed that the agent could recognize inconsistency *post hoc*, and then immediately act to remove the persistent assumption no longer consistent with the hierarchical context. Such a solution is sufficient for inconsistency arising from multiple threads of reasoning only for non-output actions. In order to prevent irrational behavior, the agent must interrupt reasoning leading to an output before it occurs. In order to do so, it needs knowledge to anticipate interactions between different threads of reasoning. Because the additional required knowledge concerns agent processing, rather than the domain, we describe this solution as the "Meta-Level Consistency Knowledge" (MLCK) solution.

MLCK requires that an agent be able to reason about its own processes. Many architectures do not support this requirement. An exception is MAX (Kuokka, 1991), an extension of Prodigy research for execution domains. The reasoning language used by MAX, *lframes*, is both executable and declarative, allowing the agent to reason about its processing. Although reasoning about threads of reasoning is not a specific capability of MAX, the *lframe* approach provides an agent with the ability to reason about its knowledge. For instance, MAX could examine its currently matching knowledge before applying it, to determine if any inconsistencies would arise.

Similar to our observations concerning knowledge-based assumption consistency, the literature is devoid of any discussion of inconsistency arising from multiple threads of reasoning or or knowledge-based solutions to this problem. Again, we believe that the problem has been ignored because the knowledge-based solution is simply regarded as part of the domain knowledge. For example, in Soar, which does not have the ability to reason about its processing as easily as MAX, the solution to some problems due to multiple threads of reasoning has been to extend a thread of reasoning artificially by breaking it into many small pieces. This process has the effect of delaying reasoning in a subtask until changes higher in the hierarchy have had an opportunity to be elaborated. Such additional knowledge requirements can add significant cost to agent development, especially identifying when the knowledge is necessary, which usually happens following the observation of errors in behavior.

4.2.2 Disallowing Context Change

This approach is the same approach we described in Section 3.3.2. The agent does not accept new input while generating the decomposition and regenerates the entire hierarchy following each poll of input. This approach solves inconsistency due to multiple threads of reasoning because, without input, context change in the hierarchy is minimized during decomposition. As before, the primary drawback of this solution is that the agent's responsiveness is compromised. In highly dynamic domains, the agent's internal processing will be consistent, but that processing may be inconsistent with the actual external environment.

4.2.3 Aggressive Response to Context Change

Again, this approach is the same as the one we described in the previous chapter, in Section 3.3.3. The agent reacts to input immediately by removing the current hierarchy. In the TacAir-Soar example, the new radio message would cause the removal of the subtask hierarchy, including the `push-fire-button` operator, and thus the missile would not be launched. Unlike the previous solution, the agent's internal processing is always consistent with the (known) external environment. However, this solution introduces inefficiency in the execution of a task because the agent responds aggressively to any input change, even if not relevant to the current task, and thus may need to regenerate its reasoning many times before it issues a motor action.

4.2.4 Unlimited Parallelism

In our discussion of Figure 4.1, we alluded to a potential solution for inconsistency arising from overly aggressive response to context change through the use of truth maintenance. In a truth-maintenance approach, the agent can simply keep track of dependencies and retract assertions when no longer justified. Knowledge can be asserted in a low level of the hierarchy before the hierarchical context is fully elaborated. The TMS determines if when particular assertions in a thread of reasoning are no longer supported due to changes occurring in a possibly different thread of reasoning. At first glance, such processes appear to offer low-cost parallelism throughout the architecture, allowing the agent to simply apply all matching knowledge simultaneously. We call this solution "Unlimited Parallelism" because knowledge is applied in parallel throughout the hierarchy, without any consideration of the dependencies between different threads of reasoning. The agent relies on the TMS to monitor the logical dependencies between the assertions and resolve inconsistencies between them. As we mentioned previously, Unlimited Parallelism is the solution used in the current Soar architecture.

Unlimited Parallelism fails in some cases. For instance, assume we are using Soar without any of the modifications we described in the previous chapter (i.e., using Fixed Hierarchical Justification as a solution to inconsistency arising from persistence). If an assumption is created that depends on an element that is changing in the higher context, the assumption will persist even after the change.[2] As we described in the previous chapter, nonmonotonic assumptions are necessary for many agent domains. Thus, Unlimited Parallelism is not sufficient for guaranteeing the consistency of unjustified assumptions with respect to changes in the hierarchical context.

As we described in the previous chapter, assumptions can be partially justified with respect to higher levels of the hierarchy. However, consider the use of Dynamic Hierarchical Justification and Unlimited Parallelism together. Unlimited Parallelism allows all applicable knowledge to be asserted simultaneously and relies on the TMS to resolve any resulting inconsistency *after* the application of the individual assertions. For an entailment, this can lead to a some inefficiency because the prior assertion is retracted. However, Dynamic Hierarchical Justification reacts to inconsistency by withdrawing not only the inconsistent assumption but all the assertions in the subtask. Therefore, Dynamic Hierarchical Justification used in conjunction with Unlimited Parallelism may lead to significant inefficiencies.

Unlimited Parallelism will also fail to circumvent inconsistency when output commands are issued. Output represents another source of nonmonotonic change, albeit indirect, in the agent's state. Regardless of whether an output action is generated through entailment or assumption, the agent generally cannot simply retract an output action (e.g., launch missile). Further, even when a command can be undone (e.g., opening the gripper), the agent requires knowledge to "undo" the action (close gripper) rather than simple retraction.[3] Even using Dynamic Hierarchical Justification, the output command can be issued before the inconsistency is apparent and the subtask retracted, thus leading potentially to irrational behavior.

Unlimited Parallelism is thus only a partial solution to inconsistency arising from multiple threads of reasoning. Like Fixed Hierarchical Justification, it prevents the occurrence of many inconsistencies and relies on additional knowledge to avoid inconsistency in the cases not resolved by architectural means. However, also like Fixed Hierarchical Justification, the agent's ability to maintain consistency in situations is limited by the knowledge design. When the knowledge fails to recognize inconsistency, irrational behavior can result.

4.2.5 Summary of Potential Approaches

Table 4.2 summarizes the approaches we have considered thus far, evaluating them by the criteria we introduced earlier. Knowledge-based consistency and Unlimited Parallelism depend on the agent's knowledge to avoid inconsistency, thus increasing the cost of building agents using these solutions and compromising their reliability. On the other hand, additional knowledge is not required in the approach that disallows changes to the

[2]More correctly, the assumption will persist unless the assertion that was retracted is a member of the fixed support set of the subtask.

[3]In this discussion, we assume an output system through which the agent directly initiates motor commands. However, some of the problems in behavior resulting from inconsistency due to overly aggressive response could potentially be solved by increasing the sophistication of the output system. For example, consider an output system that buffered all output commands for some delay. It might assume that the retraction of an assertion representing the command indicated an interruption of the activity and remove the motor command from the buffer. The motor command would thus be interrupted before irrational behavior was generated externally.

Approaches	**Evaluation Criteria**			
	Additional knowledge cost	Reduced Efficiency	Reduced Responsiveness	Single thread of reasoning within subtask
Knowledge-based Consistency	Yes	No	No	No
Disallowing Context Change	No	No	Yes	No
Aggressive Response to Context Change	No	Yes	No	No
Unlimited Parallelism	Yes	No	No	No

Table 4.2: Summary of potential approaches arising from overly aggressive response to context change, based on the evaluation criteria described in Section 4.2.

context nor in the aggressive response solution. However, both of these approaches lead to excessive performance costs. Thus, no approach we have examined meets all of our evaluation criteria.

4.3 Subtask-limited Reasoning: Limiting Reasoning to a Single Subtask

As we saw in the description of Unlimited Parallelism, full parallelism leads to inconsistency. However, one potential solution to inconsistency arising from overly aggressive response to context changes might be to limit parallelism, but less extremely than eliminating it altogether. The key issue is determining how to limit parallelism efficiently.

In this section, we describe Subtask-limited Reasoning, a solution that limits reasoning to individual subtasks. Inconsistency arises when a local thread of reasoning is pursued within a subtask while the hierarchical context is being elaborated simultaneously in a different thread of reasoning. Subtask-limited Reasoning avoids expensive computation of dependencies between assertions by sequencing the reasoning between subtasks. This process is similar to the "washing" mechanism used in PRS (Durfee, 1998). Reasoning across subtasks is serialized, progressing from the top of the hierarchy to the lowest level of the hierarchy. Thus, Subtask-limited Reasoning solves across-level inconsistency arising from multiple threads of reasoning by ensuring that the threads of reasoning in the hierarchical context have been fully elaborated before local threads of reasoning can progress. We further explore Subtask-limited Reasoning and discuss the implementation below. In the next section, we consider the role the task decomposition itself has on the costs and benefits of Subtask-limited Reasoning.

Consider the TacAir-Soar example we introduced above. Assume the processing of radio messages occurs as part of the patrol subtask (thus in the highest level of the hierarchy), as we saw in the example decomposition in Figure 3.4. As before, consider what happens if the radio message indicating the target is friendly occurs simultaneously with the act of launching a missile. Under Subtask-limited Reasoning, because the thread of reasoning that processes the radio message occurs higher in the hierarchy, it is computed first. The thread that actually fires the missile occurs in a lower level and is thus arrested. The radio message is parsed, leading to the classification of the current target as a friendly aircraft. The `intercept` subtask depends on a hostile target. Thus, the radio message leads to the removal of the `intercept` subtask and the thread that would have launched the missile is interrupted.

Subtask-limited Reasoning is a heuristic solution. Another way to limit parallelism while still supporting multiple threads of reasoning would be to determine the dependencies between different pieces of knowledge and fire in parallel any non-dependent assertions, regardless of where they appeared in the hierarchy. However, the determination of the dependencies between threads of computation is a hard problem (Almasi and Gottlieb, 1989; Kuhn and Padua, 1981). Subtask-limited Reasoning eliminates this computation by delaying *all* assertions in lower levels of the hierarchy, regardless of whether or not they are dependent on the reasoning causing the delay. For example, in the TacAir-Soar example, the radio message could be a confirmation that the target was hostile, or even a message wholly unrelated to the current tactical situation. Regardless, under Subtask-limited Reasoning and the decomposition we have described, the agent will delay firing the missile while elaborating the message.

4.3.1 Implementing Subtask-limited Reasoning

The implementation of Subtask-limited Reasoning was simple and inexpensive. In Soar, the creation of each new assertion includes a "match goal." Match goals correspond to subtask goals in our discussion; therefore, the identification of the subtask to which an assertion is attributed can be computed by simply inspecting the match goal of the assertion. As new assertions are instantiated, the architecture simply sorts the instantiations according to their match goals. Sorting the instantiations allows the architecture to determine the highest level of activity by examining the first assertion in the list. Any assertions matching in this first level are allowed to fire, the rest are left on the list of assertions for another pass of rule firings.[4] Of course, changes made by the initial round of rule firings will change the members on the instantiation list, adding some and removing others. The only non-constant time process in Subtask-limited Reasoning is thus sorting the assertions. However, the sort requires simply identifying each assertion's match goal, which is bound by $O(n)$, where n is the number of assertions (assuming a count sort). Additionally, the number of new assertions ready to fire at any one time is usually small. Thus, Subtask-limited Reasoning adds little additional computational cost to the agent's processing.

4.4 Influence of the Task Decomposition on Subtask-limited Reasoning

As we discussed above, Subtask-limited Reasoning can cause delays in reasoning even when the delays are unnecessary. A delay is unnecessary when the changes in the context are independent of the local thread of reasoning. In this section, we consider how problematic these delays may be. Again, as we observed in the last chapter, the task decomposition itself suggests that Subtask-limited Reasoning will cause few delays in actual reasoning except when delays are truly necessary.

As we described in Chapter 2, each step in the decomposition identifies important features of the external state necessary for the execution of a task. In the Blocks World, for example, the observation that the bottom block of a tower that needs to be constructed is not already on the table is represented by the `put-on-table` subtask. The subtask acts as a shorthand representation of potentially complex state features. Other subtasks, such as `pick-up` and `put-down`, can refer directly to `put-on-table` and ignore the specific features in the external state represented by `put-on-table`.

As long as the features supporting the individual subtasks are not changing, the subtasks themselves need not make further local computations. For instance, the `put-on-table` subtask, once created, simply leads to further decomposition. Little local computation is necessary as long as the bottom block of the tower is not on the table. No new local reasoning is necessary when the block is initially moved by the gripper, for example. Because local computations are not often necessary once a subtask is created, most of the activity in the agent is concentrated in the lowest level of the hierarchy. There, the current de-

[4]Soar does not immediately retract entailments but uses a "lazy retraction," a process in which entailments are retracted in parallel with new assertions. Lazy retraction affected the implementation slightly. The architecture also sorts a retraction list according to the match goal and makes retractions only in the current "active" level of the hierarchy.

	Evaluation Criteria			
Approaches	Additional knowledge cost	Reduced Efficiency	Reduced Responsiveness	Arbitrary Knowledge Selection within subtask
Knowledge-based Consistency	Yes	No	No	No
Disallowing Context Change	No	No	Yes	No
Aggressive Response to Context Change	No	Yes	No	No
Unlimited Parallelism	Yes	No	No	No
Subtask-limited Reasoning	No	No	**???**	No

Table 4.3: Qualitative evaluation of potential approaches, based on the evaluation criteria described in Section 4.2.

composition of the task serves to reduce the search for the next step in the decomposition (or a primitive action). Thus, because activity is already focused primarily on a single subtask, Subtask-limited Reasoning introduces little delay.

Now consider what happens in the situation we described in Figure 2.6. In this example, **block-3** suddenly moves. The agent observes that **block-3** is now on the table at the same time it is prepared to initiate a `move-down` command that will attempt to put **block-2** in the same space on the table. In the agent, the change in the external world leads to change in the highest level of the hierarchy and the eventual removal of the `put-on-table` subtask because the subtask is no longer consistent with the external situation. Subtask-limited Reasoning delays the application of the assertions in lower levels of the hierarchy while these computations are being made. In this situation, the delay was necessary because the change in the context led eventually to the removal of all the subtasks in the hierarchy.

These examples suggest that delays introduced by Subtask-limited Reasoning will occur primarily when the external state has changed enough that new reasoning is needed to determine if some subtask is still the right choice in the current decomposition. When such computations are necessary, all reasoning in lower levels of the hierarchy will be arrested until the local computations are made. In this situation, however, *not* delaying reasoning lower in the hierarchy is the exact circumstance in which inconsistency can result. If the subtask is still relevant, the subtasks will renew the local reasoning after the context is elaborated. If the subtask is no longer relevant, then the subtask will be interrupted, thus avoiding the potential inconsistency. Thus, again, we expect the structure of the decomposition itself will allow us to use the heuristic simplification of the dependency computation represented by Subtask-limited Reasoning with little additional delay in the actual execution of hierarchically-decomposed tasks.

4.5 Summary

In this chapter, we have shown that inconsistency arising from an agent's overly aggressive response within a subtask to context change can be avoided in a number of different ways. Our emphasis in exploring this problem was to find efficient, *process-based* solutions that would reduce the cost of knowledge design while retaining the agent's ability to pursue multiple threads of reasoning within a level of the hierarchy. Table 4.3 summarizes the results of our analysis thus far. The new approach we have introduced, Subtask-limited Reasoning, solves the inconsistency problem through architectural means. It does not require additional knowledge, is not expensive to compute, and supports multiple threads of reasoning within a subtask. Importantly, as we saw in the previous section, the sequential processing imposed by Subtask-limited Reasoning is not arbitrary. A subtask (and thus the reasoning within that subtask) is logically dependent on the hierarchical context. When assertions in the hierarchy are changing, the agent must wait to assert any dependent reasoning to avoid inconsistency, although it may be difficult to determine *a priori* if some context change is dependent. Subtask-limited Reasoning makes the heuristic simplification that *all* local reasoning is dependent on the changing context. This assumption simplifies the dependency computation but does potentially impact responsiveness by serializing the application of new knowledge across subtasks. We argued that the structure of hierarchically-decomposed tasks should cause little actual delay except when such delay is truly necessary to avoid inconsistency. However, in the next chapter, we will empirically evaluate the efficiency of our architectural solution to the inconsistency problems to determine if both Subtask-limited Reasoning and Dynamic Hierarchical Justification together allow efficient, responsive execution of agent tasks.

Chapter 5

Goal-Oriented Heuristic Hierarchical Consistency: An Empirical Evaluation using Soar

Supposing is good, but finding out is better.
– Mark Twain

In Chapters 3 and 4, we presented heuristic solutions to inconsistency arising from persistence and multiple threads of reasoning respectively. We also argued that the structure of hierarchically decomposed tasks gave us reason to expect these heuristic solutions would provide complete solutions to the inconsistency problems (i.e., require no cross-level consistency knowledge), conserve the agent's general capabilities, be efficient, and be responsive to the outside world.

In this chapter, we offer some empirical results supporting these expectations. We have added the Goal-Oriented Heuristic Hierarchical Consistency solution (GOHHC) to the Soar architecture. We call this version "Soar 8," to distinguish it from the current, unmodified version of the architecture, Soar 7. To evaluate Soar 8, we use quantitative measures of the qualitative evaluation criteria we have used in previous chapters: knowledge cost, performance efficiency, and responsiveness. Because the absolute values of these quantities are strongly influenced by both the task and the knowledge encoded in the agent, we make relative comparisons to agents implemented in the pre-existing Soar architecture (i.e., Soar 7). We consider agents in the two domains we have used for illustration in the previous discussion: the Blocks World and TacAir-Soar.[1] Our results show that overall efficiency and knowledge cost improve under Soar 8, while responsiveness sometimes declines. In the next chapter, we show that the decrease in responsiveness can be reversed through learning, in addition to further improving overall efficiency.

5.1 Methodological Overview

How should we evaluate Goal-Oriented Heuristic Hierarchical Consistency as embedded in Soar 8? Before discussing what we expect to learn from these evaluations, we discuss how we will address three methodological issues related to making empirical evaluations: relative *vs.* absolute evaluation, degrees of freedom in agent design, and the choice of

[1]We use a reduced-knowledge version of TacAir-Soar, "Micro-TacAir-Soar." See Section 5.4 for more details.

representative tasks.

5.1.1 Relative *vs.* Absolute Evaluation

The first choice we must make in evaluating Soar 8 is to determine what constitutes "good" and "poor" cost and performance evaluations. In general, an absolute evaluation of performance and cost is difficult because the task itself, in addition to the agent's knowledge and architecture, determines the overall cost and performance results.

Our methodological approach focuses on relative comparisons, rather than absolute comparisons. We will compare the cost and performance of Soar 8 agents to agents using the current implementation of Soar, which uses Fixed Hierarchical Justification and Unlimited Parallelism. We call these agents "Soar 7 agents," to distinguish them from Soar 8 agents also implemented in Soar. This relativistic comparison solves the problem of absolute evaluation because we can use the Soar 7 agents for cost and performance benchmarks. In other words, we can assess Soar 8 by determining if cost and performance improve or degrade relative to Soar 7 agents.

The relativistic focus of evaluation reduces the generality of the results only if our Soar 7 benchmarks are not representative of agent applications in general. Both of the following methodological commitments attempt to ensure representative benchmarks.

5.1.2 Addressing Multiple Degrees of Freedom in Agent Design

The relativistic approach we adopted above immediately raises an important issue: how can we know if our comparative results are valid and general if we have control over both the benchmark agents and new Soar 8 agents? There are many degrees of freedom in the design of an agent, even when both the architecture and task are fixed. Thus, we want to ensure that we are comparing Soar 8 results to well-designed, independent benchmarks.

We assume this methodological problem can be avoided (or at least diminished) by using systems already implemented by other researchers.[2] Such systems will be useful because they have been developed independently from this research, thus minimizing bias in the benchmark, and have been optimized for performance within Soar 7, providing a good baseline for comparison.

We used the Soar 7 systems as fixed benchmarks, and did not modify the base systems in any substantial way (although we did correct a few small errors in the agents' knowledge bases that were discovered in the course of this research). We also further constrained the Soar 8 agents by requiring them to use the identical task decompositions employed by the Soar 7 agents, and the same initial knowledge base. There were some clear opportunities to improve performance in the Soar 8 agents by modifying either the task decomposition or re-designing significant portions of the agent knowledge base. However, we chose the most conservative path possible, in order to ensure that the two agent classes remained tightly constrained, allowing fewer degrees of freedom in the course of our comparisons.

[2]The author actually participated in the development of both the Blocks World and, much less substantively, TacAir-Soar. However, his participation in these projects pre-dates the development of Soar 8. Thus, the general approach we outline should not be compromised by his earlier participation.

5.1.3 The Choice of Representative Tasks

Finally, because our resources are limited, we must choose only a few tasks to make empirical comparisons. The choice of only a few tasks or domains is a considerable drawback of benchmarks, in both Artificial Intelligence (Hanks et al., 1993) and, more generally, in Computer Science (Hennessey and Patterson, 1990). Further, based on the commitments we described above, the domains we will investigate must also have independently-developed Soar 7 agents.

We chose to examine agents in the Blocks World and in a reduced-knowledge version of TacAir-Soar ("micro-TacAir-Soar"). Although our choices were motivated primarily by the availability of domains with pre-existing Soar 7 agents, there are also additional experimental justifications for using these domains. We used the Blocks World as a test bed for quick assessment. We expected any inherent inefficiencies in Soar 8 to be immediately apparent in such a simple domain. Micro-TacAir-Soar, on the other hand, was selected as a more representative agent domain, requiring real-time response in a highly dynamic environment. Micro-TacAir-Soar is also a subset of fielded AI system, using real domain knowledge rather than the hypothetical domain knowledge for the Blocks World synthetic task. Thus, our choices reflect two extremes in a continuum of domain characteristics. We detail these justifications more specifically when we discuss the experimental methodology in each domain.

5.2 Evaluation Hypotheses

Before presenting experimental results from each domain, we now focus on the general expectations we will have in pursuing these empirical evaluations. In previous chapters, we have stressed the importance of three evaluation criteria for agents: cost, efficiency, and responsiveness. In this chapter, we explore these dimensions quantitatively. Although we will have specific expectations in different domains, can we say, in general, what differences in these dimensions can be anticipated when comparing Soar 8 agents to other agents? If these general expectations are confirmed, then we can argue that the specific results we present below should apply to additional domains as well. In the following, we consider each dimension separately, explaining the general expectation and introducing the metric(s) we will use to assess the expectation.

5.2.1 Knowledge Engineering Cost

We expect knowledge engineering effort in Goal-Oriented Heuristic Hierarchical Consistency agents to decrease in comparison to previously developed agents. Our goal in developing GOHHC was to remove the necessity of cross-level consistency knowledge in execution agents. Therefore, the removal of consistency knowledge in the Soar 8 agents should cause a reduction in the overall knowledge cost.

We will measure knowledge cost by counting the number of productions necessary for behavior in each type of agent. Each production represents a single, independent piece of knowledge; therefore, we assume that the addition of more productions represents an addition in cost. However, productions provide only a coarse metric of cost. Individual productions can vary significantly in complexity, some having many conditions and actions, others having only a few. However, as we described previously, the knowledge of each Soar 8 agent was initially developed from the previously-existing Soar 7 agent. Because the resulting changes impacted only a fraction of the total knowledge base (results below), the production metric will provide a reasonable measure of cost and avoid the complexity of making other cost estimates (e.g., person-hours might be a better metric, but much more difficult to measure reliably).

5.2.2 Performance: Efficiency and Responsiveness

In general, we expect performance to improve in Soar 8 agents, as compared to their Soar 7 counterparts. The consistency knowledge that was represented explicitly in the Soar 7 agents has now been embedded in the architecture. This architecture knowledge, as we described in the previous two chapters, uses heuristics to circumvent the expensive calculations required for a minimal complete solution. Therefore, because the consistency knowledge is efficiently embedded in the architecture, we generally expect Soar 8 agents to outperform Soar 7 agents.

There are three specific exceptions to this expectation. First, in domains where consistency knowledge is (mostly) unnecessary for task performance, Soar 7 agents may perform better than Soar 8 agents. In the Blocks World, little consistency knowledge is necessary. The world is endogenous and thus the context changes little, and only in ways the agent dictates. The Soar 8 agent, however, employs the GOHHC algorithms even though they are unnecessary in this domain. We expect relatively inefficient performance may result when the dependency calculations are not used. However, we assume the reduction in cost and performance improvement in more dynamic (complex) domains is more important than performance degradation in less complex domains.

Second, in Chapter 3, we described inefficiency resulting from regeneration. We argued that unnecessary regeneration, or the removal and regeneration of reasoning that was not logically dependent on a context change, could be avoided by using well-formed decompositions. However, in Soar 8, whenever the dependent context changes, a subtask will be retracted. If the change does not lead to a different choice of subtask, the subtask will be necessarily regenerated. This regeneration may be costly because the retraction and regeneration is potentially less specific than a knowledge-based approach. For example, a knowledge-based approach may be able to remove only the `empty` assumption when the context changes as we described in Figure 2.6. In Soar 8, the entire `put-down` subtask is removed. Thus, under Soar 8, we expect some subtask regeneration and, if the expense of regeneration grows costly in the execution of the task, a degradation in performance.

Third, in Chapter 4, we argued our solution to inconsistency arising from multiple threads of reasoning would not impact the actual efficiency of agents using simulated parallelism. However, Soar 8 agents will generate primitive operators less quickly than comparable agents using full parallelism on a parallel hardware architecture, leading to a potential decrease in responsiveness.

The expectations we described above are based a simple, single dimension of performance. We use the agent's CPU time to assess this gross measure of performance. We report the total CPU time used by the agent while executing its tasks, excluding interactions with the agent simulation and the cost of executing the simulation itself. Thus, the CPU time we report in individual experiments reflects the time the agent spends reasoning and initiating actions rather than the time it takes to carry out those actions in the environment.

We can also examine a number of architecture-specific performance measures and develop a more detailed assessment of the costs and benefits of Soar 8. Below, we describe these metrics and our individual expectations for each.

Decisions

In Soar, the selection of an operator is called a *decision*. When Soar selects an operator, it tries to apply the operator, as if it could be immediately applied (e.g., a primitive operator). Soar reaches an *impasse* when it cannot apply a newly selected operator. These non-primitive operators lead to the generation of a subgoal in the subsequent decision. For example, Soar selects the `put-down` operator in one decision and creates a subgoal to implement `put-down` in the subsequent decision.

We use the number of decisions as a relative approximation of the number of subtasks initiated for a task. We expect decisions to increase in Soar 8 agents because subtasks will be interrupted whenever a dependent change occurs. In Soar 7, a subtask was generally never interrupted until it terminated (either successfully or unsuccessfully). In Soar 8, interruptions will lead to an increase in total decisions. Further, if decisions increase substantially (meaning a large number of regenerations), overall performance will degrade.

Elaboration Cycles

Knowledge is retrieved and applied in individual *elaboration cycles* in Soar. In each elaboration cycle, any retrieved knowledge is applied in (simulated) parallel. Our solution to inconsistency arising from multiple threads of reasoning limits parallelism to the highest subtask in which knowledge can apply. Thus, we expect elaboration cycles to increase in Soar 8. The increase in elaboration cycles provides an approximation of the loss of real parallelism in Soar 8.

Production Firings

As we mentioned previously, Soar agents use productions to represent their task and domain knowledge. In general, we expect production firings to decrease in Soar 8, although in some cases production firings may increase. Production firings will decrease for two reasons. First, any inconsistency knowledge that was previously used in Soar 7 agents will no longer be necessary (or represented), so this knowledge will not be used in the execution of the task. Second, any reasoning that occurred after inconsistency arose in Soar 7 agents will be interrupted and eliminated, as we described in Chapter 4. Production firings may increase when regeneration is required as the task is executed. The change in production firings will thus be task-specific, dependent on the the combination of these factors for a particular task.

5.3 Empirical Evaluation in the Blocks World

We begin our empirical evaluation of Soar 8 in the blocks world. A snapshot from the execution of an actual Blocks World task in shown in Figure 5.1. This situation is the same one we saw in Figure 2.6. We begin by detailing our experimental methodology, and

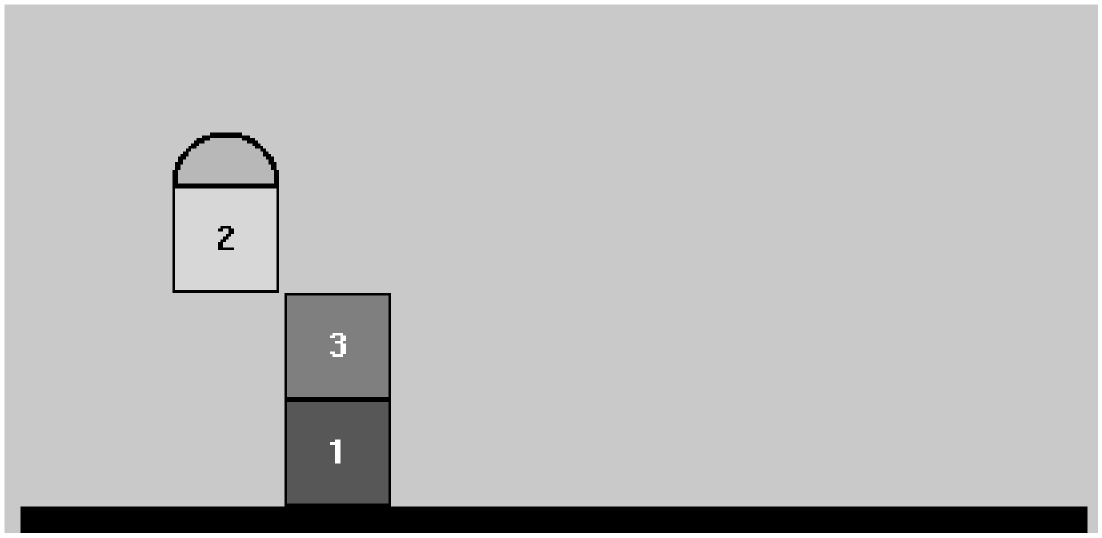


Figure 5.1: Execution of the tower building task in the Blocks World simulation. The Soar 7 and Soar 8 agents can build a three-block tower from any initial configuration of three blocks without using lookahead planning.

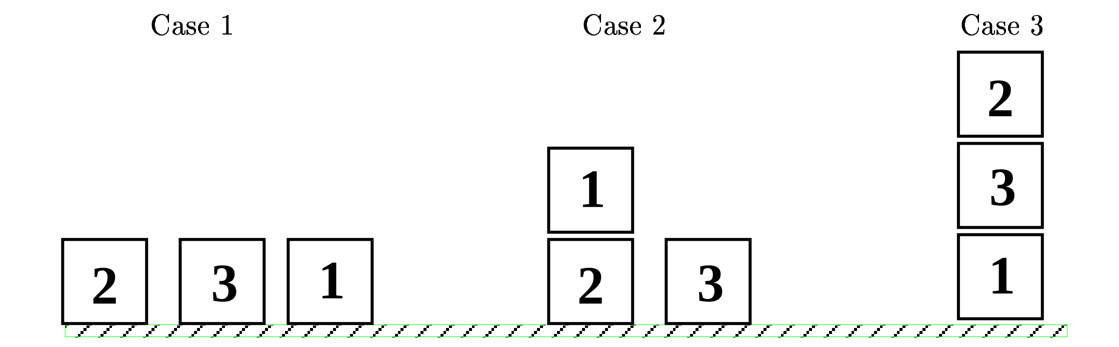
then present a summary of our results.

5.3.1 Methodology

The Blocks World is a relatively simple and wholly endogenous domain. The actual simulation we use was developed to explore issues in architecture design, although its design and implementation pre-dates the design of the Soar 8 architecture by several years. Agents in this domain have execution knowledge to transform any initial configuration of blocks into an ordered tower.

The Blocks World is a good testbed domain because any expected benefit of Soar 8 will be minimal in such a domain. An agent pays a potentially high price in performance for putting the GOHHC processes in the architecture, even though we have made an attempt to make the implementations efficient. In domains that use a lot of consistency knowledge, the cost of the new processing may be offset by the cost of the retrieving and applying the agent's explicit consistency knowledge, as we suggested above. However, the Blocks World requires little consistency knowledge. The architecture will still compute the dependencies for Dynamic Hierarchical Justification (Section 3.4.2) and sequence reasoning across subtasks (Section 4.3), even though we expect few (if any) inconsistency-causing context changes to arise due to the endogenous nature of the domain. Thus, the Blocks World should provide a lower bound on the value of the relative cost differences (because the domain requires little consistency knowledge) while also revealing any inherent efficiencies in performance.

Our Blocks World consists of three blocks, a gripper and a table the width of nine blocks. Assuming the goal is always to build the **1**-on-**2**-on-**3** tower, there are 981 unique, non-goal, initial configurations of blocks. Figure 5.2 illustrates the derivation of the individual cases. In our experiments, each agent solved each of the 981 distinct tasks.



	Case 1: Flat	Case 2: $2+1$	Case 3: Non-goal Tower
Block/space permutations:	$9!/6! = 504$	$9!/7! = 72$	$9!/8! = 9$
$\times$ Block permutations:	$\times 1$	$\times 6$	$\times 5$
Permutations per case:	504	432	45
Total permutations:			$504 + 432 + 45 = 981$

Figure 5.2: Permutations of unique, initial, non-goal configurations in the Blocks World. The derivation assumes the table is nine blocks wide and the goal is to build a **1**-on-**2**-on-**3** tower.

The order of the tasks were chosen randomly, although each agent executed the tasks in the same order. Because there is no interaction between tasks, the random order was not significant for these runs (although it will be in the next chapter when we consider learning).

The initial configuration of blocks can be described in terms of two measures of "task complexity:" the number of total blocks that must be moved during the task and the number of primitive operators executed (number of output operations). We introduce these measures of complexity because the aggregate statistics from the Blocks World runs will show a wide variation in performance statistics. However, the variation drops substantially when these complexity measures are held constant.

The number of blocks to move corresponds to the minimum number of `put-on-table` and `stack` operations necessary to execute the task. Case 1 in Figure 5.2 always requires two blocks be moved, Case 2 requires from 1 to 3 total moves, while Case 4 requires either three or four moves. In Cases 2 and 3, the same block block may need to be moved twice. For instance, in the Case 2 example in Figure 5.2, **block-1** will need to moved onto the table from its initial position, then again, when stacked on **block-2**. The number of blocks to move is determined completely by the initial situation and the agent's knowledge guarantees that the minimum number of block moves are executed.

The spatial distribution of blocks in the initial configuration and the number of blocks to move together determine the number of primitive operations or outputs necessary for solving a particular Blocks World problem. Figure 5.3 diagrams the distribution of the outputs over the initial configurations, sorted in increasing order. Recall from Figure 2.3 that the outputs in the Blocks World execution task include moving the gripper and opening and closing the gripper. Figure 5.4 shows the relationship between outputs and blocks to move. The minimum outputs necessary to complete a task increase with more blocks to move. However, the maximum outputs do not show this same relationship. In general, as the task gets more complex in the number of blocks to move, the blocks appear

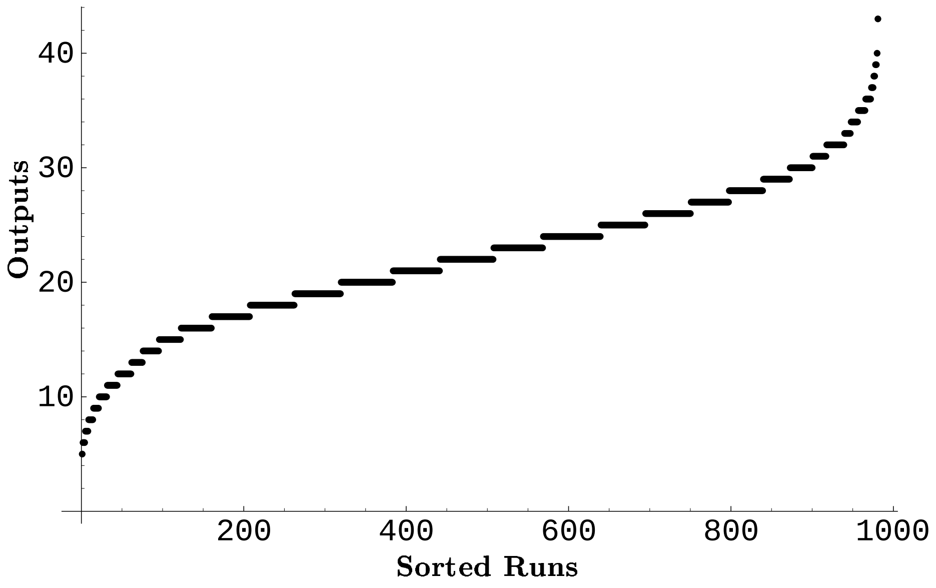


Figure 5.3: Distribution of outputs in the Blocks World.

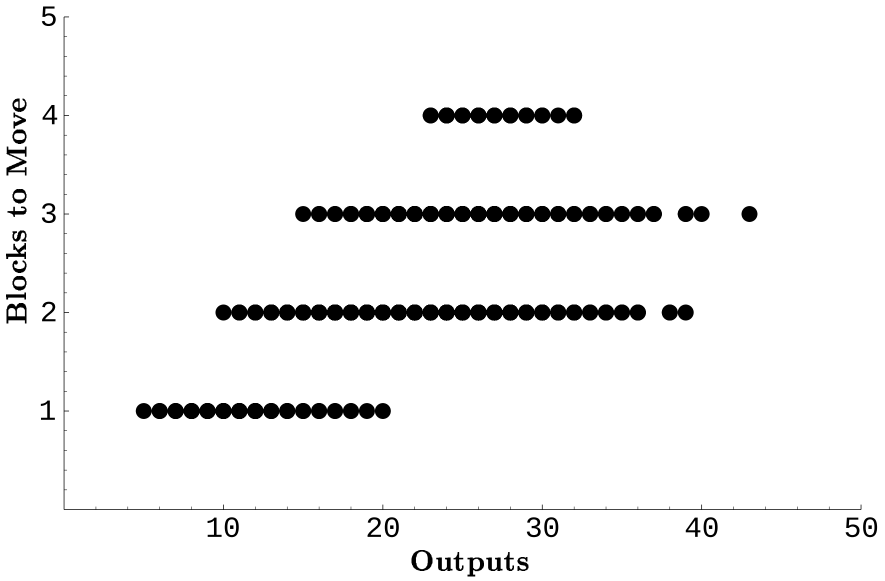


Figure 5.4: Distribution of outputs vs. number of blocks to move for the problems in the Blocks World.

	Soar 7		Soar 8	
	$\bar{x}$	s.d.	$\bar{x}$	s.d.
Rules	188		175	
Decision Avg.	87.1	20.9	141.1	38.7
Avg. Outputs	22.3	6.1	22.3	6.1
Avg. CPU Time (ms)	413.1	121.6	391.6	114.0
Avg. Elaboration Cycles	274.3	65.8	562.9	155.7
Avg. Rule Firings	720.3	153.5	855.6	199.6

Table 5.1: Summary of knowledge and performance data from the Blocks World. The agents performed the tower-building task for each of the 981 cases configurations derived in Figure 5.2. The task order was randomly determined.

closer together (e.g., in non-goal towers and two-block "mini-towers"). Thus, the most complex tasks in terms of the number of blocks to move actually require fewer outputs, on average, than cases with fewer blocks to move.

5.3.2 Results

Table 5.1 shows average data from the blocks world over the 981 cases we described above. As expected, the total knowledge necessary decreased, while overall performance improved. Decisions and elaborations increased, as expected, but the number of rule firings increased as well, the opposite result we had anticipated. In the following sections, we examine these results in greater detail to understand the impact Soar 8 has on the Blocks World agent.

Knowledge Differences

The total knowledge decreased about 7%, from 188 productions in the Soar 7 agent to 175 productions in the Soar 8 agents. This small reduction is consistent with our expectations. However, the aggregate comparison is misleading. For the Soar 8 agents, we actually both added and deleted knowledge. We deleted a total of 29 productions, representing knowledge no longer necessary under Soar 8. However, we also added 16 productions. We describe the deletions and additions below.
Removing Consistency Knowledge: Given our expectation that the blocks world would use little consistency knowledge, why were we able to delete 29 productions, about one-sixth of the original knowledge? The answer is somewhat specific to Soar. In Soar, the addition of a subtask goal is separate from the initiation of a subtask itself, as we explained when describing Soar's decision procedure (Section 5.2.2). The subtask and subtask goal are also removed separately. Recall from Chapter 3 that Soar's implementation of fixed hierarchical justification monitors impasse-causing assertions to determine if a subgoal (such as the subtask goal) should be removed. On the other hand, the removal of a subtask operator always requires knowledge (or the removal of a subtask goal higher in the hierarchy). Thus, in effect, Soar 7 treats the initiation of a subtask as a persistent assumption and requires knowledge to recognize when a selected subtask should be interrupted or terminated.

In Soar 8, we restricted assumptions to only the effects (applications) of operators.

The initiation of a subtask is now treated as an entailment and a subtask remains selected only as long as the initiation conditions for the subtask hold. This change removes the need for any consistency knowledge to terminate the subtask: when the subtask initiation conditions are no longer true, the subtask is automatically retracted. Thus, we were able to remove the termination conditions for all the subtask operators in the decomposition.
Knowledge Additions: Gaps in Domain Knowledge: The persistence of subtasks, as we described above, allows Soar 7 agents to ignore large parts of the state space in their domain knowledge. For example, the knowledge that initiates `stack` and `put-on-table` assumes that the gripper is currently not holding a block. As these tasks are executed, the gripper, of course, does grasp individual blocks. Because the subtask selection is persistent, the conditions under which a `stack` or `put-on-table` operation should be initiated when holding a block were ignored and not included in the domain knowledge.

However, the assertion that initiates the subtasks obviously becomes inconsistent with the external situation as task execution progresses. In Soar 8, when the gripper grabs the block, the architecture recognizes the inconsistency between the perceived state (holding block) and its subtasks (pursue this task when not holding a block) and retracts the subtask. The agent now needs knowledge to determine which subtasks it should choose when holding blocks. We added 16 productions, primarily for the `stack` and `put-on-table` operators, to allow the agent to recognize these states in the domain. Importantly, this knowledge is necessary domain knowledge. In the Soar 7 system, the agent could not solve any problem in which it began a task holding a block, because the domain knowledge did not cover these states. We simply added that knowledge to the Soar 8 agents. The need for these additions are thus a positive consequence of Soar 8 because the architecture's enforcement of consistency can reveal gaps in the domain knowledge during development.

Performance Differences

Somewhat surprisingly, the overall performance of Soar 8 agents (as measured in CPU time), improves slightly in comparison to the Soar 7 agents, even through each of the other performance metrics we described in Section 5.2.2 all increase. In the following, we consider each of the Soar-specific performance metrics individually, and then explain how overall performance improves.
Decisions: In Table 5.1, we saw that Soar 7 agents made, on average, considerably fewer decisions than Soar 8. Figure 5.5 plots the distribution of decisions over each individual task and shows that the increase in the number of decisions was not due to outlier differences but is consistent across all the tasks we examined. The additional decisions are due to the removal and regeneration of subtasks, which occurs when the (tested) conditions under which a subtask was selected change, as we described in the previous section. We could have modified the knowledge in the Soar 8 agents to avoid unnecessarily testing specific configurations of blocks and thus avoid the large number of subtask regenerations. Instead, however, we chose to let the architecture perform the removals and simply added knowledge to re-initiate the subtask.

As we described in Section 3.3.2, unnecessary regeneration is a potential source of inefficiency. Figure 5.6 provides a justification for not having re-designed the knowledge in the Blocks World. In the figure, we plot the number of decisions for both Soar 8 and Soar 7 as a function of the number of outputs necessary to solve individual tasks. This figure shows the structure of the task is preserved in the architectural decision processing

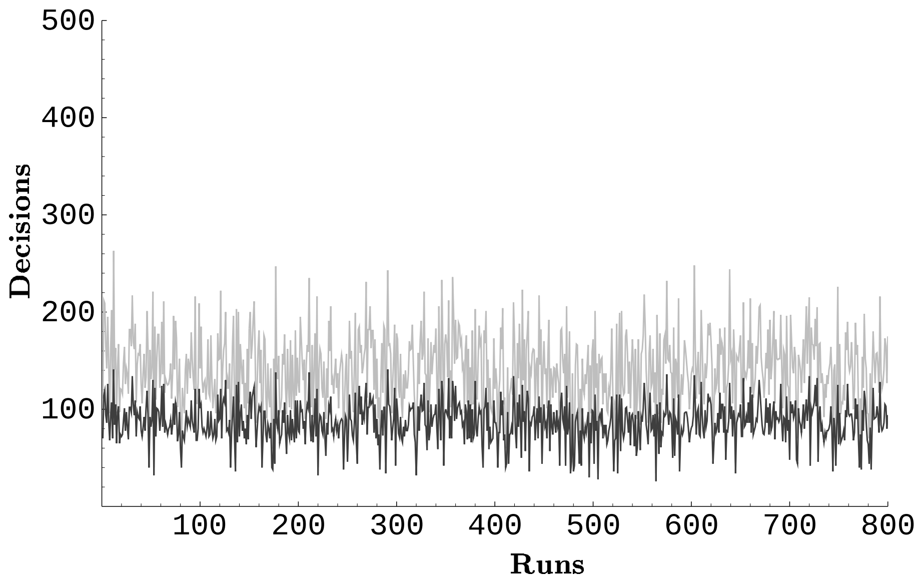


Figure 5.5: Runs *vs.* decisions for Soar 7 (black) and Soar 8 (gray). In all cases, the number of decisions increases in Soar 8.

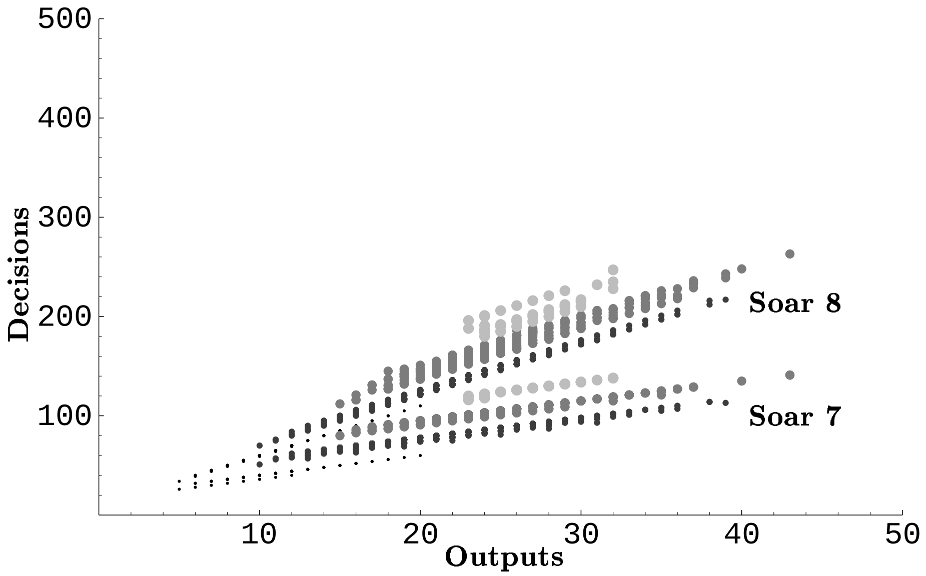


Figure 5.6: Outputs *vs.* decisions for Soar 7 and Soar 8 agents in the Blocks World. The shade and size of each datum signifies the the number of blocks moved for the run: (smallest, black) → 1 block; (larger, dark gray) → 2 blocks; (larger, medium gray) → 3 blocks; (largest, light gray) → 4 blocks.

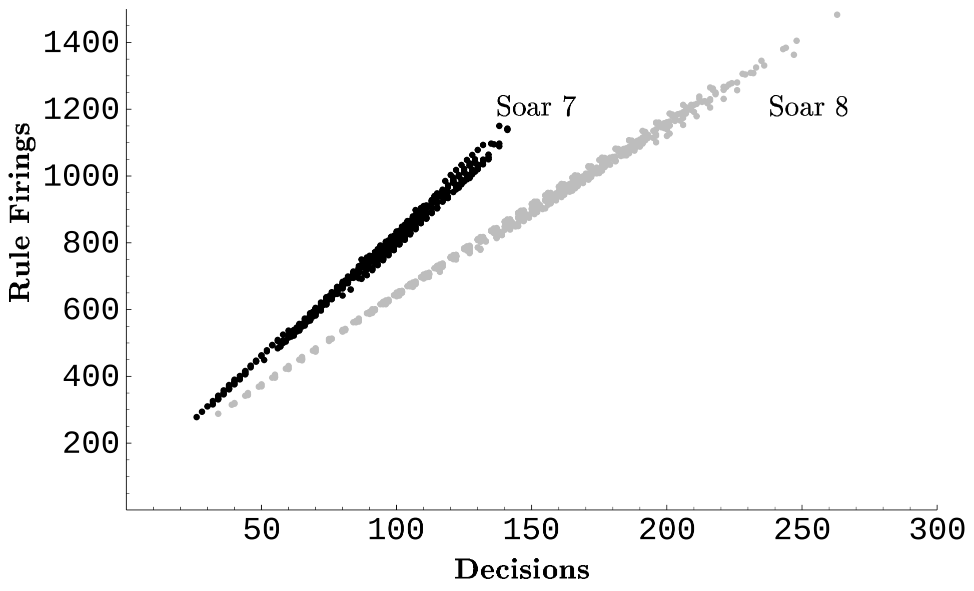


Figure 5.7: Decisions *vs.* production firings for the Blocks World agents.

in Soar 8. Although the slope of the Soar 8 lines have increased due to the regeneration, the external task complexity (outputs, blocks to move) still strongly determine the number of decisions. In a regeneration approach like we described in Section 3.3.2, we would expect the processing would not reflect the structure of the task because regeneration occurs regardless of its necessity. The preservation of the general structure of the task in the decision processing convinced us that the subtask regenerations we observed in the Soar 8 Blocks World agent were actually necessary regenerations of the subtask.

Elaboration Cycles: The average number of elaboration cycles more than doubled in the Soar 8 agents. The reason for this increase is simply the Subtask-limited Reasoning we described in Chapter 4. Although a relatively large increase, the loss of parallelism is much smaller than possible. As we argued in the previous chapter, at any point in time, we expect most problem solving activity to be focused on the lower levels of the task hierarchy, and thus the delays introduced by our solution are not as significant as they might otherwise appear. For instance, the hierarchy of subtasks in the Blocks World is as deep as five. Thus, the loss of parallelism in Soar 8 could easily be twice what we observed. The smaller number we actually measured confirms that the solution we introduced in Chapter 4 is much less limiting than it might initially appear.

Production Firings: The number of production firings also increased in the Blocks World. The increase in production firings can be attributed to the knowledge we added to the system and the resulting regeneration of the subtasks that made it necessary. However, the increase in number of production firings relative to decisions was much smaller. Decisions increased by a factor about 62% while production firings increased only by about 19%. This smaller relative increase is due to the productions that were removed (and thus did not fire) and Subtask-limited Reasoning. Figure 5.7 shows the relationship between decisions and productions firings for both Soar 7 and Soar 8. Significantly, production firings are closely correlated to the decisions. The correlation implies that production firings also are closely tied to the structure of the task (as were decisions in

Figure 5.7).[3] Thus, while our original expectation regarding production firings was not confirmed, the surprising results are explained by the addition of new knowledge, rather than any unforeseen effect of Soar 8, such as unnecessary regenerations.

CPU Time: When production firings increase in Soar, we generally expect an increase in CPU time. However, as we saw in Table 5.1, CPU time in Soar 8 decreased slightly in comparison to Soar 7 even through production firings increased. To explain this result, we have to consider some additional aspects of Soar's processing.

Different productions take different amounts of time to match and fire in Soar; production matching, especially, is not a constant time operation. In general, the match cost of a production grows linearly with the number of partial instantiations of the production (Tambe, 1991). These partial instantiations are called *tokens*. Each token indicates what conditions in the production have matched and the variable bindings for those conditions. In effect, each token represents a node in a search over the agent's memory for matching instantiation(s) of the production. The more specific a production's conditions are, the more constrained the search through memory, thus it costs less to generate the instantiation.

In the Soar 8 Blocks World agent, the productions we added were more specific to the agent's memory (i.e., its external and internal state) than the productions we removed. Further, simply having fewer total productions also reduces the amount of total search in memory (assuming the average number of condition elements is constant in the two systems).[4] An informal inspection of the match time and tokens for several Soar 7 and Soar 8 runs showed that the number of tokens consistently decreased in Soar 8 by 10-15%. This reduction in token activity is the primary source of improvement in CPU time Soar 8.

This improvement, of course, is not a general result and provides no guarantee that in some other task or domain the cost of matching will not increase rather than decrease. However, as we saw above, Soar 8 does force an agent's knowledge to be more specific to individual situations (because an assertion no longer persists over situations inconsistent with the assertion's instantiation conditions). Thus, we expect the constraint imposed by Soar 8's consistency processing will force production knowledge to be more specific, and thus usually improve overall match cost. Exploring this hypothesis is a potential direction of future work.

Responsiveness: We ignore responsiveness in the Blocks World because the domain is endogenous. The only situations to react to are the agent's own actions, which are known and and can be anticipated. However, we do offer the following observation. Soar agents poll the world for input once per decision. In the results we have discussed, the number of decisions increased in Soar 8 agents, while the total CPU time decreased. These changes reduce the decision cycle tome from 4.75 milliseconds/decision in the Soar 7 agent to 2.77 ms/decision in Soar 8. Thus, Soar 8 agents in this domain do have more opportunities to poll the world state in any given span of time, and thus are potentially *more* responsive than their Soar 7 counterparts.

[3]Figure 6.3, in the following chapter, shows the relationship explicitly.

[4]This statement is an over-simplification. The RETE algorithm (Forgy, 1979) shares condition elements across different productions. Thus, the removal of productions only decreases the total search if the removed productions contain condition elements not appearing in the retained productions. We did not perform an exhaustive analysis of the condition elements to determine if the removed productions reduce the number of unique condition elements in the RETE network, although that expectation is consistent with our results.

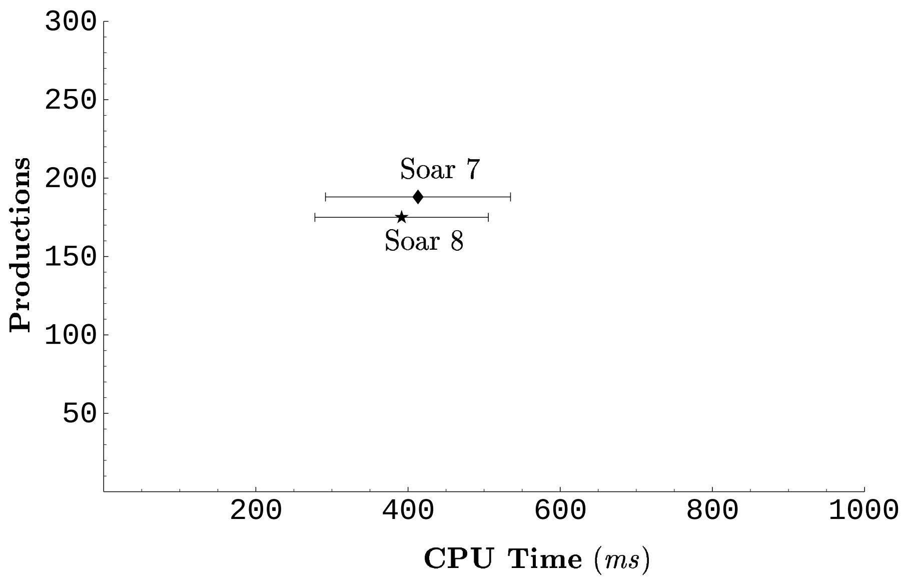


Figure 5.8: Blocks World Summary: Mean CPU Time in milliseconds *vs.* knowledge in productions for Soar 7 (diamond) and Soar 8 (star) agents. The error bars show the variation in the mean over two standard deviations.

5.3.3 Blocks World Summary

Figure 5.8 summarizes the results from the blocks world along the knowledge and performance dimensions central to our evaluation. In the figure, we see that Soar 8 improves slightly in both performance (a smaller average CPU time) and knowledge (fewer total productions). Although the gains are not large, the relatively small magnitudes of the differences were anticipated. Importantly however, Goal-Oriented Heuristic Hierarchical Consistency in the Blocks World results in these benefits to the agents:

- Provides a guarantee of consistency in processing across subtasks
- Decreases total knowledge, suggesting a decrease in total cost
- Identifies gaps in the domain knowledge
- Improves performance

Thus, Soar 8 did not degrade overall performance or increase overall knowledge costs, even in a domain where we would expect little or no gains in these dimensions. Given these positive results, we now move the empirical evaluation to a more dynamic and complex domain.

5.4 Empirical Evaluation in μTacAir-Soar

In this section, we describe our evaluation of Soar 8 agents in Micro-TacAir-Soar (μTAS for short). In the following, we describe the characteristics of μTAS that make it a good domain for assessing Soar 8, discuss some of the complications that arose when first applying Soar 8 to the domain, and then present our empirical evaluation of Soar 8 agents in the μTAS domain.

5.4.1 Methodology

TacAir-Soar would be an ideal domain for assessing the impact of Soar 8. Unlike the Blocks World, it is complex, exogenous, and requires rapid agent response to the external situation. Further, TacAir-Soar agents operate at the edge of the performance capabilities of the architecture. Our expectation is that Soar 8 will advance the "state of the art" in Soar agent design, currently represented by Soar 7 TacAir-Soar agents. However, because TacAir-Soar agents are already pushing the architecture's performance envelope, any degradation in performance under Soar 8 will not only be observable, but may also lead to qualitative degradation in the execution of the task.

TacAir-Soar is a very large system, consisting of over 5000 productions, and took many person-years to develop. The conversion from Soar 7 to Soar 8 agents would potentially require many months of effort. Therefore, we decided to evaluate Soar 8 in a research version of TacAir-Soar, "Micro-TacAir-Soar." μTAS agents use the TacAir-Soar simulation environment (ModSAF) and interface but cannot fly the complete range of missions available in the full system. However, μTAS uses the same tactics and doctrine as TacAir-Soar for its specific missions. In the restricted domain, a team of two agents ("lead" and "wing") fly a patrol mission and engage any hostile aircraft that meet their commit criteria (are headed toward them, and are within a specific range), as we described previously (Figure 3.3). The lead agent's primary role in the mission is to fly the patrol route and intercept enemy planes. The wing's primary responsibility is to fly in formation with the lead. μTAS agents require an order of magnitude fewer productions than TacAir-Soar agents. Because the total knowledge is significantly reduced but the domain retains the complexity and dynamics of the TacAir-Soar domain, converting μTAS agents should be relatively inexpensive, while our results should be representative of results in TacAir-Soar as well.

A patrol mission has no clearly-defined task termination condition, like having built a tower in the Blocks World. Therefore, we run each agent in the simulation for ten minutes of simulator time. During this time, each agent has the opportunity to take off, fly in formation with its partner on patrol, intercept one enemy agent, and return to patrol after the intercept. In an actual TacAir-Soar scenario, these activities would normally be separated by much larger time scales. However, an agent spends much of its time on a patrol mission simply monitoring the situation (waiting), rather than taking new actions. We experimentally determined that ten minutes of simulated time is a short enough time period that overall behavior is not dominated by wait-states, while long enough that individual scenarios retain a natural flow of events.[5]

When running for a fixed amount of time, an increase in the number of decisions can be attributed to either regeneration or simply an improvement in decision cycle time (which we expect, given our Blocks Worlds results). In the Blocks World, an increase in

[5]For example, in the ten minute scenarios, the agents were not forced to reason about enemy planes while taking off. In shorter scenarios, the enemy planes began very close to the agent's take off point and sometimes the agents would simultaneously be taking off and determining if the agent should intercept enemy aircraft. The ten minute scenario allowed the agents to complete one patrol pass before encountering any enemy agents, which was more consistent with the scenarios for which they were originally designed.

the number of decisions was indicative of regeneration because the task always progressed until the tower goal was accomplished. In μTAS, the simulator interpolates changes in the world according to a real-time clock, updating the world once per agent decision. The frequency of update in the simulation is dependent on the time required for all agents in the simulation to execute a decision. Thus, agents that execute decisions faster will have more decision opportunities in a fixed amount of time, and thus decisions may increase without regeneration. We obviate the conflation of these effects in our μTAS results by running μTAS in a fixed-time mode. In fixed-time mode, each simulator update represents 67 milliseconds of simulated time. Because we run each scenario for a fixed amount of simulated time, now we will also observe a fixed number of decisions for the Soar 7 and Soar 8 agents. We assume that any problems due to regeneration will be apparent in the number of rule firings and degradation in responsiveness. Further, we have confirmed that the general results from these scenarios do not change significantly if we run the scenarios in the variable time mode used normally for TacAir-Soar agents. The fixed-time mode simply eliminates some variability in the results, making them easier to analyze.

5.4.2 Initial Results

Our first attempt at creating Soar 8 agents was simply to use the Soar 8 architecture with the original Soar 7 agent knowledge base. Unlike the agents in the Blocks World, the Soar 7 μTAS agents use a knowledge-based convention for terminating selected subtasks when their initiation conditions no longer hold. Thus, we anticipated that we would not need to add the type of knowledge that we added to Blocks World. We expected that the original knowledge base could be improved (by identifying and removing consistency knowledge no longer necessary) but that Soar 8 agents could perform using this knowledge.

However, these initial agents suffered from severe regeneration when performing the scenario we described above. This regeneration resulted in significant increases in the number of rule firings and similar increases in total CPU time. For instance, the lead agent's rule firings and CPU time increased by factors of 3.8 and 4.5 respectively, in comparison to the Soar 7 lead agent. Additionally, the extra cost negatively impacted qualitative behavior when run with variable-time cycles: Soar 8 agents more frequently missed their missile shots, took longer to perform the intercept, and thus exposed themselves to more risk than the Soar 7 agents.

5.4.3 Creating Better Decompositions

These results prompted us to examine closely the Soar 7 knowledge base, in order to understand the source of the debilitating regeneration in the Soar 8 agents. One obvious limitation we discovered was that the convention for terminating subtasks, which we mentioned above, was used for only one-third of the subtasks in the μTAS decomposition. Thus, in a few cases, we added knowledge similar to the knowledge added in the Blocks World, to fills gaps in the existing domain knowledge. Although this problem again pointed out one of the problems with knowledge-based solutions for consistency (i.e., incompleteness), it did not cause large problems in the overall behavior.

The biggest deficits in performance arose because the Soar 7 agent *takes advantage* of inconsistency in asserted knowledge. In other words, the Soar 7 agent not only allowed inconsistency in the assertions but actually depended on those inconsistencies to apply new knowledge. In the following sections, we examine three different ways in which this "inconsistency knowledge" was used in Soar 7 and what we had to do to change it in Soar 8. In Section 5.4.4, we summarize the changes in terms of the productions added, deleted, and changed from the original knowledge base.

Within-level Consistency Knowledge

Recall that our solutions to consistency across different levels of the hierarchy still require consistency knowledge within an individual subtask. This knowledge is necessary in both Soar 7 and Soar 8, and serves primarily to "clean up" the subtask goal. That is, when terminating a subtask, the agent deliberately removes local assertions that contribute to the execution of the terminating subtask, to avoid the (mis)use of these assertions in later execution.

As an example, consider the `achieve-proximity` subtask. This operator is used in a number of different problem spaces when an agent needs to get closer to another agent. For instance, if the wing strays too far from the lead, it may invoke `achieve-proximity` to get back into formation with the lead. The lead, on the other hand, uses `achieve-proximity` to get close enough to an enemy aircraft to launch a missile. The operator makes many local computations. For example, the agent reasons about what heading it should take to get closer. The computation depends on what information is available about the other aircraft. When the wing is pursuing the lead, it may know the lead's heading and thus pursue a collision course to maximize the rate of convergence. Sometimes the other agent's heading is not available; in this case, the agent simply heads toward the current location of the other agent. These local computations are stored in the local subtask. When the `achieve-proximity` operator is terminated, the agent removes the local structure. Removing the structure is important both because it interrupts entailments of the local structure no longer necessary (e.g., calculation of the current collision course) and guarantees that if the agent decides to `achieve-proximity` with respect to a different aircraft, that supporting data structures are properly initialized. This knowledge thus maintains consistency in the local goal by removing the local structure when the `achieve-proximity` subtask is no longer selected.

We initially assumed this within-level consistency knowledge would need no modification. However, when using the convention-based approach, the Soar 7 agent could recognize when it was going to remove a subtask. Subtask removals do not occur immediately but rather at the end of the decision in which the termination knowledge applies. In the meantime, the termination condition could act as a signal to the within-level consistency knowledge. For instance, the knowledge that removes the local structure for `achieve-proximity` can be summarized as follows: "if the `achieve-proximity` operator is selected, but its initiation conditions no longer hold, then remove the local `achieve-proximity` data structure." Thus, the Soar 7 agent uses a recognition of inconsistency in the assertions to trigger the activation of the within-level consistency knowledge.

In Soar 8, of course, when the subtask's initiating conditions are no longer supported, the selected subtask is removed immediately. Thus, the unmodified within-level consistency knowledge in the Soar 7 agent never has an opportunity to apply. The failure to apply this knowledge, however, led to local inconsistencies. If any of these inconsistent assertions were shared across subtasks, the architecture would respond by retracting the subtask. Thus, the architecture was regenerating subtasks for both local and across-goal inconsistency, resulting in many more regenerations than we expected.

We were able to solve this problem very generally by making the creation of the local structure an entailment of the initiation conditions of the subtask itself. Thus, when the subtask initiation conditions were no longer true, both the subtask selection and the local structure would be removed immediately and architecturally. This change thus also obviated the need for some within-level consistency knowledge. The cost of this solution is that the local data structure may need to be regenerated if the `achieve-proximity` operator is temporarily displaced. For instance, the Soar 7 within-level consistency knowledge could determine under what conditions the local structure should be removed. The Soar 8 solution has lost that flexibility.

Subtasks with Complex Actions

Soar 7 allows the agent to execute a number of actions in rapid succession, regardless of any inconsistency in the local assertions. For example, a single subtask operator can be initiated in a situation representing the conditions under which to apply the first step in the sequence, and terminated when the last step in the sequence has applied. If some intermediate step invalidates the initiation conditions, the subtask can still progress through its sequence. Worse still, the individual steps may not even be logically dependent on similar subsets of the hierarchical context, thus contradicting the assumption we made in Chapter 3 about the near decomposability of subtasks in the hierarchy. The failure of the architecture to enforce consistency allows arbitrarily complex procedures to be executed by a single subtask operator.

As a concrete example, consider the process of launching a missile. The actual missile launch requires only the push of a button, assuming that previous steps such as selecting the target and an appropriate missile have been accomplished beforehand. After pushing the fire button, the pilot must fly straight and level for a few seconds while the missile rockets ignite and push the missile into flight. Once the missile has cleared the aircraft, the agent "supports" the missile by keeping radar contact with the enemy plane. In Soar 7, the `push-fire-button` subtask includes both the act of pushing the fire button, and the act of counting while the missile clears the aircraft. However, these tasks have different and mutually exclusive dependencies. The initiation condition for the `push-fire-button` operator requires that no missile is already in the air. However, the subsequent waiting requires counting while the newly launched missile is in the air.

Soar 8 always removes the `push-fire-button` subtask as soon as the missile is perceived to be in the air. Regeneration occurs because the agent never waits for the missile to clear and thus never realizes that the missile just launched needs to be supported. In the initial scenarios we ran, the Soar 8 agent would fire all the available missiles at the enemy plane, one after another.

We viewed this problem as a failure in the original decomposition to identify different subtasks. Pushing the fire button and waiting for the missile to clear are independent tasks, which happen to arise in serial order in the domain. We enforced this independence by creating a new subtask, `wait-for-missile-to-clear`, which depends only on having a newly launched missile in the air. The Soar 8 agent now pushes the fire button, selects `wait-for-missile-to-clear` to count a few seconds before taking any other actions, and then correctly supports the missile if it clears successfully.

This solution reduces regeneration and improves behavior quality but it does have a non-trivial cost. Whenever we split an operator, the effects of the operators no longer occur in rapid succession within a decision. Instead, the effect of the first operator occurs in one decision, the effect of the second operator in the second decision, etc. Thus, this solution can negatively impacts responsiveness. We will see concrete evidence of this cost in Section 5.4.4.

Global* vs. *Local Assumptions

In Figure 3.4, we showed the agent computing a new heading as a subtask of the `achieve-proximity` operator. This calculation usually depends upon the current heading. However, when the agent generates the command to turn, the heading changes soon thereafter. In Soar 7, this change is not problematic. In Soar 8 because the desired heading depends upon the current heading, the subtask that generated the command will be retracted when the heading changes, leading to many (unnecessary) regenerations of the same motor command. In our initial experiments, most regeneration resulted because individual motor commands were stored locally, in a subtask, and whenever the world changed, the subtask would be retracted and regenerated.

The agent's knowledge is treating a global value (the motor command) as a local value. The motor command is really not local to the subtask because it can (and often will) lead to changes throughout the context. For instance, the wing agent, whose job is to follow the lead agent in a particular formation, may initiate a turn when it realizes that the lead has begun to turn. Once the wing begins to turn, it will want to use the motor command (desired heading) to determine if further course changes are necessary. If the course correction is local, however, the wing cannot utilize this knowledge at higher levels.

The highest level of the hierarchy serves as a global state because assertions in it can be inspected by subtasks in any level. Additionally, assumptions in the highest level of the hierarchy are unjustified (there can be no assertions in higher levels on which they depend). Therefore, the global level also represents a level in which all assumptions are maintained wholly by within-level consistency knowledge. The Soar 8 agents were changed such that they now issue motor commands at the global level, rather than locally. The agent now has access to motor commands throughout the hierarchy and can make use of them in specific subtasks without impacting local dependency calculations. No unnecessary regeneration occurs because the agent always has access to the current motor command and thus generates a new one only when the motor command would be different. The solution, of course, requires consistency knowledge because the motor commands are unjustified and thus must be explicitly removed. However, in this specific case, the agents always replaced old motor commands when generating new ones, so no additional consistency knowledge was necessary. In general, however, making a value global necessitates consistency knowledge to manage it.

5.4.4 Results

We made modifications to the Soar 8 agents' knowledge base as described in the previous section and then ran the scenario in order to compare Soar 7 and Soar 8 agents. In the Blocks World, there was virtually no variation in the statistics when a particular task

	Lead Agent				Wing Agent			
	Soar 7		Soar 8		Soar 7		Soar 8	
Rules	591		539		591		539	
Decisions ($\bar{x}$, *s.d.*)	8974	0.0	8974	0.0	8958	0.0	8958	0.0
Outputs ($\bar{x}$, *s.d.*)	109.1	6.71	142.8	7.03	1704	42.7	869	12.8
ms of CPU Time ($\bar{x}$, *s.d.*)	1683	301	1030	242	12576	861	2175	389
Elaboration Cycles ($\bar{x}$, *s.d.*)	1590	96.9	1991	109	11820	296	7246	130
Rule Firings ($\bar{x}$, *s.d.*)	2438	122	2064	81.1	16540	398	6321	104
Number of runs (n)	43		53		56		46	

Table 5.2: Summary of μTAS run data for a scenario in which a lead and wing fly a patrol mission, intercept an enemy plane, and return to patrol following the intercept. Each scenario ran 10 minutes. The data in each column was averaged over the number of runs in the bottom row, about fifty runs.

was repeated. Although we controlled for as much indeterminism as possible, the μTAS simulator is inherently stochastic. To control for variation due to this indeterminism, we ran each scenario about fifty times. Table 5.2 lists average data for the Soar 7 and Soar 8 lead and wing agents for the patrol/intercept scenario. The results in this domain are consistent with all our expectations: total knowledge decreases, elaboration cycles increase, and rule firings decrease, substantially so in the Soar 8 wing. In the following sections, we examine each of these results in greater detail.

Knowledge Differences

Table 5.3 quantifies, by category, the changes to the Soar production rules we described in Section 5.4.3.[6] Modifications include deletions, additions and changes. We regarded a rule changed only if its conditions changed slightly, but it made the same type of computation (entailment or assumption) for the same subtask or in the same problem space. For example, most of the within-level consistency knowledge requiring modification (rather than deletion) is categorized as "changed." The knowledge now refers to a different structure but that structure is located in the same problem space. This somewhat restrictive definition of a change inflates the addition and deletion accounting. In many cases a production was "deleted" and then immediately "added" to a different problem space. For example, the productions that manipulate motor commands all were moved from local subtasks to the highest subtask level. Almost all the additions and deletions in the "Globalization" category can be attributed to this type of change, which, in reality, required no synthesis of new production knowledge.

The total overall knowledge required for the Soar 8 agents decreased, as it did in the Blocks World. This reduction of about 9% was achieved by making some type of modification to about 40% of the Soar 7 rules, and may seem like a very modest gain,

[6]This table also includes the changes that were made for learning. As we describe in Section 6.3.3, these changes were not necessary for correct learning, but made newly learned productions more general. The Soar 8 performance data we present in this chapter were generated with a knowledge base that included the learning changes. Therefore, we include the learning changes here for completeness. However, the presence of these rules in the knowledge base has no impact on the non-learning performance data we report in this chapter.

	Subtask	Within-level Consistency	Complex Subtasks	Globalization	Learning	Miscellaneous	TOTALS
Soar 7 Agent:							591
Deletions:	44	9	10	36	4	8	-111
Additions:	0	5	21	32	0	1	+59
Changes:	0	8	0	33	24	0	(65)
Soar 8 Agent:							539

Table 5.3: Quantitative summary of changes to production rules in the Soar 7 agent knowledge base for Soar 8 agents. The modification categories are described in Section 5.4.3 (changes specific to learning are described in Section 6.3.3).

given the conversion cost. However, this conversion cost is an artifact of the methodology we have chosen. Had we built the Soar 8 agents in this domain without having Soar 7 agents to which to compare them, we would expect a 9% decrease in the total knowledge, thus reflecting our goal: a reduction in the cost of the agent design. We detailed the Soar 8 modifications in the thesis because they highlight how the Soar 7 agent failed to maintain consistency among its assertions, even though the overall behavior of the system was not compromised. This failure to ensure consistency will be important in the next chapter, when we consider how to further improve performance when using hierarchical task decomposition. The conversion cost does suggest that converting a much larger system, like TacAir-Soar, may indeed be costly. In this case, however, it is significant that the modifications we describe were all pointed out by identifiable regenerations in the architecture. Thus, on a relative scale, the 235 total changes we made to the Soar 7 knowledge base were much easier to make than constructing a similar number of rules in a new agent.

Performance Differences

As the performance results in Table 5.2 show, Soar 8 agents improved in performance relative to their Soar 7 peers. However, the difference in performance improvements between the lead and wing agent was substantial. In the following, we explain the task differences in the leads and wings that led to the difference in relative improvements. We then individually evaluate the Soar 8 agents' performance along the elaboration cycles, production firings and CPU time metrics.

Lead and Wing Agents: The lead and wing agent share the same knowledge base but perform different tasks in the μTAS scenario, which leads to differences in their absolute performance. Recall that the lead's primary responsibility is to fly the patrol route and lead the intercept. Thus, high-level subtasks for the lead include `fly-patrol-route`, `intercept`, etc. On the other hand, the wing's primary mission role is to follow the lead. The wing spends most of the scenario executing a `follow-leader` subtask. These different tasks require different responses in the agent. We assume that an agent's overall

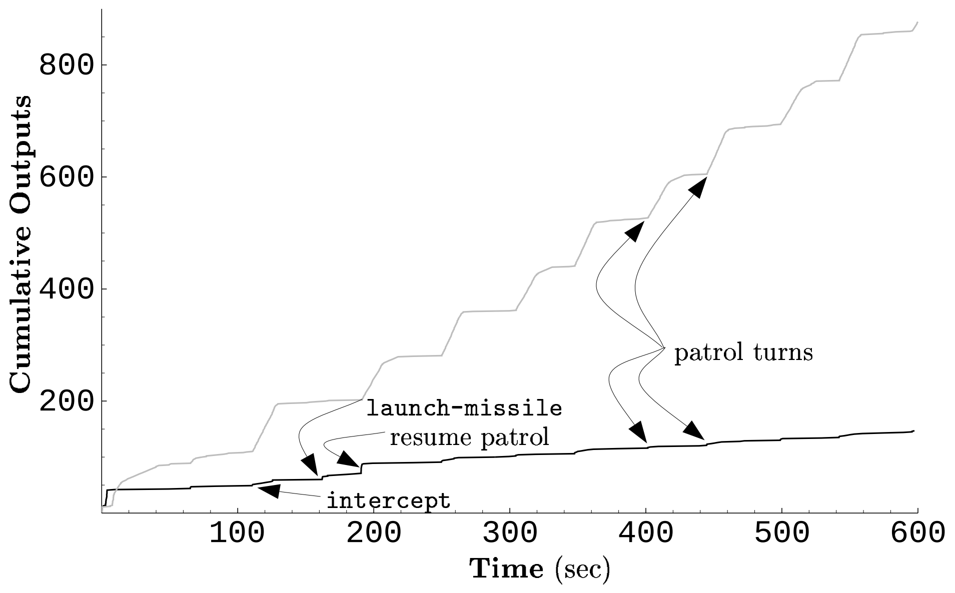


Figure 5.9: Cumulative outputs over the course of one ten minute scenario for Soar 8 lead (black) and wing (gray) agents. The lead's output activity is mostly concentrated at a few places over the course of the scenario (`take-off`, `intercept`, `launch-missile`, and resume patrol). The wing's most concentrated output activity occurs when the lead turns to a new leg of the patrol and the wing must follow the lead through a 180 degree turn.

reasoning activity is correlated by its output activity.[7] Figure 5.9 summarizes the output activity of one pair of lead and wing agents over the course of a ten-minute scenario. We will now examine the behavior of the lead and wing agents more closely to explain the differences in output activity, which, in turns, suggests an explanation of the difference in other performance metrics.

As Figure 5.9 suggests, the lead actually spends most of the scenario waiting, with short bursts of reasoning and output activity occurring at tactically important junctures in the scenario. On patrol, the lead flies straight and makes a decision to turn when it reaches the end of a patrol leg. While flying, the lead simply has to monitor the environment and search for enemy planes. This search is (mostly) passive; the agent's radar simply notifies the agent if any new entities have been detected. After detecting and classifying an enemy plane as a potential threat, the lead commits to an intercept. At this point, the lead immediately makes a number of course, speed, and altitude adjustments, based on the tactical situation. These actions are evident in the figure by the pulse labeled "intercept." The lead spends most of the time in the intercept simply closing the distance between the aircraft to get within weapon range, again having to maneuver very little and thus requiring few actions in the environment (thus the relatively flat slope of following the intercept). When the agent gets within missile range of the enemy plane, the agent executes a number of actions very quickly. The lead steers the plane into a launch window for the missile, pushes the fire button, waits for the missile to clear, and then determines a course to maintain radar contact as the missile flies to its target, all in a very short

[7]We provide a more concrete justification of this assumption in the section on production firings, below.

period of time, as indicated by the pulse of activity at `launch-missile` in the figure. Once the intercept has been completed, the lead resumes its patrol task. Again, it issues a large number of output commands in a short period of time, as shown in the figure. These examples show that the lead's reasoning focuses primarily on reacting to discrete changes in the tactical situation (patrol leg ended, enemy in range, etc.) and the response generally requires little continuous adjustment.

The execution of the wing's `follow-leader` task requires reaction to continuous change in the lead's position in order to maintain the formation. Position corrections require observing the lead's position, realizing an undesired separation in the formation, and then responding appropriately to the separation by adjusting speed, course, altitude, etc. Because the wing is following the lead throughout the scenario, it is executing this position maintenance knowledge almost constantly. When the lead is flying straight and level, as on a patrol leg, the wing's task does not require the generation of many outputs. In Figure 5.9, these periods of little activity are evident in the periodic flat segments in the wing's cumulative outputs. When the lead begins a maneuver (e.g., a turn), the wing must follow the lead to maintain the formation throughout the turn. During the turn the wing must generate many motor commands as it follows the lead. Because the turn takes a few seconds to complete, the outputs increase gradually over the course of the turn, as can be seen in the figure. Thus, the wing periodically encounters a dynamic situation that requires a large amount of reasoning and (motor) responses. Further, the response to this change is not discrete, as in the lead, but occurs continuously over the course of the lead's maneuver.

These differences in the tasks for the two agents account for the relatively large absolute differences in the performance metrics between the lead and wing agents. Because the wings are adjusting their positions relative to the leads, they issue many more output commands than the leads. This, in turn, leads to more individual elaboration cycles and rule firings as the agent asserts the knowledge it needs to issue the best motor command in the current situation. We will now look at the individual performance statistics more closely to compare the behaviors of the Soar 8 lead and wing agents in comparison to their Soar 7 counterparts.

Decisions: In fixed-time mode, we expect the number of decisions to be constant, which we observed for both lead and wings. We actually ran each scenario for 600 seconds, rather than running for a fixed number of decisions. However, because we ran in the fixed-time mode we described previously, there was no variation in the observed decisions, as expected. The differences between decisions in the lead and wing is due simply to an artifact of the data collection. The lead agents run for an extra second after the wings halt to initiate the data collection activities for batch runs.

Elaboration Cycles: In the Soar 8 lead agent, the number of elaboration cycles increased even as the rule firings decreased. As we described for the Blocks World, the increase in elaboration cycles is due to Subtask-limited Reasoning, as we described in Chapter 4. In the Soar 8 wing, the number of elaboration cycles actually decreases. However, this decrease is due to the large drop in the number of rule firings in the wings. Rule firings per elaboration cycle dropped from 1.4 in Soar 7 to .87 in Soar 8.[8] Thus, the overall result is as we expected: fewer rules are firing in parallel over the course of the task.

Production Firings: In both the lead and wing agents, production firings decrease.

[8]One might expect this ratio always to be greater than or equal to 1. However, in Soar, both assertions and retractions occur in individual elaboration cycles. Thus, the lower limit of the ratio is actually 0.5.

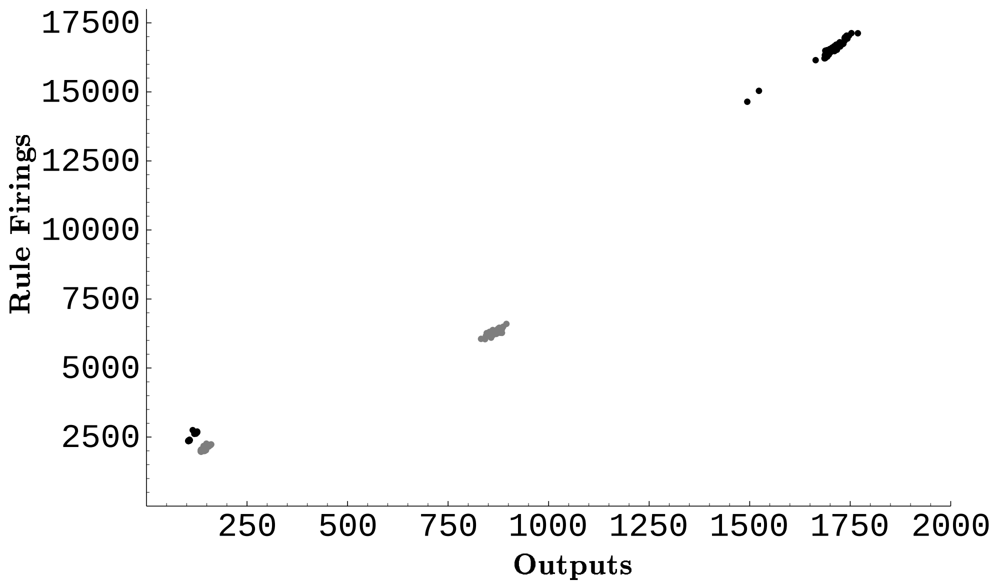


Figure 5.10: Outputs *vs.* production firings for Soar 7 (black) and Soar 8 (gray) agents. The lead agents are the leftmost pair of data, the wings are to the right. As in the Blocks World, as the number of output commands increase, the production firings increase as well.

However, the wing's production firings decrease by 62%, while in the lead, the decrease is only 15%. In order to explain these differences in the relative improvement, we look once again at the differences in the tasks of the lead and wing agents. Figure 5.10 shows that as the number of outputs generated by an agent increases, production firings increase as well. This figure thus confirms the assumption we made previously: rule firings are correlated with output activity (albeit not as strongly as the correlation we observed for agents in the Blocks World). We explain the differences in production firing improvements by further examining the changes in the number of outputs issued by the wing and lead.

The Soar 7 wing sometimes issues the same motor command more than once. The reason for this duplication is that the specific motor command is computed locally, and is thus not available to other subtasks. As we described in Section 5.4.3, we changed the Soar 8 agents so that this computation is now global. The Soar 8 wing never issues a redundant motor command because the command is availability globally, and can be inspected by all subtasks. Thus, the large relative decrease in outputs in the wing can be attributed to this change in the knowledge. Production firings thus also decrease with the decrease in output activity.

In contrast to the wing agent, the average number of outputs issued by the lead during the scenario actually increases. Regeneration is the source of these additional outputs. In a few cases under Soar 8, a subtask for adjusting heading, speed or altitude can get updated repeatedly in a highly dynamic situation (e.g., a hard turn). The Soar 7 agent uses subtask knowledge to decide if the current output command needs to be updated. However, in Soar 8, the subtask may be retracted due to a dependence on a changing value (e.g., some subtasks for turning depend on the current heading). In this case, when the subtask is regenerated following a retraction, the lead may generate a slightly different

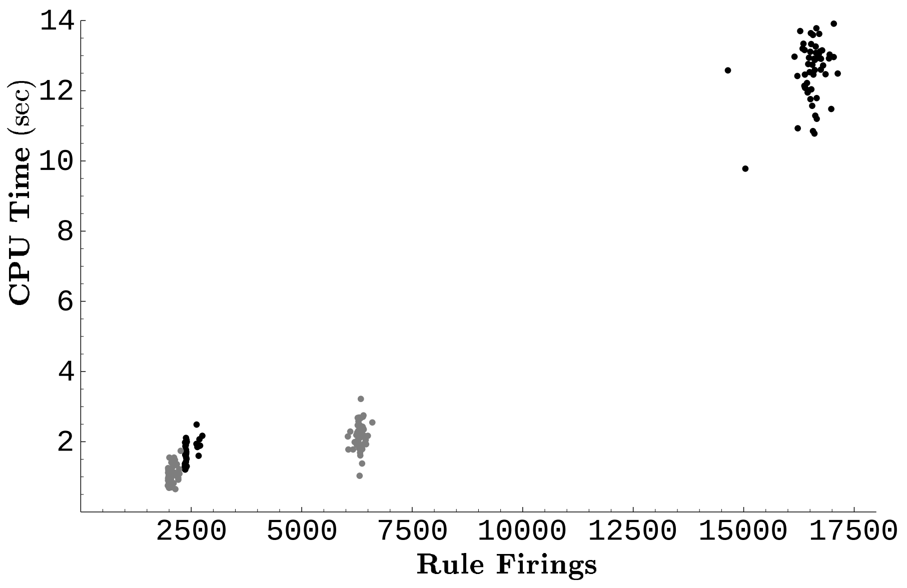


Figure 5.11: Production firings *vs.* CPU time for the Soar 7 (black) and Soar 8 (gray) agents. The lead agents are the leftmost pair of data, the wings are to the right.

motor command. For example, the lead might decide to turn to heading 90.1 instead of 90.2. This decision causes the generation of a new output command that would not have been re-issued in Soar 7 and accounts for the increase in outputs. It also suggests that the knowledge base could be further modified to avoid this regeneration and further decrease production firings.

Together, these results show that the decrease in production firings is not solely due to the global availability of the current motor commands in Soar 8. Instead, the reductions are also due to the use of less consistency knowledge and the interruption of rule firings in terminating subtasks. Although the large magnitude of the improvement in the wing is due to the globalization of the motor commands, overall rule firings decrease due to the architectural solutions to consistency as well. Significantly, the agents perform the same tasks using less knowledge.

CPU Time: In both the lead and wing agents, CPU time decreases in Soar 8. Similar to the results we observed for production firings, the improvement in the lead is about half (40%) the improvement in the wing (81%). However, Figure 5.11 shows that CPU time is poorly correlated with production firings in this domain. The poor correlation is due to the presence of wait states, which were not needed in the Blocks World. In these waits states, generally no productions fire. Thus, the agent's CPU time increases and becomes less correlated with individual production firings.

As we observed in the Blocks World, improved match time also contributes to the overall performance improvement in μTAS. In comparison to their Soar 7 counterparts, the match time of the Soar 8 wing improved by almost an order of magnitude improvement while the match time in the Soar 8 lead agent improved by 5-to-10%.[9] The main

[9]As in the Blocks World, these trends are based on a few observations of the data, rather than a significant analysis. In particular, in μTAS, we did not collect data for the number of tokens generated;

	Avg. In-Range Time	Avg. Launch Time	Reaction Time	n
Soar 7	161.816	162.084	.268	95
Soar 8	162.048	162.993	.945	99

Table 5.4: A comparison of average reaction times for launching a missile in μTAS. The "in-range" columns shows the time, on average, at which an enemy plane came into missile range during the course of the ten-minute scenario. "Launch time" presents averages of the actual times the missile was launched. The "reaction time" is the difference between the in-range time and the launch time. The reaction time is averaged over n runs.

component of these differences is attributable to the decrease in production firings. There are fewer production firings and thus fewer instantiations to generate. Additionally, the decrease in total productions may also contribute to the decrease in match time, as we described for the Blocks World results. Again, these results offer no guarantee that match time will always decrease in Soar 8. Importantly, however, we have observed in two different domains that Soar 8 reduces total knowledge and further constrains the remaining knowledge while the architecture leverages these small reductions or improvements in the knowledge design into better performance in terms of CPU time.

Differences in Responsiveness

CPU time decreases in the Soar 8 μTAS scenario in comparison to the Soar 7 agents. Thus, as we described for the Blocks World, we expect responsiveness to generally improve in Soar 8 because the architecture is processing individual decision cycles more quickly. However, in Section 5.4.3, we described changes to the Soar 8 agent's knowledge that split a complex series of actions in a single subtask into a series of subtasks. This change negatively impacts responsiveness.

Table 5.4 provides an example of the decrease in responsiveness. When a enemy plane comes in range, the agent executes a series of actions, leading to the firing of a missile. We call the time it takes from when the enemy agent comes in range to when the agent actually launches the missile the *reaction time.* Reaction time is thus a measure of the agent's responsiveness. The Soar 7 agent is able to launch the missile in just over a quarter of a second. However, the the Soar 8 agent is about three-and-a-half times slower than the Soar 7 agent in launching the missile, taking almost a full second, on average.

Split subtasks, regeneration, and subtask selection all contribute to the increase in reaction time. When we split subtasks, we recognized that tasks that previously took one decision to initiate might take many more. For example, a task with n steps, all executed serially in a single decision, might now take n decisions. However, the reaction time cannot be explained wholly by this increase. There are only a few actions that are necessary for launching a missile; therefore, we would expect only an increase of, at most, a few hundred milliseconds. However, by dividing subtasks into separate steps, the sequential series of actions can be interrupted. In particular, a number of regenerations occur in the LAUNCH-MISSILE problem space as the agent prepares to fire the missile in a highly dynamic situation. The agent sometimes chooses to undertake a prior action because the

thus, we were unable to compare the number of tokens generated, as we did for the Blocks World. The results we report here are consistent with the expectation that the token activity falls in Soar 8, as it did in Soar 7.

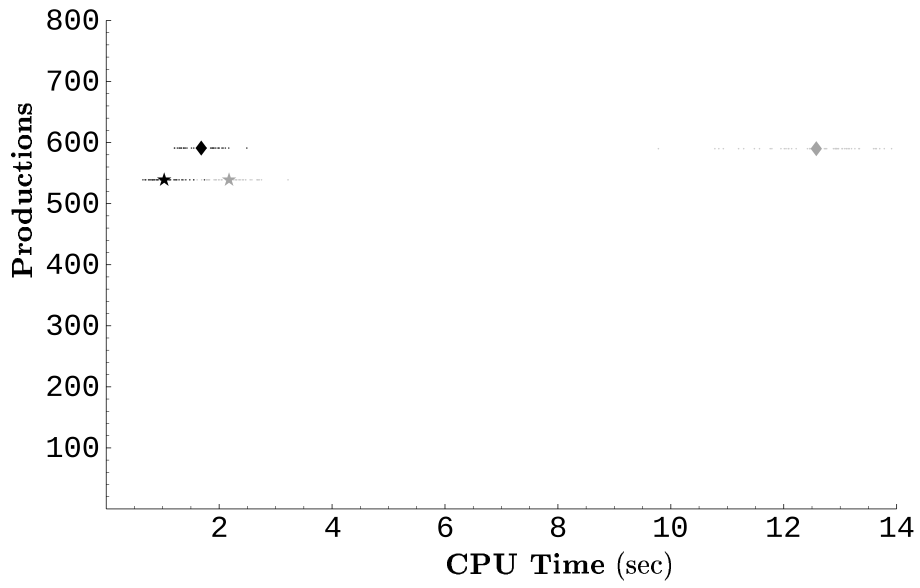


Figure 5.12: μTacAir-Soar Summary: Mean CPU Time in milliseconds *vs.* knowledge in productions for Soar 7 (diamond) and Soar 8 (star) agents. The individual points show the actual distribution of CPU time for each agent. The points for agents flying lead aircraft are shown in black, wing aircraft in gray.

situation has changed enough that a slightly different action might be necessary.[10] The combination of the split subtasks, regeneration, and the selection knowledge for individual subtasks all lead to the degradation in reaction time.

Some additional re-engineering of the knowledge could probably reduce this response time. However, this result will not be an easy one to avoid, in general. Dynamic hierarchical justification requires that subtasks with different dependencies be initiated and terminated separately, or else risk unnecessary regeneration. However, by splitting complex tasks into separate subtasks, individual actions are delayed both because the subtasks are split, and because the selection for a particular subtask in the series can be postponed when better subtask choices are available. In the next chapter, we show that learning can resolve this dilemma by compiling the results of subtask reasoning into new knowledge that can mimic the effect of a complex subtask, but at a higher level of the hierarchy.

5.4.5 μTacAir-Soar Summary

Figure 5.12 illustrates the μTAS results with respect to the knowledge and performance dimensions central to our evaluation. In the figure, we see that Soar 8 lead agents improve slightly in both performance (a smaller average CPU time) and knowledge (fewer total productions). The Soar 8 wing agent shows the identical improvement in the knowledge dimension, but a much greater improvement in overall performance. As we discussed in the preceding sections, this improvement in the performance was due both to changes

[10]This regeneration of the motor commands is one of the sources of the increase in outputs in the lead agent, as we described in Section 5.4.4.

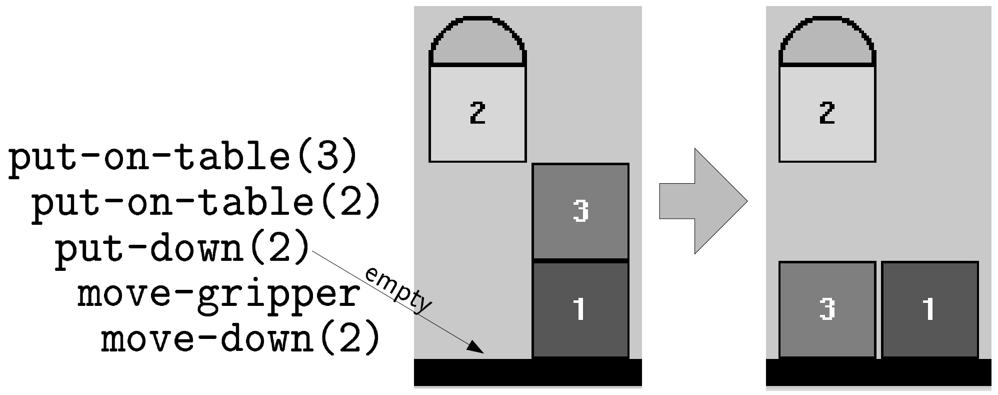


Figure 5.13: GOHHC is more robust when the dynamics of the environment change. Reproduced from Figure 2.6.

in the way some calculations were made, and to the expected performance improvement using Soar 8. These results demonstrate that Soar 8 can be expected to reduce engineering effort and improve performance in complex domains. One cost of these improvements is that responsiveness in specific situations may degrade.

5.5 Summary of Empirical Evaluation

Our results from the Blocks World and μTAS domains were consistent with our expectations: both knowledge engineering cost and overall performance in Soar 8 improved in comparison to independently-developed Soar 7 agents. However, the evaluation did reveal some surprises. In particular, we observed that ensuring consistency sometimes requires additions to domain knowledge and reorganization of that knowledge. The additions were necessary when domain knowledge had been overlooked in the Soar 7 agent that proved necessary in Soar 8. The reorganizations were necessary when multiple, independent tasks were grouped into a single, complex subtask, leading to unnecessary regenerations. Interestingly, these changes to the domain knowledge provide new constraints in agent design. Architectural regeneration identifies violations of the constraints, which, in turn, allows quick repair. Based on these results and our experience using Soar 8, we hypothesize that these design constraints can potentially improve the design of agent knowledge bases, while regeneration, used as a debugging aid, can greatly reduce design cost. That is, in addition to reducing the number of productions necessary for designing an agent for some task, Soar 8 may also reduce the time/dollar cost of writing individual productions.

Significantly, the improvements in behavior coincided with an architectural guarantee of consistency in the agents' processing across hierarchical levels. For example, as a simple experiment, we applied the Soar 7 and Soar 8 blocks world agents to the situation we described in Figure 2.6, reproduced here as Figure 5.13. The Soar 7 agent fails when the block moves because it lacks knowledge to recognize the inconsistency between its previous empty assertion and the new situation. On the other hand, Soar 8 handles this problem gracefully, because the architecture recognizes the relationship between local processing and dependencies higher in the subtask hierarchy. In the specific situation in Figure 2.6, the Soar 8 agent actually retracts the `put-on-table(3)` subtask, because **block-3** is on the table, and thus the selection of that subtask is no longer consistent with the current situation. In the next decision, the agent chooses `stack(2,3)` and decomposes this subtask into actions to put **block-2** on **block-3**. If we introduce a new block (e.g., **block-4**) and place it in the `empty` space below **block-2**, the architecture responds by retracting the subtask goal for `put-down(2)` (i.e., the subtask that contains the `empty` assumption). In the subsequent decision, it begins to recalculate empty spaces in order to continue its attempt to put **block-2** on the table. Thus, although our evaluation did not stress this aspect of Soar 8, Soar 8 agents can be expected to behave more robustly in situations for which they were not specifically designed, because the architecture, rather than agent knowledge, ensures consistency in the asserted knowledge.

These improvements were not achieved without cost. In particular, regeneration led to potential inefficiency in both the Blocks World and μTAS agents. Although overall efficiency improved, some of the improvement was due to improvements in the average match cost of productions, which cannot, in general, be guaranteed in all domains. Further, Goal-Oriented Heuristic Hierarchical Consistency requires that complex subtasks be split into independent subtasks that simplify knowledge design and reduce regeneration. However, this simplification also reduces responsiveness. In the following chapter, we explore ways in which to use learning to offset these potential performance costs as an agent gains experience in its domain.

Chapter 6

Improving Performance through Compilation

We see those of experience succeeding more than those who have theory without experience. The reason for this is that experience is knowledge of the particulars ... and actions, and the effects produced, are all concerned with the particular. – Aristotle

In Chapters 3 and 4, we presented solutions to inconsistency arising from persistence and multiple threads of reasoning. In the previous chapter we showed these solutions generally led to slight improvements in performance, in addition to guaranteeing consistency in hierarchical reasoning. In this chapter, we consider using *knowledge compilation* to further improve performance, thus addressing the second limitation of hierarchical decomposition in execution environments introduced in Chapter 2. Compilation caches the results of hierarchical decomposition in specific situations. If the agent encounters a similar situation, the compiled knowledge will apply as soon as the knowledge can be retrieved, thus obviating the delay that occurs due to decomposition. Compilation thus provides experience-directed composition of complex behavior from simple subtasks.

In the following, we introduce a framework for using compilation in execution domains. Importantly, compilation should occur as the agent executes its task, should not require significant additional knowledge engineering demands, and should not adversely impact agent performance. We show how any failure of the architecture to ensure consistency leads to specific problems, requiring solutions that violate one or more of these criteria. However, by guaranteeing consistency, Goal-Oriented Heuristic Hierarchical Consistency avoids these problems and allows real-time compilation over dynamic behavior. We apply compilation to the Soar 8 agents we used for our empirical evaluation of Goal-Oriented Heuristic Hierarchical Consistency in Chapter 5. Our results show that performance and responsiveness improve after compilation, but overall performance improvement is not significant if hierarchical knowledge is used in addition to the compiled knowledge.

6.1 Compiling Hierarchically Decomposed Knowledge

In Chapter 2 we hypothesized that compilation could be used to improve the performance of agents using hierarchical decomposition. In the following, we present an example from the Blocks World to show how agent experience, coupled with a compilation mechanism that caches the agent's task experience, can lead to improvements in performance when executing future tasks. We also detail the requirements for compilation in execution environments.

6.2 Knowledge Compilation in the Blocks World

Refer again to the Blocks World example we introduced in Figure 2.2. Assume an architecture executes the task according to rules like those introduced in Table 2.1, stepping the gripper to the right to begin building a tower. We want the compilation process to generate new knowledge that avoids intermediate goals in the decomposition, such as `pick-up`. The end result of compilation should resemble a reactive rule, one that maps the input state to an action. In this example, the resulting rule might look like this:

```
4.   IF      Task-Goal(Tower(x,y,z))
             Not(On-Table(x))
             Clear(x)
             Left-of(Gripper, x)
             Higher(Gripper, x)
     THEN    Execute(Step(right, Gripper))
```

This rule is dependent upon percepts and task goals and generates a primitive. Thus, if the agent encounters a situation in which Rules 1, 2, and 3 would have fired previously, this new rule can now subsume that function, and begin the execution of `step-right` immediately.

How can the architecture automatically generate Rule 4? The agent begins executing the task from the initial configuration in Figure 2.2. Rule 1 fires, generating the goal to put **block-3** on the table. Rule 2 then fires and establishes a goal to pick up **block-3**. Now the conditions are true for Rule 3 to fire. Rule 3 generates the primitive command, `step-right`. Because the primitive action terminates further decomposition, the generation of a primitive can act as a trigger for the compilation mechanism. In other words, when a primitive command is generated, the architecture recognizes the situation as an opportunity to compile the current hierarchy.

Goal regression provides a potential compilation process (Mitchell et al., 1986). The regression traces back through the individual rule firings until the goal (the initiation of the primitive) is expressed in terms of only task goals and percepts. For example, in Rule 3, `Left-Of` and `Higher` are percepts and thus added directly to the rule being constructed. However, the the `pick-up` goal is not a task goal or input, and so the backtrace continues to Rule 2, the rule that created `pick-up`. Rule 2's conditions are now added to the compiled rule. However, the `put-on-table` goal is not a percept and this triggers yet another regression step to Rule 1. Rule 1's conditions are solely dependent upon percepts and the task goals, terminating the compilation. The resulting rule is:

```
5.   IF      Task-Goal(Tower(3,2,1))
             Not(On-Table(3))
             Clear(3)
             Left-of(Gripper, 3)
             Higher(Gripper, 3)
     THEN    Execute(Step(right, Gripper))
```

This rule is identical to the Rule 4 except that it is specific to the particular task goal of building `Tower(3,2,1)`. Thus, simple regression provides a mechanism sufficient for compilation, at least in rule-based systems.

Explanation-based learning (EBL) approaches have used this goal regression technique to operationalize problem solving knowledge in a number of domain classes including concept formation (DeJong and Mooney, 1986; Mitchell et al., 1986), planning (Fikes et al., 1972; Minton, 1988) and scheduling (Carbonell et al., 1991). EBL uses a *domain theory* to generate an *explanation* of why some *training instance* is an example of a *goal concept* according to some *operationality criterion* (DeJong and Mooney, 1986). In execution domains, the goal concepts for EBL are the situations in which each primitive operation should be generated in the current external state, *where the current external state is defined by the available percepts and task goals.* This definition of external state serves as our operationality criterion for execution domains: a condition is operational if it is a direct input from perception or a task goal and does not require decomposition.[1] Explanation occurs over the domain theory, or the hierarchical knowledge about the task domain. The training example is the specific current situation. Therefore, the object of EBL in execution environments is to operationalize appropriately the generation of a primitive output command. EBL algorithms such as EBG (Mitchell et al., 1986) and chunking (Laird et al., 1986a) also use generalization schemes that result in compiled rules more general than Rule 5.

6.2.1 Requirements for Knowledge Compilation in Execution Domains

The compilation process we have described above is sufficient for the general goal of caching the results of hierarchical decomposition in particular situations. However, our larger goal is to incorporate compilation within agents behaving in dynamic domains. As we have seen previously, the impact of a new capability on knowledge design cost and performance must be considered as well. In the following, we outline additional requirements for compilation in execution domains, based on the particular constraints of agent execution systems.

Does Not Introduce New Knowledge Design Costs: Ideally, the compilation mechanism caches the results of a specific decomposition as a new rule in the domain irrespective of the representation. If the learning algorithm is complete and correct (as required below), then the agent will not need to represent its knowledge in any special way for the learning to occur, nor need special knowledge to enable learning itself. Thus, the introduction of compilation should not introduce new costs in the design of an agent.

Does Not Degrade Performance or Responsiveness: The performance system must remain responsive to its environment, even with learning. Compilation requires near-continuous learning (whenever a primitive is generated) and thus must be efficiently implemented. The explanation generation component of EBL occurs during the generation of the primitive, guided by the execution system. Thus, this component of EBL impacts actual performance minimally. This leaves the sec-

[1]In Section 6.4.2, we will relax this operationality criterion for Soar to include any data in the highest level of the hierarchy.

ond part of the algorithm, generalization, to consider. While some EBL system's "re-prove" the explanation to create a generalized explanation, several systems use architecture-specific techniques to improve efficiency. Examples include Prodigy's EBS algorithm (Minton, 1988) and Soar's chunking mechanism (Laird et al., 1986a).

Agent knowledge bases may grow significantly as compiled experience increases the number of rules. The addition of new knowledge, especially by an automatic process such as compilation, can increase the cost of knowledge search for individual pieces of knowledge such that knowledge search is more expensive than problem search (i.e., the decomposition) in a smaller knowledge base. This *average growth effect* (Doorenbos, 1993) is a specialized case of the well-known *utility problem* (Minton, 1988). We require that the agent's reasoning not slow down significantly as the number of rules in the agent increases. For instance, for a rule-based system, the match time for a rule set should be no more than linear in the number of rules and preferably sub-linear or independent of the number of rules. The RETE algorithm, which we have discussed in previous chapters, provides one example of current rule matching that technology can support such a requirement (Doorenbos, 1994).

Preserves Correctness: Execution during and after compilation on a given task must be as proficient as the same system without learning. This goal requires a computationally inexpensive compilation algorithm, as discussed above, and a compilation process that preserves correctness. The compiled knowledge must preserve the original behavior in the learned rules such that an agent, in the same or similar situation, will repeat its original behavior. Although this requirement is important for the compilation algorithm, it also makes demands on the execution system. For instance, the target language of compilation must be sufficient for describing the necessary computations in the subtask (Laird et al., 1986b).

6.3 Problems in Knowledge Compilation

Knowledge compilation has been used successfully in static domains and in dynamic domains in which the learning occurs "off line" from the execution. For example, STRIPS (Fikes et al., 1972) compiled macro-operators over a static planning space even though these operators were then used to direct a robot in the external world. However, EBL has been applied to external domains in only limited cases (e.g., (Bresina et al., 1993; Laird and Rosenbloom, 1990; Mitchell, 1990)). In an interactive domain, where the training instance may change over time, knowledge compilation methods will be potentially difficult to incorporate. In particular, previous systems have been dependent upon specific representation schemes to avoid problems resulting from compilation when the data base of assertions can be inconsistent. These problems include creating rules that include features that never co-occur (the non-contemporaneous constraints problem) and conflicts between compiled and original task knowledge (the knowledge contention problem). These problems result when an architecture allows persistence and multiple threads of reasoning but does not ensure consistency in the hierarchical processing. We now introduce these problems specifically, and also review two additional problems, over-specific and over-general compilation, that are not specific to execution domains.

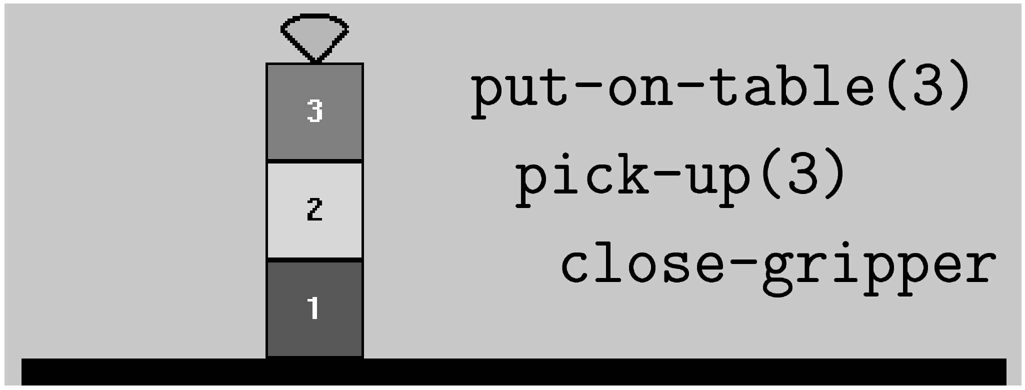


Figure 6.1: A potential source of non-contemporaneous constraints. The original selection of `put-on-table(3)` is dependent on *clear*(**block-3**). However, `close-gripper` requires that the **gripper** be immediately above **block-3**, meaning it is no longer *clear*. Regressing through these conditions for compilation leads to the conditions *clear*(**block-3**) and *immediately-above*(**gripper**,**block-3**), which do not occur simultaneously in this domain.

6.3.1 The Problem of Non-Contemporaneous Constraints

Non-contemporaneous constraints (Wray et al., 1996) occur in the conditions of a learned rule when two or more conditions match against features which never occur simultaneously in the domain. A rule with non-contemporaneous constraints can never apply. The agent expends effort to learn something useless and misses an opportunity to have learned some useful knowledge. Non-contemporaneous constraints occur in compilation when the architecture creates persistent assumptions that can become inconsistent with the hierarchical context. Thus, the compilation of non-contemporaneous constraints is another example of a problem that may arise when the agent fails to react to inconsistency-causing changes in hierarchical context, as described in Chapter 3.

To see how non-contemporaneous constraints arise, assume we are using a knowledge-based assumption consistency solution such as Soar 7 and refer again to the task posed to the execution system in Figure 2.2. Suppose now that the gripper has executed the first two step right commands and is now ready to pick up the block by closing the gripper over **block-3**. This situation is illustrated in Figure 6.1. Rule 6 in Table 6.1 fires to close the gripper because the `pick-up` goal created by the firing of Rule 2 has not yet been achieved (its termination conditions are given in Rule 8) and because the gripper is now immediately above **block-3**. Because `close-gripper` is a primitive, executable command, the compilation process is initiated when Rule 6 fires.

The compiled rule will be identical to Rule 5 except that *Immediately-Above* from Rule 6 replaces the *Left-Of* and *Higher* relations that came from Rule 3. The similarity results because both `pick-up` and `put-on-table` are still instantiated goals (their termination conditions as given in Table 6.1 have not yet been met) and thus the regression is repeated through Rule 2 and then Rule 1, as we described previously. The new rule is:

```
9.   IF       Task-Goal(Tower(3,2,1))
              Not(On-Table(Bottom-Block(Tower(3,2,1))))
              Clear(Bottom-Block(Tower(3,2,1)))
```

```
6.   IF      Goal(Pick-Up(x))
             Immediately-Above(gripper,x)
     THEN    Execute(Close-Gripper)

7.   IF      Goal(Put-On-Table(x))
             On-Table(x)
     THEN    DeleteGoal(Put-On-Table(x))

8.   IF      Goal(Pick-Up(x))
             Holding(Gripper, x)
     THEN    DeleteGoal(Pick-Up(x))
```

Table 6.1: Rules needed for the execution of the tower-building task in progress in Figure 6.1.

```
             Immediately-Above(Gripper, Bottom-Block(Tower(3,2,1)))
     THEN    Execute(Close-Gripper)
```

All of these conditions are satisfied at some point during the course of execution. For this rule to fire, the gripper must be immediately above **block-3** and the block must also be clear. However, for a block to be clear, nothing can be immediately above it. Thus, in this domain, these conditions will never be true at the same time; they are non-contemporaneous constraints in the compiled rule. As we have seen in previous chapters, the context of execution changes as it progresses. The regression process inspects every relevant item used in the creation of the current subtask hierarchy and includes all the operational tests it finds in the compiled rule. Thus, non-contemporaneous constraints result when the architecture fails to react to changes in its hierarchical context, and then attempts to compile over the resulting inconsistent hierarchy.

Consider the problem from an explanation-based learning point-of-view. The non-contemporaneous constraints problem arises because the training instance over which compilation occurs changes with time, or more precisely, when the training instance includes multiple, mutually exclusive states. The *clear* relation is fully operational but, at the time at which Rule 6 fires, is no longer true in the environment. When EBL is applied to compile control knowledge within planning systems, even for dynamic domains, non-contemporaneous constraints do not arise because the training instance is static during the generation of the plan. 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. The non-contemporaneous constraints problem can occur in such static environments, if the problem solver is allowed to create persistent features based on other context features that may change. However, the problem is exacerbated in external environments, where change can be both exogenous and endogenous, and occur with higher frequency.

A rule with non-contemporaneous constraints will not lead to inappropriate behavior but rather will never apply. This problem is a serious one because it presents further difficulty in realizing the goal of using a straightforward EBL approach to operationalize execution knowledge in external domains. Instead, rules that can never fire due to non-contemporaneous constraints in the conditions are added to the knowledge base, consuming resources (e.g., they are included in the rule match). Additionally, the same useless rule may be created repeatedly, wasting still more architectural resources. Perhaps most importantly, there is an opportunity to learn something in these situations but the straightforward approach learns something useless instead. Thus, although the learned rules do not lead to errors themselves, some alternative is necessary to avoid undue waste of resources, as well as to operationalize the execution knowledge properly.

Because the compilation of non-contemporaneous constraints arises from inconsistency due to persistence, we expect that any of the approaches we described in Chapter 3 will avoid the problem (including, of course, Dynamic Hierarchical Justification). For example, neither Theo (Mitchell et al., 1991) nor ERE (Bresina et al., 1993) suffer from non-contemporaneous constraints in learned rules. Theo never creates persistent assumptions. Thus, the hierarchy never contains assertions derived in a context nonmonotonic to the current one and compilation is not problematic. ERE, on the other hand, uses domain constraints as a knowledge-based solution to maintaining consistency and thus avoids non-contemporaneous constraints when learning.

Significantly, as we saw in the last chapter, knowledge-based methods do not guarantee consistency. However, without this guarantee, non-contemporaneous constraints can result in compilation even when behavior is not impacted by inconsistencies. For example, if we attempt to use compilation with the Soar 7 agents from the previous chapter, the agents learn many rules with non-contemporaneous constraints, due primarily to the persistence of subtasks. Although it is possible to structure knowledge to attempt to avoid non-contemporaneous constraints, only an architectural solution provides a low-cost guarantee of consistency and thus also guarantees no rules with non-contemporaneous constraints will be learned.

6.3.2 Contention Between Compiled and Hierarchical Knowledge

We now consider another problem, which derives from multiple threads of reasoning in the architecture. *Knowledge contention* arises when knowledge specifying the same action is simultaneously asserted in two different levels of the hierarchy. Before the addition of compilation, we could safely assume that it was unlikely for the same assertion to be duplicated in different levels of the hierarchy because the knowledge engineer would avoid such duplication. However, compilation, by definition, leads to knowledge operationalized at higher levels in the hierarchy and thus potentially duplications in actions. We now introduce an example to show how knowledge contention arises and the problem it presents to the agent.

Suppose again the execution system encounters the situation presented in Figure 2.2. We assume that the agent has compiled a rule for executing a rule to `step-right` in this situation, such as Rule 4. However, now suppose that in addition to the conditions we have seen previously, each rule issuing executing a primitive operation checks that the gripper is *ready* before beginning execution. This assumption results in a compiled rule such as Rule 10:

```
10.  IF      Task-Goal(Tower(x,y,z))
             Not(On-Table(x))
             Clear(x)
             Left-of(gripper, x)
             Higher(gripper, x)
             Ready(gripper)
     THEN    Execute(step-right(gripper))
```

If the gripper is *ready* in the Figure 2.2 state, Rule 10 would fire and execute the `step-right` action. This result is exactly what we desired from the compiled knowledge, moving the gripper without referencing the intermediate goals (and thus not delaying while creating them).

However, consider what happens if it takes time following an action for the gripper to become ready and that the gripper is not ready in the situation we have just described. What does the agent do? It begins to decompose the problem as we described the process in Chapter 2. Rule 1 fires to create the `put-on-table(3)` goal, then Rule 1 fires to create the `pick-up` goal. It this point, the agent can progress no further until the gripper is ready. We assume that the agent has no direct means of making the gripper ready; therefore, the agent just waits to receive the *ready*(**gripper**) signal from the external environment. What happens when the *ready* signal is received? In a parallel architecture, both Rule 10 and Rule 3 fire simultaneously. Depending on the specific implementation of the agent's motor system, the agent may step once to the right, two times to the right, or not at all.

This example illustrates the potential of contention between an agent's original task knowledge and its compiled knowledge. The agent now has two different rules that specify the same action in the same state. However, the architecture cannot generally know the result of a rule firing until after that rule has fired. For instance, two rules could fire in parallel specifying the same actions (or different actions) for the same external situation. Knowledge contention arises because the architecture is pursuing multiple threads of reasoning simultaneously, in multiple levels of hierarchy, similar to the basic problem we described in Chapter 4. Thus, knowledge contention is an example of inconsistency arising from overly aggressive response to changes in the context.

6.3.3 Learning Over-Specific and Over-General Rules

Compilation may sometimes lead to both over-specific and over-general rules. An over-specific rule is one that is unnecessarily restrictive. Rule 5 is an example of an over-specific rule because the name of the actual blocks (e.g., **block-3**) are really not important. Instead, only the current relationships of the tower's bottom block to the table and gripper are necessary for an appropriately general rule.

An over-general rule is one that will fire in a circumstance for which it is inappropriate. For instance, imagine Rule 1 without the condition that the bottom block not already be on the table. This rule would then create the goal to put a bottom block on the table even when the block was already on the table. Learning over-general rules violates one of the general requirements for knowledge compilation, that learning preserve correctness. Thus, avoiding over-general rules is necessary in any approach.

Both over-general and over-specific rules are closely tied to the representation of the task as chosen by the user and the generalization process used in conjunction with regression. Thus, neither of these problems are specific to a system that compiles over hierarchical execution knowledge. Both problems have been described for static domains (Laird et al., 1986b; Rosenbloom and Laird, 1986).

However, both of these problems are complicated by interaction with external environments. For instance, if the representation of the environment via input contains absolute information (e.g., coordinates in Cartesian space) then using this input directly may lead to over-specific rules, depending on the specific generalization technique. Therefore, care must be taken in not only choosing the task representation but also how input knowledge is represented. For both these problems, careful choice of representation and task knowledge will lead to the correct level of generalization.

In the Blocks World and μTAS domains, in which we have used compilation (as described below), we had only to modify slightly the input representation to avoid over-specific compilation in the Blocks World and μTAS. These changes account for 23 of the 24 productions changed for learning in Table 5.3. No changes were necessary for over-general learning in the Blocks World, and only a single production in μTAS needed to be changed.[2] Because these problems are highly dependent on the specific compilation algorithm and proved to be insignificant in our empirical investigations, we will not consider them further.

6.4 Compilation under Goal-Oriented Heuristic Hierarchical Consistency

In the previous section, we introduced the non-contemporaneous constraints and knowledge contention problems that result from inconsistencies in agent processing. As we described in Chapters 3 and 4, Goal-Oriented Heuristic Hierarchical Consistency guarantees consistency in the agent's processing; thus, we expect that GOHHC will provide a solution to both of these problems. In the following, we describe how Goal-Oriented Heuristic Hierarchical Consistency solves the two compilation problems. Having solved these problems, in the following section we then use compilation with the agents we used in the previous chapter and observe if compilation under GOHHC meets our requirements for knowledge compilation and leads to improvement in performance.

6.4.1 Solutions for Knowledge Compilation

Goal-Oriented Heuristic Hierarchical Consistency prevents the compilation of non-contemporaneous constraints in learned rules. Dynamic Hierarchical Justification never allows a memory feature to persist any longer than features in the hierarchical context that led to its creation. For example, in Figure 6.1, when the gripper moves above **block-3**, the architecture removes the `put-on-table` subtask, because **block-3** is no longer clear, and thus the subtask initiation conditions are no longer supported in the current environment. At this point, a new `put-on-table` operator is instantiated in the current situation, which includes the relation *immediately-above*(**gripper**,**block-3**) rather than the *clear*(**block-**

[2]This change was motivated by locally-negated condition, the source of most over-general learning in Soar agents. Because the architecture cannot determine, in general, the situations in higher levels of the hierarchy that lead to something *not* being true in the local subgoal, these conditions are ignored when compiling, and can thus lead to over-general compiled knowledge.

3) relation from the initial `put-on-table` operator. When the close gripper command is generated, the compilation process will use the current instantiation of `put-on-table`, resulting in Rule 11 rather than Rule 9:

```
11.  IF      Task-Goal(Tower(3,2,1))
             Not(On-Table(Bottom-Block(Tower(3,2,1))))
             Immediately-Above(Gripper, Bottom-Block(Tower(3,2,1)))
     THEN    Execute(Close-Gripper)
```

Thus, by ensuring consistency among assertion in the hierarchy, GOHHC also avoids the non-contemporaneous constraints problem when compiling over the hierarchy.

GOHHC also prevents simultaneous knowledge contention. Recall from Chapter 4 that Subtask-limited Reasoning limits reasoning to only a single level of the hierarchy at a time. In the example we presented above, both the compiled Rule 10 and Rule 3 match simultaneously. However, under GOHHC, Rule 10 would always be asserted before Rule 3 because Rule 10 matches at a higher level of the hierarchy. Subtask-limited Reasoning thus provides conflict resolution between compiled and original task knowledge, always preferring the compiled knowledge because it necessarily matches higher in the hierarchy.

Subtask-limited Reasoning does not provide a complete solution to the knowledge contention problem. It prevents the rules from firing simultaneously, but would not prevent Rule 3 from firing sometime after Rule 10, provided Rule 3 still matched. We assume that the agent's knowledge includes knowledge that prevents the duplication of assertions. For practical purposes, this requirement is already met by the agent's knowledge. For instance, Rule 3 in the Soar 7 agent's knowledge base actually includes the condition that the motor command has not already been initiated. Subtask-limited Reasoning provides a solution for knowledge contention that an agent cannot recognize, due to the parallel execution of identical actions. When the actions are not initiated simultaneously, the agent can simply rely on its task knowledge to avoid knowledge contention.

6.4.2 Empirical Results

We expect Goal-Oriented Heuristic Hierarchical Consistency to avoid the compilation problems without requiring extensive additions to an agent's knowledge base or special representation conventions. We have also suggested that Soar's chunking algorithm provides the run-time performance and correctness properties that we require for knowledge compilation and that the architecture's RETE algorithm should not cause substantial slowdown as the knowledge base grows with experience. Provided all these assumptions are valid in Soar, we expect that Soar 8 agents should improve their performance by using compilation.

In this section, we explore the use of compilation under Goal-Oriented Heuristic Hierarchical Consistency empirically, by assessing the impact compilation has on the performance and knowledge of Soar 8 agents in the Blocks World and μTAS domains. We begin by outlining in greater detail how Soar's learning mechanism meets the performance requirements of agents behaving in dynamic domains. We then present results from the Blocks World and μTAS domains that show performance improves in the Blocks World, while response time improves in μTAS.

Compilation using Soar

In this section, we look more closely at Soar's learning mechanism, chunking, in order to better understand how chunking meets the requirements for knowledge compilation we described previously. The details of chunking are described elsewhere (Laird et al., 1986a; Newell, 1990; Tambe et al., 1990); we instead concentrate on a general description of the chunking process and a brief analysis of the algorithm to show how it interacts with the requirements.

Chunking compiles threads of reasoning whenever a new assertion is returned to a state higher in the hierarchy than the one in which it was created. Chunking produces a new production, or chunk, that can generate this *result* in the future, without the subgoal. As problem solving progresses, the dependencies between assertions are maintained in a data structure. When a result is generated, the chunking process regresses through the stored dependencies until the result can be summarized from assertions in memory at the level of the result (and above). The backtrace process is similar to the algorithm we introduced in Figure 3.3 for building assumption justifications. Because multiple assertions can be created from a single rule firing, the backtrace process is linear in the number production instantiations. Soar uses a heuristic generalization process which can lead to over-specific and over-general knowledge, as we discussed previously. However, this generalization technique does avoid re-proving the problem solving, a more conservative but more computationally costly generalization technique used by many EBL algorithms. Because individual chunks can be constructed in linear time, the chunking process can be used as behavior progresses without significantly impacting performance. Thus, chunking should meet this requirement for knowledge compilation.

We prefer to use chunking without making modifications specific for execution, and without using a special knowledge representation for compilation. In execution agents, decomposition leads to the eventual execution of a primitive action. In Soar, the interface to the external environment is part of the top state or *base level space* (Newell, 1990). Therefore, the initiation of an external action requires returning a motor command to the top state as a result, and chunking compiles the decomposition that led to the result. Thus, we can use chunking as it exists, using existing agent knowledge.

There is one exception to the unproblematic use of chunking for compilation in execution environments. As we described in Chapter 5, Soar selects an operator and then creates a subtask (as a subgoal) to implement the operator. Because the highest operators appear in the top state, the chunking process does not automatically remove these operators from compiled rules. For example, instead of Rule 4, the production that Soar's chunking mechanism actually creates is better summarized by Rule 12:

```
12.  IF      Clear(Bottom-Block(Tower(x,y,z)))
             Left-of(Gripper, Bottom-Block(Tower(x,y,z)))
             Higher(Gripper, Bottom-Block(Tower(x,y,z)))
             Operator(put-on-table)
     THEN    Execute(Step(right, Gripper))
```

The conditions of this rule are the same as those we described previously, except that the rule includes the highest operator in the current hierarchy, `put-on-table`, and does not include the conditions in Rule 1 because regression terminates at the top-level `put-on-table` operator. Thus, the actual rules learned by Soar are not completely reactive to the environment, as we desired. Although it is possible to structure the knowledge to avoid including the highest operator in the compiled rules, rules with these conditions have a number of desirable properties. First, including the operator allows each compiled rule to reflect a deliberately chosen goal, such as `put-on-table`. After compilation, the `put-on-table` operator, once chosen, will be able to issue all the primitive actions necessary for putting a block on the table. Rather than learning reactive rules for behavior, compilation in Soar via chunking allows the agent to initiate complex behaviors through the deliberate selection of a single goal. Thus, chunking composes complex operators from simple operators lower in the decomposition. Second, only a single operator can be chosen at any particular time for any particular level in the hierarchy. The presence of a known single-valued condition helps reduce the total match cost of the learned rules by providing additional match constraint in compiled rules.

Finally, we suggested earlier that Soar would not suffer from the utility problem as it compiled. What causes us to believe that the architecture will not slow down with additional productions? Recall from our discussion in Section 5.3.2 of the previous chapter that match time in Soar is bounded by the number of partial production instantiations, or tokens. The total size of the token space is bounded by the number of *unique* conditions in the agent's productions, rather than the number of individual production conditions. For example, Rule 2 and Rule 12 share two identical conditions, the *left-of* and *higher* relations. In general, compilation adds new rules to the agent's knowledge base but few new unique conditions. The conditions in the compiled rules are derived from conditions in the rules that led to the creation of the result, directly or indirectly. For example, the conditions from Rule 2 are included in the compiled rule because the `pick-up` goal contributed to the execution of the `step-right` primitive. Of course, there is a non-zero "bookkeeping" cost for mapping conditions to productions when creating new production instantiations. However, (Doorenbos, 1993) shows that for RETE (and thus Soar), the number of compiled productions must grow several orders of magnitude to see appreciable utility problems from this effect. Thus, the RETE algorithm and the duplication of conditions from domain knowledge in compiled knowledge allow Soar to minimize utility problems when adding new productions.

Based on these details of Soar's chunking algorithm, we expect chunking will be sufficient for compilation in external domains. Assuming GOHHC solves the compilation problems, chunking allows us to 1) use existing domain knowledge, with few changes necessary specifically for learning; 2) learn while also behaving with minimal impact on performance; and 3) avoid utility problems. Some changes will be necessary to address over-specific and over-general learning, however. In the next section, we empirically explore these claims.

Compilation in the Blocks World

In the Blocks World, we simply repeat the experiment we reported in the previous chapter, but the Soar 8 agent learns while executing the individual tasks we described in Figure 5.2. As the agent gains experience in the domain, we expect the knowledge it learns to transfer to new tasks (i.e., different initial configurations of blocks). This transfer will obviate the need for decomposition in the later execution tasks and thus we expect the performance of the learning agent to improve over the execution of all the tasks in the Blocks World, relative to the non-learning agents.

Table 6.2 summarizes the results from the Blocks World for our agents. We include,

	Soar 7		Soar 8			
	No Learning		No Learning		Learning	
	$\bar{x}$	s.d.	$\bar{x}$	s.d.	$\bar{x}$	s.d.
Rules	188		175			
Learned Rules	0		0		164	
Decision Avg.	87.1	20.9	141.1	38.7	29.1	10.4
Avg. Outputs	22.3	6.1	22.3	6.1	22.3	6.1
Avg. CPU Time (ms)	413.1	121.6	391.6	114.0	288.3	71.1
Avg. Elaboration Cycles	274.3	65.8	562.9	155.7	200.5	57.6
Avg. Rule Firings	720.3	153.5	855.6	199.6	395.2	77.9

Table 6.2: Summary of knowledge and performance data from the Blocks World with compilation.

for comparison, the non-learning results presented in Table 5.1. As we anticipated, the Soar 8 agent experienced no problems in compilation, while the Soar 7 agent, if learning is used, was not able to complete even one task due to the compilation problems we described previously. Additionally, CPU time, decisions, elaborations and productions all decrease as the Soar 8 agent learns while executing the Blocks World tasks, leading to improvement in the learning agents' performance in comparison to the non-learning agents. We consider specific results in more detail below.

We begin the analysis by examining the content of compiled rules and the learning rate. Compiled knowledge in the Blocks World recognizes sub-configurations of blocks. For example, after compiling Rule 12, the agent can immediately recognize that it should `step-right` in any situation in which **block-3** is not on the table and is *clear*, and the gripper is to the left of the block, and not holding anything. After having compiled this rule, the agent is able to avoid the decomposition necessary to execute `step-right` for this situation, regardless of the specific locations of the other two blocks, or the distance between the **block-3** and the gripper. Thus, as we expected, the agents improve their performance by recognizing situations they have previously encountered and executing a primitive without decomposition.

The large magnitude of the improvement in performance occurs because the agents are able to transfer their experience to most new tasks. For example, Rule 4 is applicable in many more tasks than the one in which it was compiled. Figure 6.2 shows that the transfer in the Blocks World is significant. Initially, the agents compile a large number of new rules. For example, in the first fifty tasks (i.e., the first 5% of the tasks), the Soar 8 agent compiles 62% (103 of 164) of the total number of rules it will compile over all the tasks in the Blocks World. After a relatively short time, the agent is able to avoid decomposition throughout execution and no new rules are learned, corresponding to the flat part of the Soar 8 curve. When a novel configuration is encountered (e.g., Run 173), the Soar 8 agent reverts to decomposition to execute the task and compiles the results of the decomposition in the new situation.

Compilation reduces the need for decomposition to find a primitive action to execute, but does not reduce the number of outputs or number of blocks that must be moved for any initial configuration of blocks. As we showed in Chapter 5, production firings are closely correlated to these two measures of complexity in the Blocks World. Figure 6.3 shows the relationships of production firings to outputs and number of blocks to move in the Soar 8 non-learning and learning agents. The learning curve uses all but the first

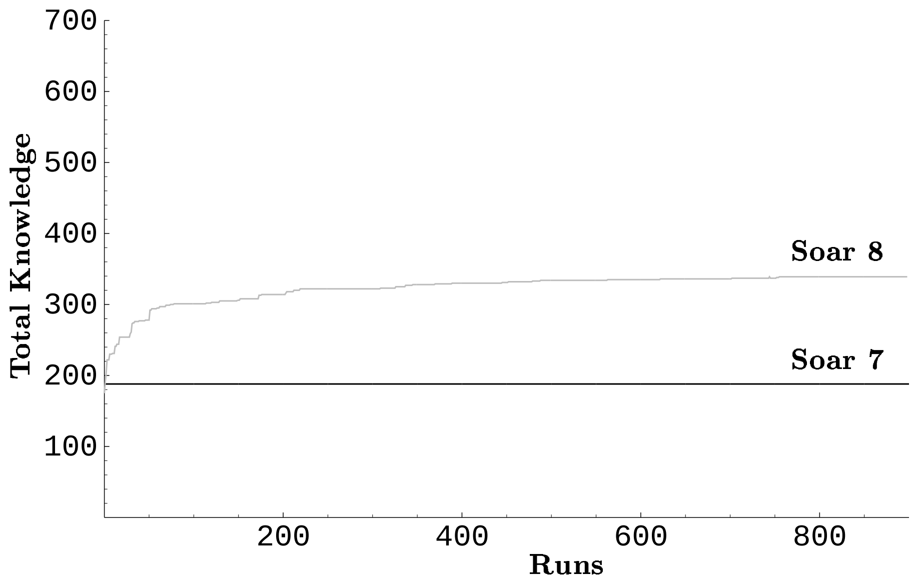


Figure 6.2: Knowledge as a function of runs in the Blocks World.

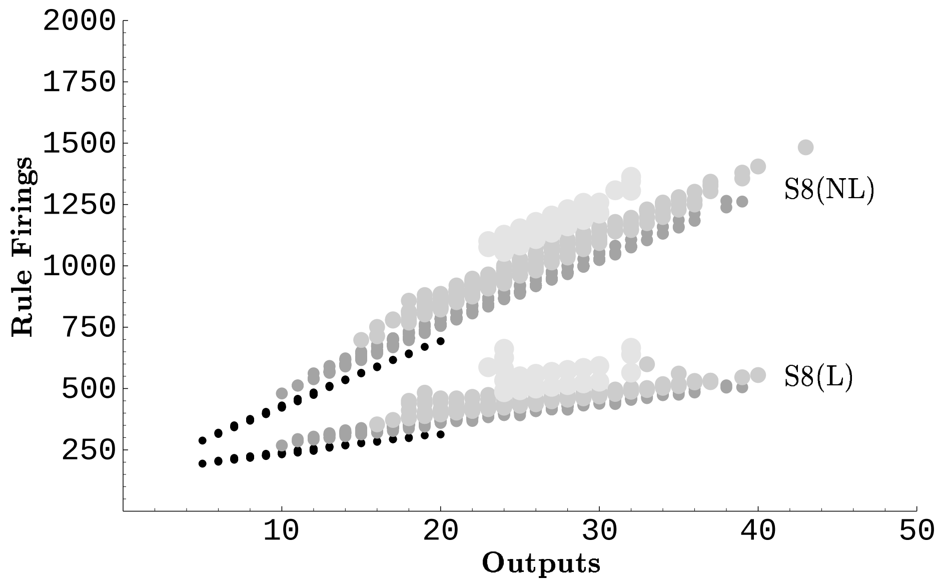


Figure 6.3: Outputs *vs.* production firings in the Soar 8 agent for both learning (L) and non-learning (NL) runs. The plot for the Soar 8 learning agent does not include data from the first 100 cases (i.e., during the runs in which most compilation occurs). The shade and size of each datum signifies the the number of blocks moved for the run: (smallest, black) $\rightarrow$ 1 block; (larger, dark gray) $\rightarrow$ 2 blocks; (larger, medium gray) $\rightarrow$ 3 blocks; (largest, light gray) $\rightarrow$ 4 blocks.

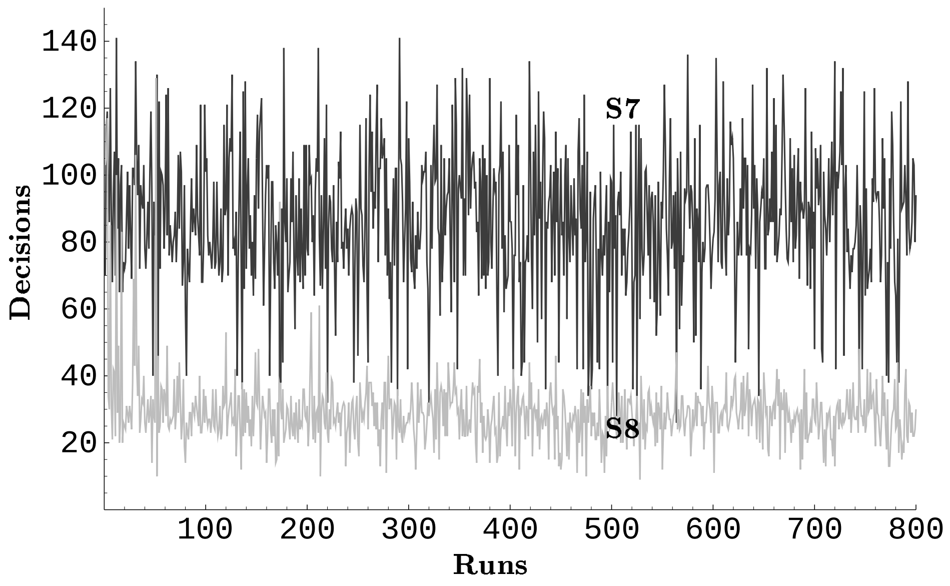


Figure 6.4: Runs *vs.* decisions with compilation in Soar 8.

100 cases, which excludes about three-fourths of the learning. Thus, we can regard the learning curve in the figure as a "post-learning" result. As is clear in the figure, productions firings remain closely correlated to the complexity of the task. When the compiled knowledge supersedes the decomposition, fewer total production firings are necessary, but the resulting production firings are still largely determined by the task complexity. Thus, because the agent must still execute the same number of outputs for any task, the complexity of the task provides a lower bound on the number of production firings.

In the non-learning agents, decisions were also closely correlated with task complexity, as we saw in Figure 5.6. However, recall that decisions are roughly proportional to the number of subtasks selected over the course of the execution of a task. Decisions improve in greater proportion than the production firings in the learning agent because compilation makes decomposition unnecessary for the majority of tasks the agents execute, thus requiring many fewer subtasks. For example, Figure 6.4 shows how the decisions vary over the individual tasks. Initially in the Soar 8 agent, the number of decisions for individual tasks is greater than for the Soar 7 agent. However, within just a few runs, the number of decisions drops significantly, as compiled knowledge intervenes to avoid decomposition. For example, in the Soar 8 agent, the average over the first three tasks is about 109 decisions. However, in the third triplet of tasks, the average has decreased to 31.3 decisions.

When no decomposition is necessary, one might expect the number of decisions to be bounded by the number of blocks to move. For instance, for a task in which all the blocks begin on the table, we would expect one decision for `stack(2,3)`, and one decision for `stack(1,2)`. However, the simulation of block movement is synchronized with the decision procedure, and the agents execute only a single motor command per decision. Therefore, the minimum number of decisions for the learning agents is bounded by the

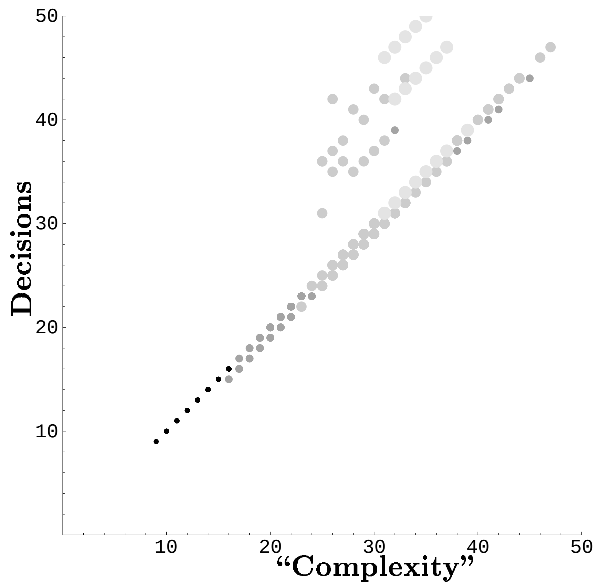


Figure 6.5: Soar 8 decisions *vs.* the sum of outputs and number of blocks for move for (primarily) post-learning cases in the Blocks World. The shade and size of each datum signifies the the number of blocks moved for the run: (smallest, black) $\rightarrow$ 1 block; (larger, dark gray) $\rightarrow$ 2 blocks; (larger, medium gray) $\rightarrow$ 3 blocks; (largest, light gray) $\rightarrow$ 4 blocks.

sum of the number of outputs and the number of blocks to move.[3] Figure 6.5 shows that, post-learning, decisions are roughly equal to this sum. Thus, the learning agent quickly learns to generate close to a minimum number of decisions for each task, as it also learned to fire a minimum number of productions. However, the difference between the decision minimum and the decisions made when not learning were much greater than the corresponding production firing differences, resulting in the greater relative decision improvement in the learning agents.

The results we have examined thus far suggest that the Soar 8 agent achieves significant improvement, reducing production firings by about half and decisions by almost 80%. However, the resulting decrease in total CPU time was only 26%. In the non-learning agent, the average rate of production firings was 2.18 pf/ms, while in Soar 8, the average rate decreased to 1.37 pf/ms, a decrease of about 37%. As the Soar 8 agent learns, its knowledge base almost doubles. Thus, the difference in the rate of production firings suggests the agent is experiencing some utility cost due to the new productions.

There are also two reasons to expect the rate to decrease, independent of utility problems. First, after learning, the runs are much shorter in length. There is a substantial amount of initialization in the Blocks World, because the agent creates an internal representation of the external environment. In one fully-compiled run, chosen at random, one-sixth of the total number of changes to the agent's memory over the course of the

[3]Some primitives take multiple steps to execute. In particular, `move-up` and `move-down` take an argument, n, indicating the number of spaces of move up or down. The execution of this primitive requires n decisions to complete (i.e., each gripper step still requires one decision). We normalized the data in Figure 6.5 to offset the effect of multi-step primitives. However, without this normalization, because only a few multi-step primitives occur in the execution of the tasks, the results are nearly identical.

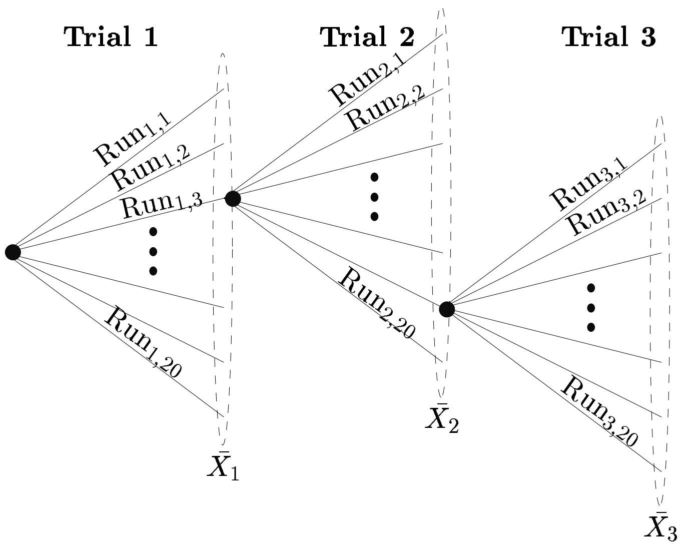


Figure 6.6: Methodology used for μTacAir-Soar compilation experiments.

run occurs in the first decision. The time required for this initialization is amortized over fewer decisions (and production firings) when the agent learns, thus contributing to the decrease in production firing rate. Second, in the learning runs, the agent is, of course, using the compilation process, where it did not in the non-learning runs. Although we argued this algorithm would not be prohibitively costly, it does add to the cost of the executing the task. These effects both contribute to the increased cost of firing individual productions. However, total match cost does increase slightly as well. These results suggest that utility effects may be a concern for external domains, although further, finer-grained experimentation would be necessary to determine the contributing factors to the decrease in production firing rate.

Compilation in μTacAir-Soar

The complexity and nondeterminism of μTacAir-Soar present two issues that make determining the impact of compilation in this domain more difficult than in the Blocks World. First, the agents will not learn the complete task in a single run. In the Blocks World, an agent can fully compile the steps for solving a particular configuration in a single run. Subsequent runs then require no further decomposition. In μTAS, novel situations not experienced in the first run may be encountered in subsequent runs of the same scenario. Second, the knowledge learned in a single run may be different from run to run. This variation contrasts with the Blocks World, where a particular agent performing a specific task would always compile the exact same knowledge. Due to the nondeterminism in μTAS, the agent may experience only a fraction of the possibilities in the scenario; thus, from run to run, the specific experiences of the agent can be different, resulting in differences in the compiled knowledge. These issues make assessing the impact of compilation more difficult because variations in performance from run to run can be attributed both to the nondeterminism in the domain and to previous experience. Over multiple learning scenarios, the content of an agent's compiled knowledge and the agent's performance can vary substantially, leading to potential difficulties in attributing performance differences to the learned knowledge.

Because the agent compiles only a small number of all the rules that could be compiled in the scenario, our goal is to measure the change in performance through a series of learning "trials," in which the agent uses the knowledge learned in previous trials to act in a subsequent trial. We can solve the problem in performance variation as we did in the last chapter, by averaging performance data over a number of runs. However, the knowledge learned in each of these runs is different as well, and we cannot in general, average the actual knowledge learned from run to run. Differences in the compiled knowledge make it difficult to determine what knowledge should be used in the following learning trial. If we randomly select knowledge from runs in the previous trial, large variation in the individual runs results, unless we control for variability across those runs by making a large number of runs for each set of knowledge chosen. However, this strategy requires an exponential number of trials.

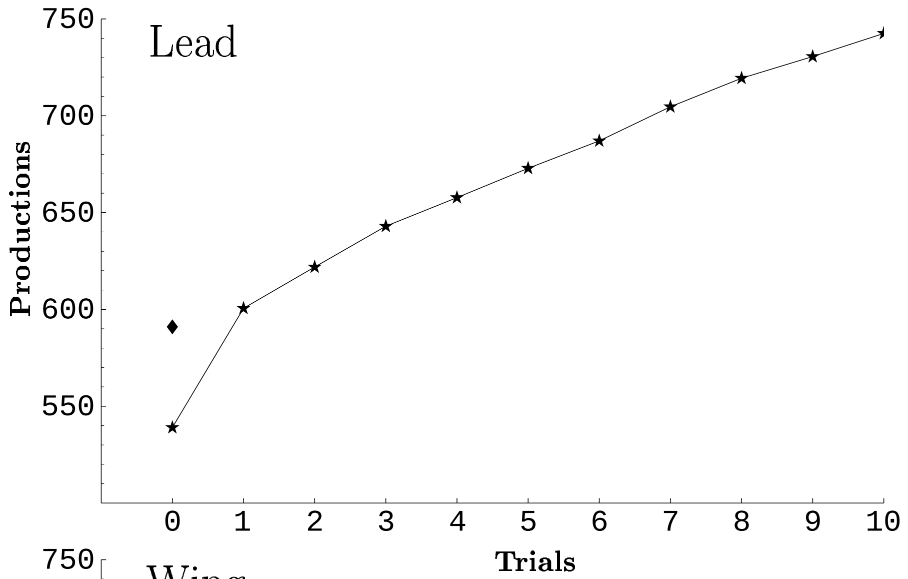
We developed the methodology illustrated in Figure 6.6 to combat this difficulty. We divide groups of scenario runs into trials, as we suggested above. Each trial consists of about 20 runs. We present average performance data from each trial in Table 6.3. Within a trial, each lead agent begins with the same knowledge as the other leads. Similarly, each wing shares the same knowledge as the other wings. The knowledge bases for the leads and wings are identical between particular wing and lead agents only for runs in the first trial. In subsequent trials, the compiled knowledge from *one* of the runs in the previous trial, chosen randomly, along with the original domain knowledge serves as the baseline knowledge base for the next trial. In the figure, the lead agents in Trial #2 use the original domain knowledge and the knowledge compiled in $\text{Run}_{1,3}$. Similarly, in Trial #3, the leads use the knowledge compiled by the lead in $\text{Run}_{2,19}$, which also includes $\text{Run}_{1,3}$ compiled knowledge. Thus, within a trial, in each individual run the agent will compile different experiences with resulting variation in both performance and compiled knowledge. However, having controlled for potential variation in individual experiences within a run, across trials we expect to see a gradual improvement in performance as the agents compile responses to larger portions of the state space.

This methodology may be overly conservative and other methodological approaches are also possible. We developed this solution because it allows us to measure the change in performance across trials while the variation between runs within a trial does not deviate substantially from that of non-learning agents. It also reduces storage requirements because only one set of compiled rules needs to be retained per trial.

Table 6.3 shows a summary of average data for the learning wing and lead agents in μTAS over 10 learning trials, using the methodology we described in Figure 6.6. Importantly, even in the much more dynamic domain, we observed no problems due to non-contemporaneous constraints or knowledge contention using Soar 8. Additionally, we needed to make no changes to the knowledge specifically for learning, other than those we described for over-general and over-specific learning, as we described in Section 6.3.3. These results are significant because they suggest that compilation can be used routinely in dynamic domains, without having to engineer the knowledge for anything other than task performance. However, if we compare the performance metrics to those we summarized in Table 5.2, the results are much less encouraging. Overall performance does not improve relative to the Soar 8 non-learning agent. In this section, we explore the reasons for this lack of improvement while in the following section we will observe that responsiveness does improve significantly as the agents gain experience in the domain.

	Soar 7	Trial 0	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10
Soar 8 Lead												
Learned Rules	0	0	61.57	82.9	104	119	133.9	148.0	166	180	191.6	201
Decisions	8974	8974	8974	8974	8974	8974	8974	8974	8974	8974	8974	8974
Outputs	109.1	142.8	151.7	155.4	160.7	160.0	160.9	160.3	160.8	160.7	160.2	160.4
ms of CPU Time	1683	1030	1026	960	932	1174	937	1093	982	1001	963	1140
Elaboration Cycles	1590	1991	1846	1671	1676	1668	1664	1654	1654	1649	1655	1647
Rule Firings	2438	2064	2104	2071	2085	2077	2076	2071	2074	2071	2070	2066
Number of runs (n)	43	53	21	18	24	20	23	24	20	18	26	57
Soar 8 Wing												
Learned Rules	0	0	33.3	46.2	54.8	62.3	67.9	73.5	79.7	86.4	92.0	99.3
Decisions	8958	8958	8958	8958	8958	8958	8958	8958	8958	8958	8958	8958
Outputs	1704	869	901	903	909	913	910	912	911	915	907	912
ms of CPU Time	12576	2175	1845	2138	2097	2252	2141	2054	2101	2190	2254	2311
Elaboration Cycles	11820	7246	6913	6770	6789	6790	6805	6774	6796	6833	6762	6797
Rule Firings	16540	6321	6212	6493	6505	6529	6513	6506	6513	6545	6467	6522
Number of runs (n)	56	47	18	19	21	14	16	20	19	22	9	34

Table 6.3: Summary of μTAS average run data with compilation.



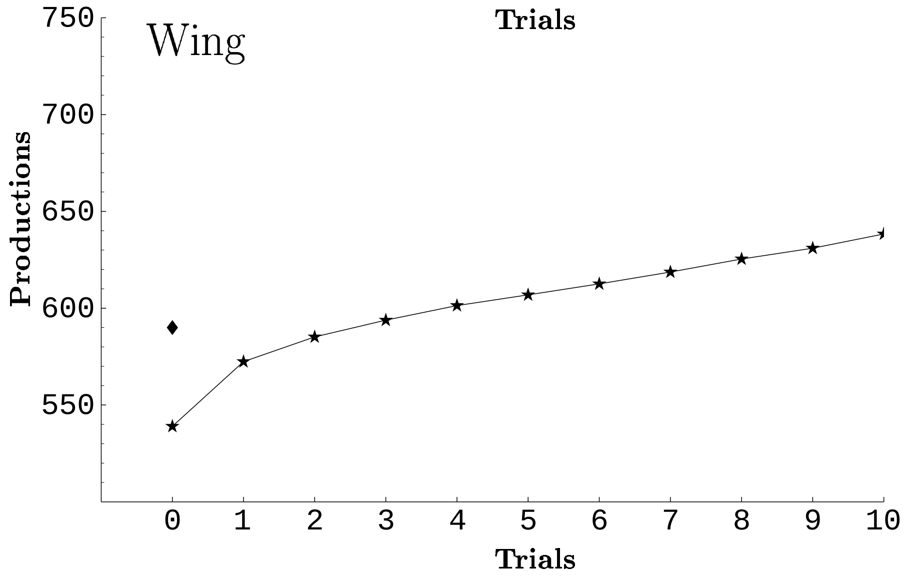


Figure 6.7: Total production rules for the lead and wing agents across μTAS learning trials using Soar 8 (star). The Soar 7 non-learning agent (diamond) is included for reference. The points represent the averages of the production rules at the end of each trial.

We begin by examining the effects of compilation on the total knowledge, illustrated in Figure 6.7.[4] We expect the learning rate to decrease over several trials, as the agents gain experience in repeating the same task over successive trials. This expectation is realized in the first trial, where both the lead and wing learn about a third of the total new production rules learned over the ten trials. However, in further trials, the learning is roughly constant (although the number of new rules per run is decreasing slightly in the later trials). We discovered that both the lead and wing agents are still learning some over-specific rules. For example, when executing the patrol mission, the lead determines when to turn based, in part, on the distance from a reference point. When it executes the turn, the agent also learns a new rule that will execute the patrol-leg turn in the future without decomposition. However, this new rule depends on the exact distance to the reference point and is thus unlikely to be used again.

Consistent with our overall methodology, we made very conservative changes to the agent's knowledge base for avoiding over-specific compilation. In particular, we ensured only the compilation of any rule that changed a particular motor command (e.g., the current heading) did not depend on the numeric value of that particular motor command. In the turning example, however, the decision to turn is based on a different categorical value (distance) and thus we did not change this knowledge. Importantly, the rules that are over-specific do not cause behavior problems or represent incorrect learning; they are simply so specific to the current situation that they have little chance of being used again.

We did not make eliminating over-specific learning a priority. Recent work has shown that chunking can be modified to use more typical explanation-based learning generalization techniques and thus avoid the limitations of the variabilization process in chunking that leads to over-specific rules (Kim and Rosenbloom, 1995). However, the occurrence of over-specific rules in our results does impact the learning rate, as we observed above, and means that our estimate of the changes necessary to address over-specific learning is low for the current implementation of chunking in Soar.

Unlike the decrease we observed in the Blocks World, the number of production firings does not change with learning in the μTAS agents. Figure 6.8 shows the average production firings for the lead and wing agents as a function of the number of trials. In the leads, the production firings increase initially and then return to about the same level as in the non-learning lead. In the wings, the production firings increase slightly, about 2.6%. This lack of improvement occurs even though the agent are compiling new rules throughout the trials.

In order to explain the difference between the Blocks World and μTacAir-Soar results, we again must examine the differences between the tasks in these domains. In the Blocks World, the execution of a primitive action causes a transition to a qualitatively different state: the position of the blocks and grippers have changed such that another action should be generated. The agent is able to speed up its overall behavior because the transition from one state to another occurs as quickly as it can generate the actions to move from state to state. Production firings decrease as a result because the compiled knowledge is able to effect an immediate transition to a new state, thus avoiding the delay necessary for decomposition.

In contrast, when a μTAS agent executes a particular action in the environment, particular values in the state may change quickly (e.g., aircraft heading in a hard turn)

[4]In this figure, as in the ones which follow, Soar 8 non-learning results are included as Trial 0.

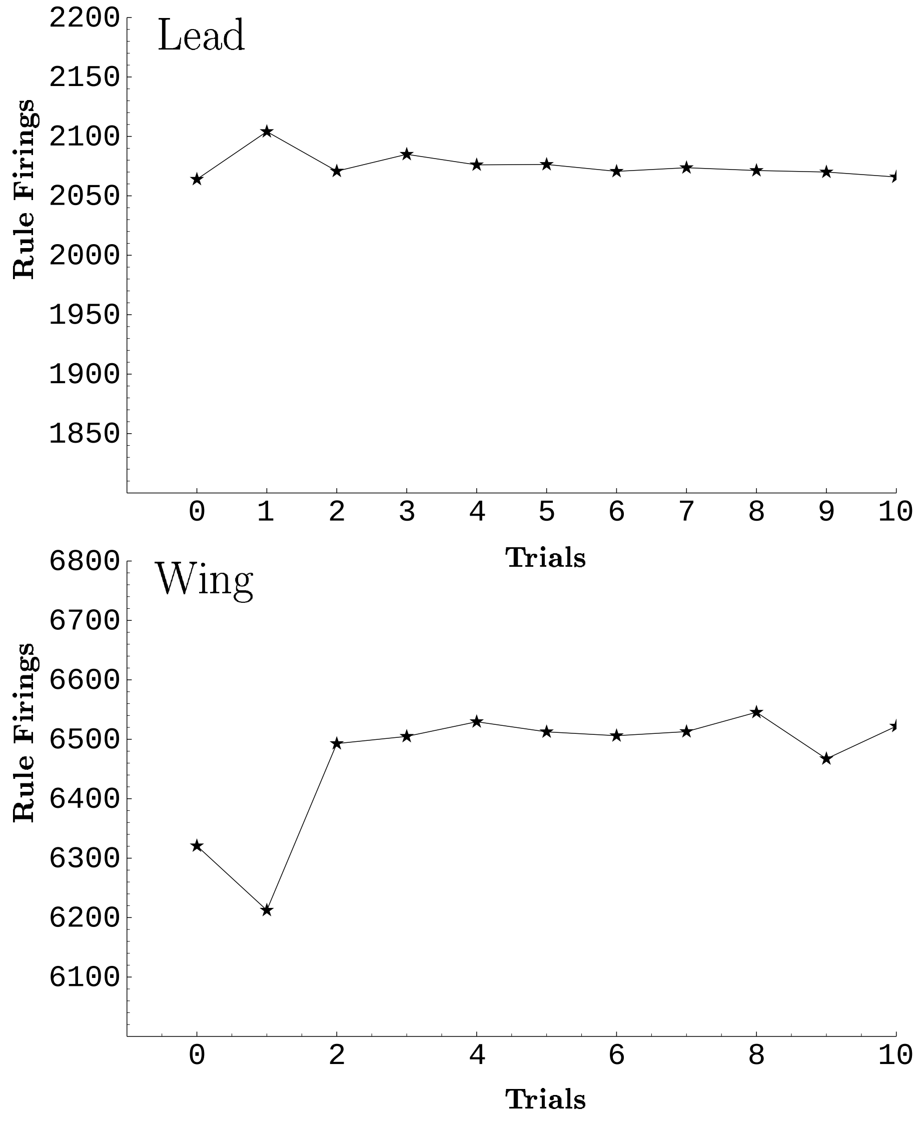


Figure 6.8: Productions firings for the lead and wing Soar 8 agents across the μTAS learning trials.

but the qualitative state changes at only a few points over the scenario, such as when the lead finishes a patrol leg or when an enemy plane enters into missile range, as we observed in Figure 5.9. Although the agent may execute many primitives in order to effect a change to a different qualitative state (e.g., maneuvering to get into missile range), it takes a long time, relative to an agent's reasoning cycle, for an agent to effect a change in the qualitative state.

When qualitative states change slowly, the agents have opportunity to access their original domain knowledge. Production firings do not decrease in μTAS because the agent generates its hierarchical reasoning while waiting for the transition to a new state. By generating the task hierarchy, the agent can ensure that its current course of behavior, already initiated by the compiled knowledge, is consistent with the domain knowledge. In some cases, the current course of action will be further elaborated by the decomposition knowledge, leading to both an increase in the total production firings and the compilation of new rules. Because the agent has decomposed the problem, it can respond immediately if some new situation arises for which it currently lacks domain knowledge. Thus, the use of the original domain knowledge is not a liability. Further, as we will see in the next section, responsiveness does improve with compilation because the compiled knowledge is applied *before* the task is decomposed.

Figure 6.9 suggests this additional elaboration is occurring in μTAS. The number of outputs increases as learning progresses, by about 12% in the lead and 5% in the wing. Because the agent need not delay reaction while decomposing the task for the current situation, the agent can react more quickly and in some cases more often than it could previously. Thus, the μTAS agents' lack of improvement in production firings is attributable to the domain rather than the agent architecture or compilation. Moreover, decomposition occurs after the execution of primitives and thus action itself is no longer delayed by the decomposition, as we will see in the following section.

Because production firings increase slightly, and the agent is invoking the compilation process as well, we now expect to observe a slight increase in CPU time across trials. Figure 6.10 shows the actual average change in CPU time across the learning trials. Although the overall increases in CPU times are reasonable (10.6% in the lead, 6.3% in the wing) and all within a single standard deviation of the Trial 0 CPU time, the average CPU time does appear to be increasing slightly with additional learning trials. For instance, the number of production firings is roughly constant over the final 5 trials while CPU time is increasing almost linearly in the wing and in the lead as well, although less regularly. Additionally, unlike the Blocks World, the increase in CPU time cannot be attributed to either a decrease in decisions or the learning process itself. If the compilation process was causing the increase, we would expect to see a larger increase in the first trial, where about about twice to three times as many total rules are learned as compared to the subsequent trials. These factors suggest that the increase in CPU time is due the increase in the number of the learned rules. In other words, the agent may be experiencing a slight average growth effect. However, without extensive profiling of the system, we can not say with certainty if the increase in CPU time is due primarily to average growth effects. We will return to this subject when we consider directions for future work, in Chapter 6.

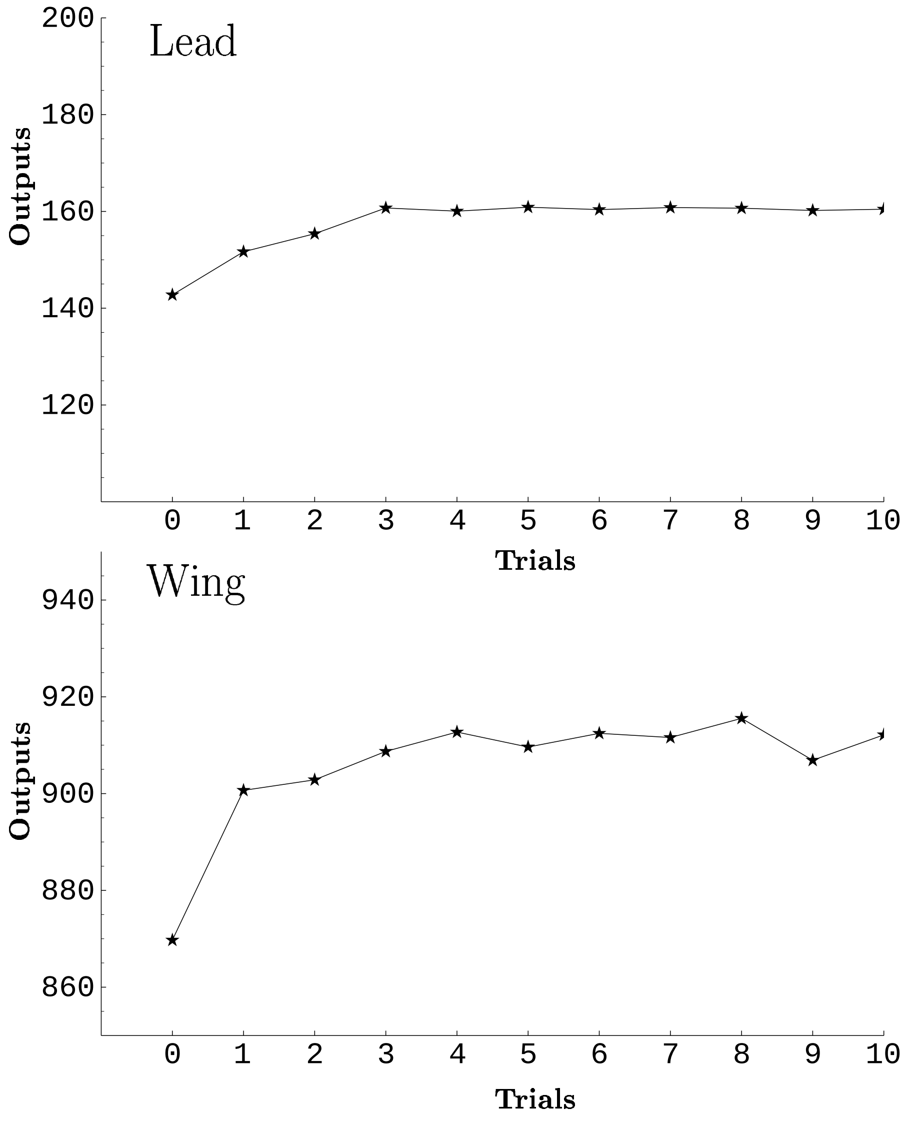


Figure 6.9: Number of primitive actions taken by the lead and wing Soar 8 agents across the μTAS learning trials.

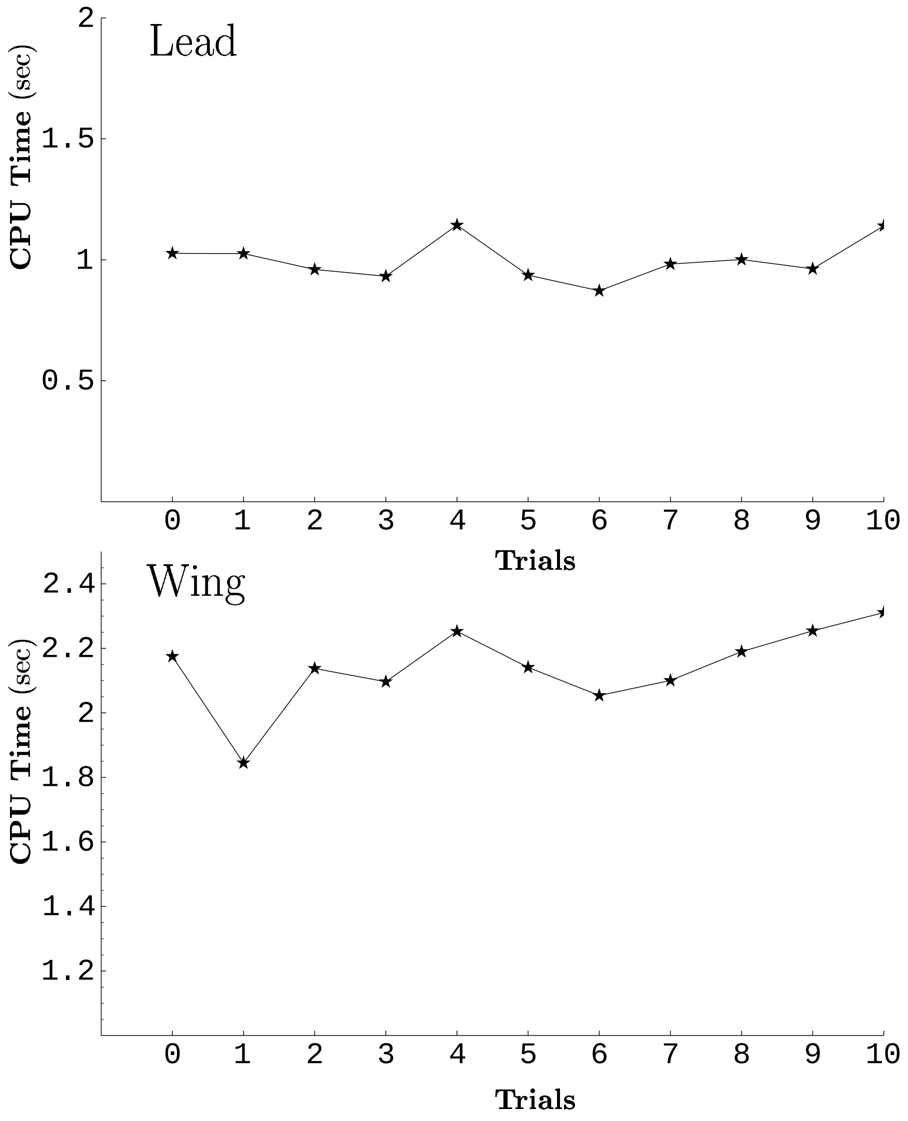


Figure 6.10: The change in the CPU times (in seconds) of the lead and wing Soar 8 agents across the μTAS learning trials.

	Avg. In-Range Time	Avg. Launch Time	Reaction Time	n
Soar 7	161.816	162.084	.268	95
Soar 8				
Trial 0	162.048	162.994	0.945	99
Trial 1	161.927	162.891	0.964	39
Trial 2	161.493	162.050	0.557	41
Trial 3	161.327	161.830	0.503	45
Trial 4	161.377	161.757	0.380	42
Trial 5	161.644	161.889	0.244	40
Trial 6	161.458	161.617	0.159	46
Trial 7	161.260	161.406	0.146	39
Trial 8	161.457	161.584	0.127	40
Trial 9	161.245	161.370	0.125	37
Trial 10	161.450	161.552	0.103	107

Table 6.4: The effect of compilation on reaction time in μTAS. Each row of the table shows the average times (in seconds) for successive learning trials. Each trial was averaged over n runs.

Effect of Compilation on Responsiveness

We now examine the impact of compilation on responsiveness. In the previous chapter, we observed that responsiveness can actually decrease in Soar 8 even though overall performance improves. We hypothesized that compilation would allow the agent to improve responsiveness both by avoiding the delays due to decomposition and also by composing the actions of individual subtasks into more complex behaviors.

As we did in the last chapter, we examine one specific example of responsiveness, the reaction time for launching a missile. Table 6.4 shows average reaction time results from the individual trials, along with the results from Table 5.4, reproduced here for convenience. Figure 6.11 illustrates the change in average reaction time with learning. As these results indicate, average reaction time increases substantially with learning in Soar 8. In the first trial, there is no improvement. In each individual scenario, there is normally only a single opportunity to launch a missile. Thus, in all the Trial 1 runs, the agent uses only its domain knowledge to execute the task. Just one learning opportunity decreases the reaction time by 40% in the second trial. By the fifth learning trial, reaction time in Soar 8 has improved in comparison to Soar 7. In the final learning trial, reaction time has decreased to about one-tenth of a second, a 61% improvement in comparison to the Soar 7 agent.

The improvement in reaction time is attributable to two different factors. First, after compilation, the agent is able to recognize the conditions under which a missile should be launched without referring to the hierarchy. Thus, the agent has gained more operational knowledge of when to fire a missile, as anticipated, and no longer delays the initiation of the missile launch for decomposition. Second, individual actions have been composed into operators higher in the hierarchy and no longer require individual selection and application. Thus, multiple actions can be taken within a single Soar decision. Significantly, these results show that although overall performance does not improve in μTAS, responsiveness can improve substantially with compilation thus improving the quality of agent behavior. Further, the potential degradation in responsiveness using Goal-Oriented

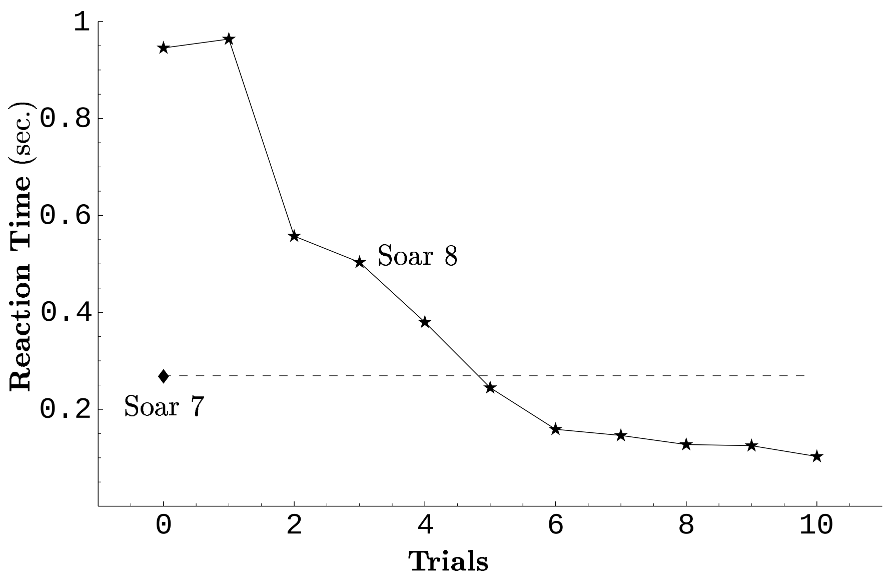


Figure 6.11: Improvement in reaction time with learning.

Heuristic Hierarchical Consistency can be successfully avoided as well.

6.5 Summary

The results of our empirical investigation of compilation are not as decisive as those of the previous chapter. Figure 6.12 shows compilation results from the Blocks World in terms of the overall performance and knowledge dimensions. Figure 6.13 similarly illustrates the results from the lead and wing agents in μTacAir-Soar. In both the Blocks World and in μTacAir-Soar, we were able to use compilation with minimal additional knowledge engineering cost because Goal-Oriented Heuristic Hierarchical Consistency provided solutions to both the non-contemporaneous constraints and knowledge contention problems. The learning agents automatically acquired new productions, leading to an increase in the total number of productions in the agent but not an increase in total knowledge design cost.

In the Blocks World, average CPU time decreases, indicating an overall performance improvement, but we observed that the decrease in CPU time was modest in comparison to the decrease in production firings and decisions. In μTAS, overall performance does not improve. The lack of overall improvement can be explained, in part, by the agent's continued use of its hierarchical knowledge in conjunction with compilation. As we saw in Figure 6.11, responsiveness does improve with compilation. Thus, even when overall performance does not improve, the agent is able to react to the current situation without the hierarchy and thus improve behavior.

Although we did observe some performance improvement, we also were concerned that the average growth effect was contributing to some slowdown in the average rate of matching and firing individual productions. Determining whether or not the average growth effect is leading to a degradation in performance in these systems is a subject of

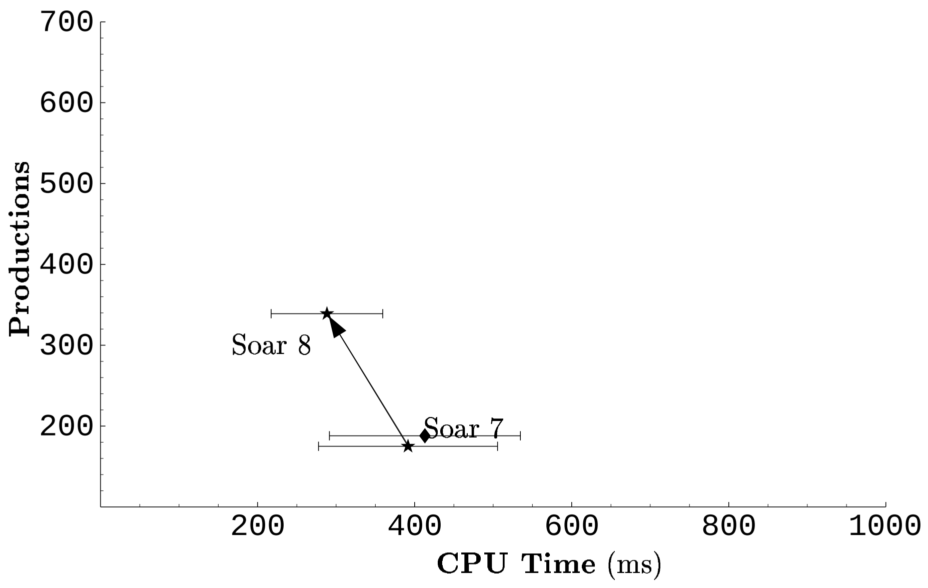
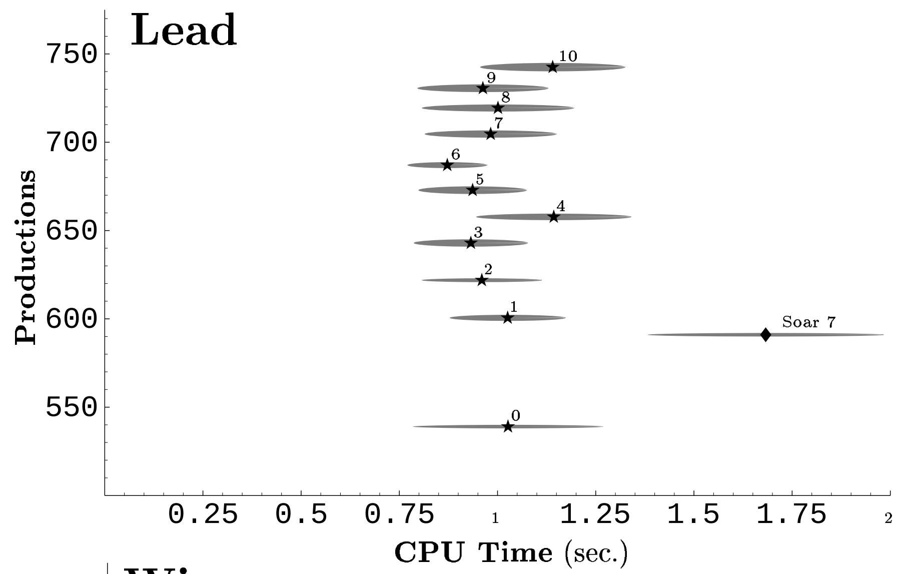


Figure 6.12: Blocks World Summary: Mean CPU Time in milliseconds *vs.* knowledge in productions for Soar 7 (diamond) and Soar 8 (star) agents.

future work. Evidence from many previous systems suggests that utility problems can be avoided in Soar. However, these previous systems have generally not tried to use learning with an existing knowledge base while also attempting to minimize changes to that knowledge base. Additional work may show that the use of compilation does require additional revision to existing knowledge bases to avoid over-specific learning (which causes increases in match cost) and over-general learning (which causes incorrectness in behavior).

Significantly, however, our experiences show that compilation can be used "on line": agents can learn while also behaving in dynamic domains with little detriment to their overall performance. For example, even over ten learning trials in μTAS, the average CPU time during the tenth trial was still well within the standard deviation of the average CPU time prior to learning. Compilation not only allows an agent to act without delaying for decomposition, it also allows it parallelize and compose it actions, releasing the initiation of actions from the seriality of the architecture's subtask selection process. Thus, assuming the technical problems we introduced can be addressed, compilation allows significant improvement in performance with experience, while maintaining the low knowledge design cost of hierarchical agents.



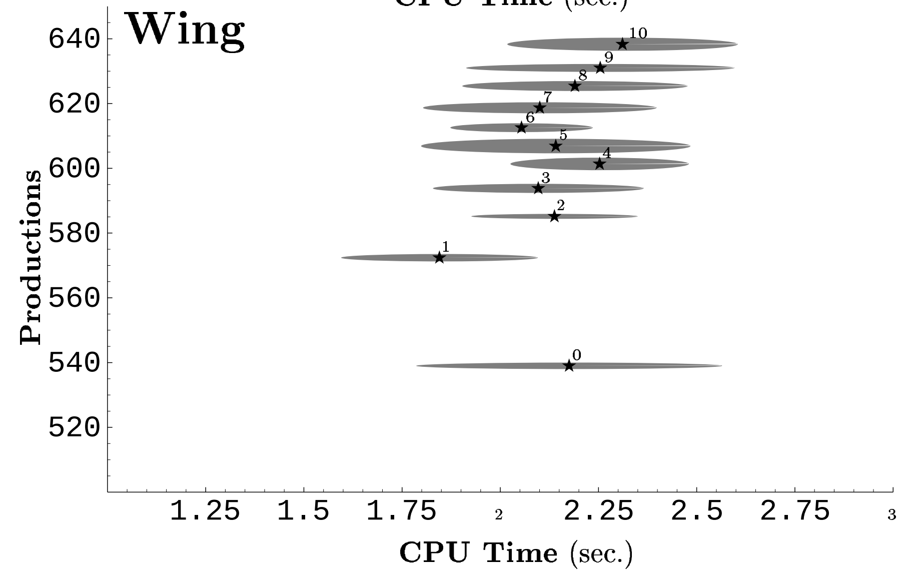


Figure 6.13: μTacAir-Soar Summary: Mean CPU Time in seconds *vs.* knowledge in productions for Soar 7 lead agent (diamond) and learning trials for the Soar 8 lead and wing agents (star). The Soar 7 wing at {12.6 sec, 591 productions} is not shown. Each data point is surrounded by an ellipsis of horizontal width equal to one standard deviation in CPU time and height of one standard deviation in the new knowledge for that trial. The 0'th trial is the Soar 8 non-learning run.

Chapter 7

Summary and Future Work

There is more to life than increasing its speed.
– Mohandas Gandhi

In the previous chapters, we have developed and empirically tested an efficient, architecture-based solution that ensures processing consistency in agents employing hierarchical task decompositions. This solution allows agents to act reliably in complex, dynamic environments while also maintaining the low cost of agent development provided by hierarchical task decomposition. This solution also solves several extant problems in knowledge compilation, thus enabling compilation in complex, dynamic environments which leads to improvements in agent performance.

In this chapter, we explore a number of additional aspects of this research. First, we summarize the contributions of this work, considering especially the generality of the methods and solutions we have described in the previous chapters. Second, we describe the impact of this work on the Soar architecture, both in terms of its impact on existing Soar systems and the repercussions on Soar as the implementation of a psychological theory. Finally, we describe some potential future directions for this research, summarizing and expanding some of the ideas we have presented in previous chapters.

7.1 Contributions

In this section, we revisit the contributions we introduced in Chapter 1. The primary contributions of this thesis have been to:

- Provide an understanding of how inconsistency can arise due to persistence in hierarchical architectures and develop potential solutions that efficiently solve this problem.

  Inconsistency in processing can arise when the context changes and an agent fails to update a persistent assumption in the subtask hierarchy that is logically dependent on the changed value in the hierarchy. Many solutions to this problem rely on specific knowledge to resolve the inconsistencies, which can be difficult to identify, costly to create, and provides no guarantee of consistency in all circumstances. We introduced two new solutions, Assumption Justification and Dynamic Hierarchical Justification, that require no additional knowledge because the determination of consistency is made by the architecture. Both solutions limit persistence by justifying local assumptions with respect to the hierarchical context. This partial justification allows local nonmonotonic reasoning while ensuring that local assumptions remain consistent with the hierarchical context. Dynamic Hierarchical Justification avoids some of the computational expense of Assumption Justification by associating dependencies with subtasks rather than individual assertions.

- Provide an understanding of how inconsistency can arise due to multiple, simultaneous threads of reasoning in a hierarchical architecture and develop solutions that efficiently solve this problem.

  Inconsistency in processing can also arise when the agent reacts to a context change in a local level of the hierarchy before all the ramifications of the context change have been elaborated. Reacting too quickly can cause some inefficiency, because newly asserted knowledge may be retracted as soon as the context is fully elaborated. More problematically, nonmonotonic assertions, such as the initiation of motor commands, can lead to irrational behavior. Again, previous solutions to this problem use agent knowledge to avoid inconsistency. We introduced a solution, Subtask-limited Reasoning, that sequences the reasoning in the hierarchy, allowing multiple local threads of reasoning but serializing threads of reasoning across subtasks. Subtask-limited Reasoning avoids the computational expense of computing dependencies between different assertions by recognizing that progress in a subtask can not proceed until the agent is certain that the subtask remains a valid task in the current situation.

- Provide an empirical analysis of Goal-Oriented Heuristic Hierarchical Consistency, a combination of Dynamic Hierarchical Justification and Subtask-limited Reasoning.

  The analysis of Dynamic Hierarchical Justification and Subtask-limited Reasoning suggested that these solutions could guarantee consistency without sacrificing performance in execution. However, these solutions are heuristic: they simplify the computation of dependencies in reasoning between assertions by taking advantage of the locality of assertions in the hierarchy. These simplifications have the potential to increase the time necessary to execute a task, even though the individual procedures are inexpensive. Therefore, we evaluated an implementation of the complete solution, Goal-Oriented Heuristic Hierarchical Consistency, in two different domains. Our results show that, at least for the tasks we evaluated, GOHHC can actually reduce the time cost of execution, because the overall knowledge requirements are reduced, leading to less knowledge accessed over the course of executing the task. This evaluation confirms GOHHC does indeed provide an efficient solution to inconsistency and thus may be a worthwhile solution for other architectures.

- Introduce a new methodology that allows comparison of a new architecture to a baseline architecture by comparing the relative behavior of agents implemented in the architectures.

  We wanted to evaluate GOHHC according to its knowledge requirements, performance and responsiveness. However, these criteria are strongly determined by specific tasks as well as specific architectures. Therefore, in order to evaluate GO-HHC, we compared GOHHC agents to agents performing the same tasks in a "non-GOHHC" architecture using knowledge-based solutions for inconsistency. This methodology allows one to constrain the degrees of freedom in agent design and thus reduce the possible variation in results. However, this methodology requires agents for identical tasks developed in two different architectures. We avoided some of this disadvantage by choosing previously-existing agents, already optimized for behavior, and made conservative changes to those agents for the GOHHC architecture. We looked at only two domains. However, we chose these domains deliberately. The Blocks World is a synthetic, endogenous, and relatively simple domain. μTacAir-Soar is a real domain (albeit a simulation environment), highly dynamic and exogenous, and relatively complex. Thus, we chose two tasks at the extremes of these particular dimensions. Additional task dimensions could be explored as well, as we discuss below. However, this methodology should be useful as a general tool for the assessment of the impact of new architecture capabilities and features.

- Provide an understanding of how inconsistency in an agent's processing can lead to specific problems in compilation and show that Goal-Oriented Heuristic Hierarchical Consistency provides a guarantee of consistency sufficient for on-line compilation of subtask processing in the hierarchy.

  Inconsistency in processing also makes knowledge compilation difficult. We described two specific problems, non-contemporaneous constraints and knowledge contention, and showed that GOHHC solves these problems because it guarantees consistency in reasoning. Importantly, we showed that consistency also eliminates much of the specific knowledge representation requirements for learning, thus allowing the compilation of execution knowledge not specifically designed for learning.

  We employed compilation to improve performance in agent domains. We learned that overall performance is not as important as responsiveness in some domains. In particular, compiled knowledge can be used to generate responses immediately, without incurring the cost of decomposition while hierarchical reasoning can evaluate the behavior generated by the compiled knowledge, ensuring behavior agrees with an agent's goals. In this situation, overall performance as measured by CPU time is not expected to improve, but the qualitative behavior improves.

  Compilation improves performance as long as the utility problem and, more specifically, average growth effects, are not overly problematic. We observed a slight potential average growth effect in our empirical results with compilation but not one that slowed reasoning more than a standard deviation from the non-learning baseline. However, further understanding how new knowledge can impact the responsiveness of the architecture is a subject of future work, as we outline below.

How applicable will these analyses and solutions be for other architectures? Our analysis of the sources of across-subtask inconsistency may make other researchers aware of similar issues in their own architectures. As we described in Chapters 3 and 4, oftentimes these problems are viewed as requirements on the agent's domain knowledge. Our analysis points out that this knowledge is expensive to develop, degrades the modularity and simplicity of the hierarchical representation, and is only as robust as the knowledge designer's imagination. When agents are developed in sufficiently complex domains, the expense of creating this knowledge may grow prohibitive, and lead researchers to consider architectural assurances of consistency.

The introduction of new capabilities in the architecture, such as compilation, may also motivate architectural solutions to consistency. We were led to consider the issue of inconsistency by our desire to use compilation in Soar in execution environments without having to structure domain knowledge specifically for learning. Of course, our specific solutions may not be useful for other architectures, which include different capabilities. However, we attempted to reduce the specificity of our results to Soar in two ways. First, we explored a space of solutions to the particular problems and outlined the costs and benefits of each. For other architectures, some of the solutions not optimal for Soar may not be as costly and can be adopted. Second, the specific solutions we chose were based both on the architecture and on the structure of hierarchically decomposed tasks. Although the specific implementations and procedures may be different for other architectures, the heuristic simplifications employed by Goal-Oriented Heuristic Hierarchical Consistency should transfer to any architecture utilizing hierarchical task decomposition.

7.2 Impact on the Soar Architecture

In this section, we describe some repercussions of Goal-Oriented Heuristic Hierarchical Consistency on the Soar architecture, in addition to the general results we described in Chapters 5 and 6. We concentrate specifically on the compatibility of GOHHC with existing systems and the impact of GOHHC on Soar as a psychological theory. Although this section will be of primary interest to those who use Soar, this discussion suggests potential impacts the inclusion of Goal-Oriented Heuristic Hierarchical Consistency may have in other architectures as well.

7.2.1 Compatibility with Existing Systems

Soar has been used as the basis for a large number of artificial intelligence systems and cognitive models. Throughout its history, the architecture has been modified to accommodate new features and capabilities (Laird and Rosenbloom, 1995). In this section, we consider how currently existing Soar systems will be impacted by the incorporation of the Goal-Oriented Heuristic Hierarchical Consistency in the architecture.

As we discussed in Chapter 5, we converted two existing systems designed for the current architecture for the GOHHC version of Soar, Soar 8. Neither of these systems was designed to learn. As we observed, the conversion to Soar 8 for these systems could be substantial, especially as measured by the number of rules that needed to be added, deleted, or changed. For instance, μTacAir-Soar required some change to about 40% of the original knowledge base.

These experiences suggest that converting existing Soar systems to Soar 8 will be costly. However, there are two reasons to anticipate less costly conversion. First, using the number of rules changed as a metric leads to overestimates of cost. Many of the required changes can be made to an existing system automatically, without human analysis (e.g., the deletion of termination conditions for operators). Further, as we pointed out in Chapter 5, some of the changes were required to repair problems in the original decomposition. Regenerations simplify the identification of these problems.

Second, conversion cost should be significantly less expensive for existing Soar systems that were designed to learn. In non-learning systems, only those inconsistencies that manifest themselves behaviorally are usually identified and fixed. However, as we described in Chapter 6, systems designed for learning are necessarily more thorough in removing inconsistency. Thus, we expect these systems can be converted with much less expense. For example, although we do not report the results directly in this thesis, we have developed additional Blocks World agents for two representational conventions, or *modeling idioms* (Lallement and John, 1998), that make learning unproblematic. The emphasis of the modeling idioms is psychological rather than functional, although both do provide knowledge-based solutions to inconsistency (Lehman et al., 1995; Rieman et al., 1996). The Blocks World agents using these conventions were developed for Soar 7. However, in both cases, these agents also run under Soar 8 with virtually no modification. Although we do not expect every existing Soar 7 learning system to run immediately under Soar 8, these experiences suggest that the conversion to Soar 8 for systems designed for learning in Soar 7 will be not be as extensive as required by the examples in this thesis.

7.2.2 Effect on Psychological Theory

Soar has been used to model psychological phenomena and has been proposed as a unified theory of cognition (Newell, 1990). We now briefly consider the potential impact of GOHHC on Soar as a psychological theory and tool for psychological modeling.

The basic commitments of the Soar architecture – associative retrieval of knowledge, problem spaces, impasse-driven subgoaling, and chunking – are unchanged in Soar 8. Psychological models built using Soar usually describe phenomena at this level of description and should be unaffected. For example, as we saw above, learning systems, which include many psychological models built using Soar, may run with only minor modification in the GOHHC version of Soar. As another example, models of dual-task performance implemented in Soar 7 (using a particular modeling idiom) and in Soar 8 have both been shown to match human performance data closely (Lallement and John, 1998). In general, most psychological phenomena modeled with Soar will not intersect with the changes necessary for Goal-Oriented Heuristic Hierarchical Consistency.

One seemingly possible impact of the GOHHC architecture is that it will force Soar to always behave "rationally." Psychological research shows that people often reason inconsistently, even when the total knowledge available to them would allow correct conclusions. For example, Tversky and Kahneman (1982) relate many examples of situations in which subjects make errors in judgment under uncertainty. In one experiment, after observing a series of coin tosses in which the coin consistently shows "tails," subjects, when asked the probability of the next toss resulting in "heads," consistently choose a probability greater than one-half, even if told the coin is fair.

Superficially, these results appear to conflict with Goal-Oriented Heuristic Hierarchical Consistency. However, GOHHC enforces consistency in processing – the manipulation of symbols – but says nothing about potential inconsistencies in the interpretation and use of those symbols. Tversky and Kahneman attribute the errors in judgment to the information available when the judgment is made (not to some inherent error in the processing). This information then biases the resulting judgment. Although modeling the effects of such judgments would probably involve details of Soar's architectural commitments, Goal-Oriented Heuristic Hierarchical Consistency alone would not preclude the use of Soar for such a model. The information and biases in the knowledge would be the strong determiners of behavior, rather than the underlying manipulation processes.

However, some details of the architecture have changed under GOHHC and may have some implication for psychological modeling. As one example, consider post-completion errors (Byrne and Bovair, 1997). Post-completion errors are errors in behavior that arise when a goal is completed but some cleanup is necessary following the achievement of the goal. For example, when making a photocopy, one might forget to retrieve the original along with the copy. People find it very difficult to learn to avoid post-completion errors and they appear across tasks and domains. These characteristics suggest an architectural source of post-completion errors.

Soar 8 might provide a parsimonious explanation of post-completion errors. Recall from Chapter 5 that Soar 8 automatically terminates subtasks when their initiation conditions are no longer satisfied. Thus, Soar 8 would predict that post-completion errors arose because the `make-copy` goal is retracted as soon as the goal is achieved. This termination contrasts with knowledge-mediated termination in the current Soar architecture.

A simple model of `make-copy` might include a proposal for the subtask with very general conditions (e.g., "need copy"). Then, when the copy is available, the goal is satisfied and automatically removed by the architecture, causing the post-completion error. Learning to remember to retrieve the original would be difficult because it would require specialization of the original subtask initiation conditions to include the original in the achievement of the goal ("need copy *and* original"), even though the original is already possessed when the goal is initiated.

Fully understanding the role of GOHHC in the Soar's psychological theory is an open question. However, this discussion points out our expectation that GOHHC should have little impact on that theory because the major architectural commitments used in computer simulations of psychological phenomena are unchanged.

7.3 Directions for Future Work

Future work can be divided into two distinct categories. The first category widens the scope of our evaluation of Goal-Oriented Heuristic Hierarchical Consistency by exploring different architectures and tasks. The second category deepens the analysis within Soar in order to understand more completely how GOHHC interacts with the Soar architecture.

7.3.1 Further Investigations in Breadth

- Implement and evaluate GOHHC in other plan execution architectures.
  Our evaluation necessarily concentrated on a single architecture. However, one obvious next step would be to implement Goal-Oriented Heuristic Hierarchical Consistency in another plan execution architecture. The goal of a new implementation would be to further understand the requirements and costs for both Dynamic Hierarchical Justification and Subtask-limited Reasoning.

  A candidate architecture should have several important features in addition to using hierarchical task decomposition and supporting persistence and multiple threads of reasoning. For example, in order to compute dependencies for Dynamic Hierarchical Justification, the architecture should support dynamic access to assertions outside of the local subtask. Similarly, Subtask-limited Reasoning requires that the architecture be able to determine (inexpensively) to which subtask a particular assertion is local. An efficient match process is not necessarily required because the Soar's RETE matcher was critical for utilizing compilation, rather than GOHHC. Because the GOHHC heuristic is based on the structure of the task, as we described above, we would expect similar results in evaluating another architecture, although the specific magnitudes of cost and performance differences would be different. Implementations in other architectures would thus lead to a more complete understanding of the benefits of Goal-Oriented Heuristic Hierarchical Consistency, irrespective of the underlying architecture.

- Evaluate GOHHC along additional task dimensions.
  Our evaluation concentrated on a few tasks. We argued above that the differences between the Blocks World and μTacAir-Soar represented two extremes along a few dimensions. However, additional tasks would allow us to further explore GOHHC and characterize it more completely. For example, in both the Blocks World and μTacAir-Soar, there are only a few high-level goals (e.g., `stack`, `patrol`). Another task class to evaluate would be one with more high level goals, and more switching between those goals.

- Develop better metrics for design cost, performance, and task complexity.
  Our overall conclusions from the evaluation of Goal-Oriented Heuristic Hierarchical Consistency are weakened by the lack of concrete metrics for cost, performance, and task complexity. For example, we hypothesized that making changes to an existing knowledge base was less costly than creating a knowledge base with a similar number of rules. Intuitively, this assumption seems correct but without appropriate metrics we could not draw quantitative conclusions. Drawing on results from software engineering, more descriptive quantitative metrics could be developed, although they might be restricted to particular architectures or domains. However, even with this restriction, more descriptive metrics would provide a way of exploring quantitative relationships between cost, performance, and task complexity, such as the one we hypothesized in Figure 2.7.

7.3.2 Further Investigations in Depth

- Assess the impact of GOHHC on the RETE algorithm.
  We observed in Chapter 5 that some of the improvement in overall performance using Soar 8 could be attributed to lower match cost. We hypothesized that this decrease in cost might be explained by additional constraint in the Soar 8 agent's knowledge. Therefore, one question to explore for future research would be to determine if this result is consistent across other tasks. If confirmed, this result would indicate GOHHC leads to improvements in match cost (with RETE) in addition to to the other improvements made possible by Goal-Oriented Heuristic Hierarchical Consistency.

- Explore average growth effects for compilation in dynamic domains.
  We observed slight increases in CPU time with compilation in the μTAS agents. Is this increase due to an average growth effect? If so, what properties of the task, architecture, or, less plausibly, GOHHC are causing the increase in match cost? The answers to these questions will be important for further use of compilation in dynamic environments.

- Evaluate maintainability of Soar 8 agents.
  Maintainability has previously been identified as a problem for large Soar systems. Goal-Oriented Heuristic Hierarchical Consistency eliminates the need for consistency knowledge that reasons about the interactions between the knowledge in different levels of the hierarchy. Thus, the removal of this knowledge should make the knowledge of Soar agents more modular and maintainable. Confirming this hypothesis would provide further motivation for adopting GOHHC in other architectures.

Finally, we also showed in this research that Goal-Oriented Heuristic Hierarchical Consistency solves the non-contemporaneous constraints and knowledge contention problems, allowing on-line compilation of behavior in dynamic domains. In Chapter 6 we showed that compilation could provide improvements in responsiveness and overall performance. Additional directions for research in compilation include exploring more fully the use of on-line compilation in external environments. For example, we described using an EBL version of chunking to avoid over-specific compilation in Chapter 6 and further characterization of average growth effects above.

Non-problematic, on-line compilation also facilitates the inductive use of learning in execution environments. For example, chunking can be used to compile inductive problem solving, resulting in inductive learning (Rosenbloom et al., 1988). Importantly, compilation provides the potential for knowledge-rich, computationally efficient inductive learning that can occur in conjunction with behavior. Inductive learning could be used to modify behavior with changing domains, self-correct errors in execution knowledge, etc. Although explorations of inductive learning have not yet been attempted in this framework, Goal-Oriented Heuristic Hierarchical Consistency provides an important stepping-stone towards the achievement of these goals.

Appendices

Appendix A

Typography Conventions

This appendix describes the typographical conventions used in the text of the dissertation.

Representation	Font	Example
Relations input from external environment	*slanted*	*clear*
Relations internal to the agent	sans serif	empty
Objects	**bold**	**block-1**
Subtasks/operators in the hierarchy	`typewriter`	`put-on-table`
Problem Spaces	SMALL CAPS	STRUCTURE

Appendix B

Summary of Assumptions

Assumption 1: Experiment in simulation domains. The difficulty inherent in building physical agents has the potential to distract from the goal of addressing the limitations of hierarchical decomposition for plan execution systems. Further, the complexity of both domains and tasks make complete formalization of the characteristics of hierarchical decomposition intractable. Therefore, we will pursue an empirical characterization of hierarchical decomposition and limit these investigations to simulation domains where the engineering effort for the interface is more easily controlled. Support for this assumption includes (Hanks et al., 1993), which suggests that simulated "test beds" are a good choice for empirical studies because the experimenter has control over the underlying domain. As a further control, we will use simulation tasks designed independently of this research.[1]

Assumption 2: Focus on the execution level. The complexity of agent design for interactive, real domains makes simplification necessary. For our empirical investigations, we assume that agents have all the execution knowledge necessary for their tasks, and thus no planning layer is necessary. A number of successful execution systems have been built without a planning component (Bonasso et al., 1997; Georgeff and Lansky, 1987; Pearson et al., 1993; Tambe et al., 1995). We also assume that our agents can treat reactive skills as primitives by making them directly executable in simulation. Thus, no control layer will be used. We will be careful to point out when the specifics of some technique impacts processing in either the planning or the control layer but we will ignore these layers for the most part in our analysis and experiments.

Assumption 3: Focus on a particular implementation. Although we are interested in the general characteristics of hierarchical task decomposition, we will focus our empirical experimentation in one plan execution system: the Soar architecture (Laird et al., 1987). We originally became interested in the characteristics and problems of hierarchical task decomposition through the use of Soar in a number of different plan execution environments constrained by **Assumption 2** (Laird and Rosenbloom, 1990; Pearson et al., 1993; Tambe et al., 1995)). Given the complexity inherent in both domain and architecture, we decided to limit our actual

[1]The specific methodology used in the thesis is discussed in Chapter 5.

implementations to this single architecture. Where possible, we will point out how some particular analysis applies to other architectures but we will not attempt to implement our approaches in different architectures.

Assumption 4: Focus on tasks that are (nearly) decomposable. Hierarchical task decomposition is based on the premise that tasks can be broken down into discrete units that have little interaction with other units. Simon (1969) argues that hierarchic structure is a natural consequence of evolutionary processes; a hierarchy simply provides stable, intermediate structure during the design process, offsetting increasing complexity. This intermediate structure gives hierarchical decomposition the advantages for execution we describe below. In this research, we assume that the tasks we will be addressing are *nearly decomposable*. A fully decomposable system has no interaction with other units in the system while a "nearly decomposable" system has limited interaction among different units. For example, in the Blocks World, the tasks of picking up a block and putting a block down have little interaction; each can be executed with no reference to the other. On the other hand, the `put-down` operator will need to know where to place the block, leading potentially to some interaction with an operator that determines the right space. The prevalence of planning and execution systems suggests this assumption is a reasonable one, although it does suggest that our methods and results will not apply to non-decomposable tasks.

Assumption 5: Engineering effort should be minimized. An important evaluation criterion in our work will be to ask how an approach or solution impacts the knowledge design or engineering effort associated with building an agent for a particular task. Solutions that minimize effort with acceptable task performance will be preferred over solutions that require more engineering effort. Because engineering effort is difficult to measure precisely, our experimental methodology will rely on relative measures of engineering effort rather than absolute ones. For instance, we will measure the number of rules required for particular tasks and compare this measure of effort to those of other approaches for the same task. However, we will not compare the number of rules for one task versus another because rules may not be a good measure of effort across different tasks.

Assumption 6: Development cost can be amortized over many applications. An architectural solution presents a dilemma because it is generally easier to develop a knowledge-based solution for a specific task than it is to develop a general solution applicable to all tasks. Thus, this approach appears to be in conflict with **Assumption 5**. However, we assume that there will be many agents developed using the architectural solution. Thus, the effort necessary for developing a general solution can be amortized over the development of all the agents using the architectural solution.

Assumption 7: The world has regularities that make learning useful. We assume that the world has goal-relevant regularities (Laird et al., 1996). A block stacking robot regularly encounters situations in which it must stack blocks. The can-collecting robot regularly encounters situations in which it must move, search, and grasp cans. Regularities in the task domain provide the impetus to learn from experience because it is reasonable to assume similar situations will be encountered again.

References

Agre, P. E. and Chapman, D. (1987). Pengi: An implementation of a theory of activity. In *Proceedings of the National Conference on Artificial Intelligence*, pages 196–201, Seattle, Washington.

Agre, P. E. and Horswill, I. (1997). Lifeworld analysis. *Journal of Artificial Intelligence Research*, 6:111–145.

Almasi, G. S. and Gottlieb, A. (1989). *Highly Parallel Computing.* Benjamin/Cummings, Redwood City, California.

Anderson, J. R. (1987). Skill acquisition: Compilation of weak-method problem solutions. *Psychological Review*, 94(2):192–210.

Bonasso, R. P., Firby, R. J., Gat, E., Kortenkamp, D., Miller, D. P., and Slack, M. G. (1997). Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental and Theoretical Artificial Intelligence*, 9(1).

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

Brooks, F. P. (1995). *The Mythical Man-Month; Essays on Software Engineering, Anniversary Edition.* Addison Wesley.

Brooks, R. A. (1986). A robust layered control system for a mobile robot. *IEEE Journal on Robotics and Automation*, RA-2(1):14–22.

Byrne, M. D. and Bovair, S. (1997). A working memory model of a common procedural error. *Cognitive Science*, 21:31–61.

Carbonell, J. G., Knoblock, C. A., and Minton, S. (1991). PRODIGY: An integrated architecture for planning and learning. In VanLehn, K., editor, *Architectures for Intelligence*, chapter 9, pages 241–278. Lawrence Erlbaum Associates.

Chapman, D. (1987). Planning for conjunctive goals. *Artificial Intelligence*, 32(3):333–377.

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*, pages 288–292, Evanston, Illinois.

Cox, B. J. (1986). *Object-Oriented Programming: An Evolutionary Approach.* Addison-Wesley.

Dechter, R. (1990). Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition. *Artificial Intelligence*, 41:273–312.

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*, pages 1153–1159, Montreal, Canada.

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

Doorenbos, R. B. (1993). Matching 100,000 learned rules. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*, pages 290–296.

Doorenbos, R. B. (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)*, Seattle, WA.

Doyle, J. (1979). A truth maintenance system. *Artificial Intelligence*, 12:231–272.

Durfee, E. H. (1998). Personal Communication.

Elsaesser, C. and Slack, M. G. (1994). Integrating deliberative planning in a robot architecture. In *Proceedings of the AIAA/NASA Conference on Intelligent Robots in Field, Factory, Service, and Space (CIRFFSS '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*, pages 1123–1128, Seattle, Washington.

Fikes, R. E., Hart, P. E., and Nilsson, N. J. (1972). Learning and executing generalized robot plans. *Artificial Intelligence*, 3:251–288.

Firby, R. J. (1987). An investigation into reactive planning in complex domains. In *Proceedings of the National Conference on Artificial Intelligence*, pages 202–206, Seattle, Washington.

Forbus, K. D. and deKleer, J. (1993). *Building Problem Solvers.* MIT Press, Cambridge, MA.

Forgy, C. L. (1979). *On the Efficient Implementation of Production Systems.* PhD thesis, Computer Science Department, Carnegie-Mellon University.

Forgy, C. L. (1981). OPS5 reference manual. Technical Report CMU-CS-81-135, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania.

Gaschnig, J. (1979). Performance measurement and analysis of certain search algorithms. Technical Report CMU-CS-79-124, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania.

Gat, E. (1991). Integrating planning and reacting in a heterogeneous asynchronous architecture for mobile robots. *SIGART BULLETIN 2*, pages 71–74.

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

Goel, A. K. (1991). Knowledge compilation: A symposium. *IEEE Expert*, 6(2):71–93.

Gupta, A. (1986). *Parallelism in Production Systems.* PhD thesis, Carnegie-Mellon University. (Also published as Technical Report CMU-CS-86-122, Computer Science Department, Carnegie Mellon University.).

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.

Hayes-Roth, B. (1990). An architecture for adaptive intelligent systems. In *Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 422–432, San Diego, CA.

Hayes-Roth, B., Pfleger, K., Lalanda, P., Morignot, P., and Balabanovic, M. (1995). A domain-specific software architecture for adaptive intelligent systems. *IEEE Transactions on Software Engineering*, 21(4):288–301.

Hennessey, J. L. and Patterson, D. A. (1990). *Computer Architecture: A Quantitative Approach.* Morgan Kaufman.

Kim, J. and Rosenbloom, P. S. (1995). Transformational analyses of learning in Soar. Technical Report ISI/RR-95-4221, Information Sciences Institute, Marina del Rey, CA 90292.

Knoblock, C. A. (1991). Search reduction in hierarchical problem solving. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 686–691, Anaheim, California.

Korf, R. E. (1987). Planning as search: A quantitative approach. *Artificial Intelligence*, 33:65–88. (Also published in *Readings in Planning*, James Allen and James Hendler and Austin Tate, editors, pages 566-578, Morgan Kaufmann, 1990.).

Kuhn, H. T. and Padua, D. A., editors (1981). *Tutorial on Parallel Processing.* IEEE.

Kuokka, D. R. (1991). MAX: A meta-reasoning architecture for X. *SIGART BULLETIN 2*, pages 93–97.

Laird, J. E. (1998). Personal Communication.

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

Laird, J. E., Pearson, D. J., Jones, R. M., and Wray, R. E. (1996). Dynamic knowledge integration during plan execution. In *Papers from the 1996 AAAI Fall Symposium on Plan Execution: Problems and Issues*, pages 92–98, Cambridge, Massachusetts. AAAI.

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*, pages 1022–1029, Boston, Massachusetts.

Laird, J. E. and Rosenbloom, P. S. (1995). The evolution of the Soar cognitive architecture. In Steier, D. and Mitchell, T., editors, *Mind Matters: Contributions to Cognitive and Computer Science in Honor of Allen Newell.* Lawrence Erlbaum Associates, Hillsdale, NJ.

Laird, J. E., Rosenbloom, P. S., and Newell, A. (1986a). Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning*, 1(1):11–46.

Laird, J. E., Rosenbloom, P. S., and Newell, A. (1986b). Overgeneralization during knowledge compilation in Soar. In Dietterich, T. G., editor, *Proceedings of the Workshop on Knowledge Compilation.*

Lallement, Y. and John, B. E. (1998). Cognitive architecture and modeling idiom: an examination of three models of the Wickens's task. In *Twentieth Annual Conference of the Cognitive Science Society*, Madison, Wisconsin.

Lee, J., Huber, M. L., Durfee, E., and Kenny, P. G. (1994). UM-PRS: An implementation of the Procedural Reasoning System for multirobot applications. In *Proceedings of the AIAA/NASA Conference on Intelligent Robots in Field, Factory, Service, and Space (CIRFFSS '94).*

Lehman, J. F., Dyke, J. V., and Rubinoff, R. (1995). Natural language processing for IFORs: Comprehension and generation in the air combat domain. In *Proceedings of the Fifth Conference on Computer Generated Forces and Behavioral Representation*, Orlando, Florida. Institute for Simulation and Training.

Martin, J. (1993). *Principles of Object-Oriented Analysis and Design.* Prentice-Hall.

McCarthy, J. and Hayes, P. J. (1969). Some philosophical problems from the standpoint of artificial intelligence. In Meltzer, B. and Mitchie, D., editors, *Machine Intelligence 4.* Edinburgh University Press.

McDermott, D. (1991). A general framework for reason maintenance. *Artificial Intelligence*, 50:289–329.

McDermott, J. and Forgy, C. (1978). Production system conflict resolution strategies. In Waterman, D. A. and Hayes-Roth, F., editors, *Pattern-directed Inference Systems*, pages 177–199. Academic Press, New York.

McGregor, J. D. and Sykes, D. A. (1992). *Object-Oriented Software Development: Engineering Software for Reuse.* Van Nostrand Reinhold.

Minton, S. (1988). *Learning Effective Search Control Knowledge: An Explanation-Based Approach.* Kluwer Academic Publishers.

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

Mitchell, T. M., Allen, J., Chalasani, P., Cheng, J., Etzioni, O., Ringuette, M., and Schlimmer, J. C. (1991). Theo: A framework for self-improving systems. In VanLehn, K., editor, *Architectures for Intelligence*, chapter 12, pages 323–355. Lawrence Erlbaum Associates.

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

Newell, A. (1980). Reasoning, problem solving, and decision processes: The problem space as a fundamental category. In Nickerson, R., editor, *Attention and Performance VIII*. Erlbaum.

Newell, A. (1990). *Unified Theories of Cognition*. Harvard University Press.

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.

Pell, B., Gat, E., Keesing, R., Muscettola, N., and Smith, B. (1996). Plan execution for autonomous spacecraft. In *Papers from the 1996 AAAI Fall Symposium on Plan Execution: Problems and Issues*, pages 109–116, Cambridge, Massachusetts. AAAI.

Prieto-Diaz, R. and Arango, G., editors (1991). *Domain Analysis and Software Systems Modeling*. IEEE Computer Society Press, Los Alamitos, CA.

Pryor, L., editor (1996). *Plan Execution: Problems and Issues: Papers from the 1996 AAAI Fall Symposium*. AAAI Press, Menlo Park, CA. Technical Report FS-96-01.

Rieman, J., Young, R. M., and Howes, A. (1996). A dual-space model of iteratively deepening exploratory learning. *International Journal of Human-Computer Studies*, 44:743–775.

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

Rosenbloom, P. S., Laird, J. E., and Newell, A. (1988). The chunking of skill and knowledge. In Bouma, H. and Elsendorn, A., editors, *Working Models of Human Perception*. Academic Press, London, England.

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

Schoppers, M. J. (1986). Universal plans for reactive robots in unpredictable environments. In Georgeff, M. P. and Lansky, A. L., editors, *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*. Morgan Kaufmann.

Schwamb, K., editor (1998). *Soar Workshop 18*. Explore Reasoning Systems, Vienna, Virginia.

Simon, H. A. (1969). *The Sciences of the Artificial*. MIT Press, Cambridge, MA.

Stallman, R. M. and Sussman, G. J. (1977). Forward reasoning and dependency-directed backtracking in a system for computer aided circuit analysis. *Artificial Intelligence*, 9(2):135–196.

Tambe, M. (1991). *Eliminating Combinatorics from Production Match*. PhD thesis, Carnegie-Mellon University. (Also published as Technical Report CMU-CS-91-150, Computer Science Department, Carnegie Mellon University.).

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.

Tambe, M., Newell, A., and Rosenbloom, P. S. (1990). The problem of expensive chunks and its solution by restricting expressiveness. *Machine Learning*, 5:299–348.

Tate, A. (1976). NONLIN: A hierarchical non-linear planner. Technical report, Department of Artificial Intelligence, University of Edinburgh.

Tversky, A. and Kahneman, D. (1982). Judgement under uncertainty: Heuristics and biases. In Kahneman, D., Slovic, P., and Tversky, A., editors, *Judgement under Uncertainty: Heuristics and Biases*. Cambridge University Press.

Wilkins, D. E., Myers, K. L., Lowrance, J. D., and Wesley, L. P. (1995). Planning and reacting in uncertain and dynamic environments. *Journal of Experimental and Theoretical Artificial Intelligence*, 7.

Wray, R. and Laird, J. (1998). Maintaining consistency in hierarchical reasoning. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence.*

Wray, R., Laird, J., and Jones, R. M. (1996). Compilation of non-contemporaneous constraints. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 771–778, Portland, OR.

Yu, S., Slack, M. G., and Miller, D. (1994). A streamlined software environment for situated skills. In *Proceedings of the AIAA/NASA Conference on Intelligent Robots in Field, Factory, Service, and Space (CIRFFSS '94).*