Validating Complex Agent Behavior

by

Scott A. Wallace

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
2003

Doctoral Committee:

Professor John E. Laird, Chair
Associate Professor William P. Birmingham
Assistant Professor Thad A. Polk
Professor Martha E. Pollack

Abstract

Validating Complex Agent Behavior

by

Scott A. Wallace

Chair: John E. Laird

Developing software agents that replicate human behavior, even within a narrow domain, is a time consuming and error prone process. The most widely used methodology for designing these agents is based on the complementary processes of knowledge acquisition and validation, both of which have been cited as significant bottlenecks. In this thesis, we identify two methods for comparing actors' behavior that have the potential to decrease the cost of validation.

The first is a simple sequence-based approach that can be used to compare many different aspects of two actors' behavior. Although initially promising, our empirical and analytical analysis exposes significant limitations with this general class of approaches, especially as the complexity of the domain increases. As a result, we turn to a novel comparison approach that we call behavior bounding. Unlike the sequential approaches, behavior bounding uses a concise representation of an actor's aggregate behavior as a basis for performing its comparison. We show that behavior bounding requires minimal human effort to use and that its representation of behavior is efficient to construct and maintain even as the complexity of the environment increases. Furthermore, we show that behavior bounding outperforms the sequential comparison approach in two domains of distinct complexity. Finally, we provide empirical evidence that behavior bounding's summary of the differences in two actors' behavior can be used to significantly speed up the knowledge validation process.

© Scott A. Wallace / All Rights Reserved 2003

For Jen, who makes even the dreariest of times brighter.

Acknowledgements

I would like to begin by thanking my adviser, John Laird, for all his efforts helping me along the sometimes arduous path of graduate school. His support has been strong from the beginning, and I have learned a great deal about both research and life itself because of his openness. I would also like to thank the other members of the Soar research group at the University of Michigan, including: Mazin As-Sanie, Karen Coulter, Mike van Lent, Bob Wray, John Hawkins, Brian Magerko, Tolga Konik, Mike James and Andy Nuxoll. The members of our group have always been available to discuss new ideas, and they have been willing to offer both constructive feedback and support through the many phases of my work at U of M.

In addition, I would like to extend my thanks to the members of my thesis committee: Bill Birmingham, Thad Polk and Martha Pollack. They have provided insights during the course of this project that have helped to make the work stronger and they have suggested a number of possible avenues for future work. Working with Professor Polk early in my graduate career provided a valuable insight into how research was performed outside of the Soar group, and I am very thankful of the time and effort he spent with me during that period. Professor Birmingham's encouragement and frank advice during the last few years of my graduate experience has also been very helpful as I try to plan and navigate the years ahead.

Without the strong group of friends I have here at Michigan, my experience would have been vastly different and much more difficult. My friends and officemates, especially Andy, Jeff, Chris, Mike, Lukasz, Brad and Uluç, have shared most of the high and low points in my graduate career. They have provided endless amounts of laughter, support and interesting conversation. Outside of the lab, my best friend Jen, as well as Branden, Laura, Tam and Boone have helped ensure that I don't sit in front of the computer all day and that I save time to do the other things in life that I love.

Finally, I thank my family, especially my wife Jen and my brother Mike, as well as my parents Roger and Linda. Together, you have provided much of the encouragement and cheer as well as the patience and support that have helped make me into the person I am today. I love you all very much.

Table of Contents

Dedication . . . ii

Acknowledgements . . . iii

List of Tables . . . viii

List of Figures . . . ix

List of Appendices . . . xi

Chapters

1 Introduction . . . 1
- 1.1 Verification & Validation . . . 2
- 1.2 Automatic Detection of Behavioral Errors . . . 3
- 1.3 Comparing Behavior: A Fundamental Problem . . . 5
- 1.4 Thesis Overview . . . 6
- 1.5 Research Contributions . . . 7

2 Toward Correct Human-Level Behavior . . . 9
- 2.1 Human-Level Agents . . . 9
- 2.2 Obtaining Correct Human-Level Behavior . . . 10
  - 2.2.1 Automated Learning . . . 12
  - 2.2.2 Automatic Refinement . . . 13
  - 2.2.3 Manual Knowledge Acquisition . . . 14
  - 2.2.4 Summary . . . 14

3 Related Work . . . 16
- 3.1 Properties of an Ideal Error Detection System . . . 16
- 3.2 Knowledge Based V & V . . . 19
  - 3.2.1 Validation Theory . . . 20
  - 3.2.2 Validation Tools . . . 21
  - 3.2.3 Summary . . . 24
- 3.3 Model-Based Diagnosis . . . 24
- 3.4 Knowledge Base Refinement . . . 26
  - 3.4.1 Traditional Refinement Systems . . . 26
  - 3.4.2 CLIPS-R . . . 27

3.4.3 IMPROV & EXPO . . . 29
3.4.4 Summary . . . 30
3.5 Discussion . . . 30

4 Behavior . . . 32
4.1 The Chain of Reasoning . . . 34
4.1.1 Complete Chains of Reasoning . . . 35
4.1.2 Complete Symbol Chains . . . 37
4.1.3 Chains of Goals and Actions . . . 37
4.1.4 Chains of Actions . . . 39
4.1.5 Appropriate Abstractions for Human-Level Agents . . . 40
4.2 Behavior Traces . . . 40
4.3 Correctness . . . 42
4.4 Errors . . . 48
4.5 Behavior Comparison & Error Detection Methods . . . 51
4.6 Evaluating an Error Detection Method . . . 53

5 Sequential Approach to Comparing Behavior . . . 61
5.1 A Basic Comparison Method . . . 61
5.1.1 Acquire . . . 61
5.1.2 Extract . . . 62
5.1.3 Compare . . . 66
5.1.4 Report . . . 68
5.2 Analysis . . . 69
5.2.1 Properties of the Behavior Space . . . 69
5.2.2 Sample Complexity . . . 71
5.3 Empirical Evaluation . . . 73
5.3.1 The Object Retrieval Environment . . . 74
5.3.2 The MOUT Environment . . . 81
5.4 Satisfying Properties of an Ideal Approach . . . 84

6 Behavior Bounding . . . 86
6.1 The Hierarchical Model . . . 87
6.1.1 The Create-Hierarchy Algorithm . . . 89
6.1.2 Leveraging Assumptions . . . 91
6.2 Identifying Errors . . . 92
6.3 Analysis . . . 94
6.3.1 Learnability . . . 95
6.3.2 Efficacy of the Lower Boundary Node . . . 96
6.3.3 Expressivity Limitations . . . 97
6.3.4 Comparison to other Model-Based Approaches . . . 99
6.4 Empirical Evaluation . . . 101
6.4.1 The Object Retrieval Environment . . . 101
6.4.2 The MOUT Environment . . . 103
6.5 Efficacy as a Validation Tool . . . 106
6.6 Extensions to Behavior Bounding . . . 112
6.6.1 Manual Definition of Lower Boundary Node . . . 112

6.6.2 Sometimes/Always Constraints 113
6.7 Satisfying Properties of an Ideal Approach 113

7 Conclusion 116
7.1 Contributions 118
7.2 Future Work 119

Appendices 121

Bibliography 133

List of Tables

Table

2.1 Important properties for determining a suitable general approach to obtaining correct human level behavior 11
2.2 Comparison of general approaches for obtaining correct agent behavior . . 14
3.1 Properties of an ideal approach to error detection 17
3.2 Properties of v & v approaches 24
3.3 Properties of model-based diagnosis 26
3.4 Properties of KR approaches 29
3.5 Summary of the properties in prior work 30
5.1 Individual experiments in family one 75
5.2 Properties of the sequential approach 84
6.1 Properties of expert & novice agents in the validation efficacy test 107
6.2 Assignment of validation methods and novice agents 108
6.3 Properties of behavior bounding 114

List of Figures

Figure

4.1 A chain of reasoning resulting in action A. 33
4.2 A chain of reasoning containing a sequence of symbol modifications. . . . 37
4.3 Capturing a behavior trace using a simulator 40
4.4 A space of possible behaviors 42
4.5 An incorrectly identified boundary between correct and incorrect behavior 45
4.6 Confusion matrix 46
4.7 Mismatches in the behavior trace indicate errors 48
4.8 Multiple simple errors may form a compound error 49
4.9 A cascade of errors follows an incorrect goal selection 50
4.10 The framework of a behavior comparison or error detection method 51
4.11 An overview of the steps in our evaluation process 54
4.12 Interpreting summary information from an error detection method. 57
5.1 A behavior trace that may profit from feature removal 63
5.2 Effects of different sampling methods 65
5.3 Mapping between complete and generalized behavior spaces 69
5.4 Effects of two extraction methods 71
5.5 Sequential approach: sensitivity in the object retrieval domain 77
5.6 Sequential approach: specificity in the object retrieval domain 79
5.7 Sequential approach: report density in the object retrieval domain 80
5.8 Sequential approach: sensitivity in the MOUT domain 82
5.9 Sequential approach: report density in the MOUT domain 83
5.10 Sequential approach: false positives in the MOUT domain 83
6.1 Hierarchical behavior representation & goal stack 88
6.2 Constructing the hierarchical behavior representation from a behavior trace 88
6.3 The Create-Hierarchy algorithm 90
6.4 Imposing order on the behavior space 93
6.5 Duplications cannot be directly identified by the HBR 98
6.6 Behavior bounding: sensitivity in the object retrieval domain 101
6.7 Limitations of behavior bounding's HBR in experiment family seven . . . 102
6.8 Behavior bounding: report density in the object retrieval domain 103
6.9 Behavior bounding: sensitivity in the MOUT domain 104
6.10 Behavior bounding: report density in the MOUT domain 105

6.11 Behavior bounding: report density in MOUT when ignoring floating operators . . . 106
6.12 Total time required by each participant to identify and correct the agent error . . . 109
6.13 Time required by each participant to identify the agent error . . . 110
A.1 The CREATE-HIERARCHY algorithm redeux . . . 123

List of Appendices

Appendix

A The CREATE-HIERARCHY Algorithm 122
B Validation Efficacy Pre-Experiment Handout 126
C Notes and Details from Analytic Results 130

Chapter 1

Introduction

As intelligence becomes an increasingly important attribute of software, we will begin to expect and value sophisticated agents that act faithfully on our behalf, or entertain us through interaction. This achievement has long been pursued by the artificial intelligence (AI) community–especially by those interested in producing agents with human characteristics. A number of intelligent systems capable of replacing a human counterpart have been demonstrated within restricted domains such as medical diagnosis [49] or simulated air combat [20]. However, we have yet to see these types of human-level agents make their way into mainstream software, or to perform adeptly in a wide range of situations with a diverse set of skills. Much of this impasse may be attributable to the fact that developing these agents is often extremely time consuming and expensive.

Developing a complex agent with human-like abilities typically begins with the process of knowledge acquisition. During this phase, human domain experts are interviewed to determine underlying rules that guide their behavior. A knowledge engineer uses this information to encode the agent's knowledge in a form that is usable by the underlying agent architecture. This traditional approach of knowledge acquisition is highly prone to errors in part because the human participants are stretched beyond their areas of expertise. For the domain expert, this means communicating knowledge about how tasks should be performed instead of simply performing them. For the knowledge engineer this means understanding the problem space well enough to determine how to translate the expert's descriptions into instructions that can be interpreted by the computer and that can be applied to appropriate situations. Although alternative methods of knowledge acquisition have been proposed and tested within a limited setting (e.g., [54]), for the most part, they have not been incorporated into widespread use. The result of this is

that developing complex intelligent agents remains a time consuming and error prone process.

The cost of introducing errors into mainstream software should not be underestimated; it may account, in part, for why human-level agents are not more prevalent. According to a recent study by the Research Triangle Institute and funded by NIST that assessed the costs of added labor, lost transactions and processing delays caused by software errors within the automotive, aerospace and financial service industries, bugs have a cost of nearly $60 billion on the U.S. economy [18]. Moreover, the study also found that between 30% and 90% of software development labor costs can be attributed to finding and correcting these errors. Given these figures and the widely held viewpoint that human-like agents are difficult to build and debug [52, 61], perhaps it is not surprising that these agents are not more common. In this thesis, we will show how automated comparison tools can be used to find errors in complex agents and thus reduce the cost of producing these agents.

1.1 Verification & Validation

Within the software engineering community, the process of finding errors in software is known as verification and validation. Although these terms have relatively distinct meanings in that field, within the context of intelligent systems literature these definitions have become somewhat confused. The result of this confusion has been inconsistent use, and a host of new definitions (see for example [28, 38, 39]). Even after the IEEE [17] issued a glossary defining verification and validation, these problems were not resolved. Recently, Gonzalez and Barr surveyed the use of these terms within the intelligent systems community and proposed definitions that clearly distinguish between verification and validation and maintain consistency with many trends observed in past definitions [11]:

Verification: the process of ensuring that the intelligent system 1) conforms to specifications, and 2) its knowledge base is consistent and complete within itself.

Validation: the process of ensuring that the output of the intelligent system is equivalent to those of human experts when given the same inputs.

According to this definition, verification tests conformance to a specification as well as consistency and completeness. Traditionally, specifications have taken the form of written documents that indicate well defined and measurable properties of the product such as the signal to noise ratio on an amplifier, the number of pixels on a CCD, or the weight of a laptop. When dealing with intelligent systems, many attributes that the user would desire from an agent are not, in fact, well defined or easily measured. As a result, these formal specifications have often been of secondary importance [11]. The other important aspect of verification is to ensure that the knowledge base (KB) is consistent and complete within itself. Note that this process is typically possible without actually examining the agent's behavior. Moreover, many problems that would lead to inconsistencies or incompleteness can be identified using relatively simple syntactic checking. Not surprisingly, a significant amount of research has focused on developing automated methods to perform this aspect of verification (e.g., [9, 37, 43, 47, 51]).

In contrast to verification, validation deals directly with the problem of establishing whether the agent performs its functions in the same manner as the human expert. Additionally, it is clear that the agent's behavior must be examined *as it is engaged in the process of problem solving* to make this determination. Because validation requires observing the agent and making judgments about whether its behavior is suitable, a typical validation approach involves significant oversight from the human expert and potentially from the knowledge engineer as well. As a result, validation can quickly become the source of great expense. To reduce this cost, an automated approach would be highly desirable.

1.2 Automatic Detection of Behavioral Errors

One could imagine that automated error detection would be possible by simply comparing the agent's behavior to the domain expert's behavior on a set of tasks and checking for deviations. Unfortunately, identifying errors without direct help from a human is not a straightforward task because not every deviation in behavior is an error. In fact, the concept of an error is often ambiguous and closely tied to the properties of the underlying task that is being performed. The complexities surrounding this problem can be illustrated best with a concrete example.

Consider an airborne defense mission in which the pilot flies combat air patrol in-

tercepting enemy planes as they are identified. Initially, the expert pilot and its agent counterpart fly identically. Once an enemy is identified, both the expert and the agent decide to use the same tactic to engage the enemy. While executing their maneuvers, however, there are significant differences between the speed and altitude of the expert and the agent, even though they both succeed in shooting down the enemy. Finally, on the return to base, there is once again a deviation between the expert's and the agent's speed and altitude even though their actions are otherwise identical.

The problem is to determine whether the difference between the agent's behavior and the expert's behavior indicates an error, or whether the agent's actions are within the scope of correct behavior. It is clear that this distinction is intimately tied to properties of the task being performed. Nonetheless, we could imagine that while engaging the enemy plane the range of acceptable behavior is relatively broad so that the minor differences between the expert and the agent are inconsequential and do not represent an error. On the other hand, we could also imagine that a plane's "return to base" profile (i.e., its speed, altitude, and perhaps its heading) as it approached base would be used by the base's air control operators to discriminate between friendly and hostile aircraft when automated identification (IFF) fails. In this scenario, any differences between the expert plane's profile and the novice plane's profile would constitute an error and therefore would be crucial to identify and correct.

In some cases, examining the external behavior of the expert and agent may not be sufficient to identify all of the salient differences in their behavior. Consider once again the military pilot engaging an enemy in combat. Although their low level actions (`set-speed-`x and `set-altitude-`x) differ, their higher-level goals (e.g., `destroy-enemy` and `perform-maneuver-`y) are identical. We could imagine a reverse situation, however, in which the only difference between the agent and the expert's behavior were at the level of goals, not actions. During landing, for example, the expert pilot may pull up on the stick just prior to touch down in order to help achieve the `land-softly` goal. A novice agent in the same position might also pull back on the stick just prior to touch down because of an overwhelming fear of approaching the ground. Although their external actions are identical, the motivation behind their behavior is nonetheless divergent. In this particular case, it is likely that this divergence would indicate an error that should be corrected. Moreover, this deviation could not have been identified by comparing the

expert's and the agent's actions alone.

The examples above illustrate how difficult it may be to correctly discriminate between appropriate and inappropriate behavior. Indeed, it may not be possible to develop a single error detection system that can operate autonomously across a large set of complex domains without relying on a large body of domain specific knowledge. In fact, most prior work on automatic error detection relies on a number of assumptions to simplify the problem of determining what is and what is not an error. These assumptions may limit the nature of the tasks being examined, for example by being applicable only to non-interactive classification type problems. Alternatively, they may put limits on the types of errors that can be identified, for example by restricting the notion of an error to only those conditions that prevent the agent and the expert from reaching the same final state. Both of these assumptions are often too restrictive for many interesting domains. Our research takes a step toward the goal of automated error detection by identifying the research challenges and outlining a set of methods that can be used to compare behavior and detect many types of errors in the agent's knowledge across different domains.

1.3 Comparing Behavior: A Fundamental Problem

Although clearly necessary for detecting errors, general methods for identifying similarities and differences in agent behavior are important for a number of other tasks. In fact, comparing behavior is a fundamental problem that has a bearing on:

- Scoring a strict Turing Test: a general approach to detecting behavior errors could be used to objectively measure differences between human behavior and software agent behavior. A perfect score would be indicated if no detectable differences were identified.

- Automatic correction of knowledge bases: a system which modifies an agent's knowledge to remove bugs must determine when the agent's knowledge needs to be changed and when changes have improved the agents behavior. A general approach to error detection could serve both of these purposes.

- Intelligent tutoring systems: fundamentally, the problem of identifying errors in human behavior is very similar to identifying errors in software agent behavior. So long as the error detection system relies on information that can be obtained from

either a human or a software agent, it should be possible to reverse their roles such that a novice human's behavior is compared against an expert agent's behavior to determine when an error has been made.

- Intelligent interfaces: some interfaces, or controls, are used for relatively narrow, focused tasks. In these situations it would be possible to provide a usage model that covers each of the interface's prescribed functions. By comparing the user's behavior to this gold standard, the interface may be able to help the user perform their task, or even recover from incorrect user behavior.

- Machine learning; systems that learn how to perform a task must be able to evaluate the knowledge they have created. In particular, systems that learn by observation stand to gain from generalized error detection approaches by increasing their ability to reflect on how their learned behavior deviates from observations of expert behavior.

Generating meaningful behavior comparison in a completely automatic way remains a challenge. However, progress in this direction, even without achieving full automation, has the potential for making contributions to each of these areas of research.

1.4 Thesis Overview

The primary goal of this research is to explore different methods of comparing behavior with an emphasis on comparing human and software agent behavior. In the previous section, we identified a number of potential applications for this research. Throughout the majority of this thesis, however, we will focus on applying behavior comparison techniques to the problem of validating an agent's knowledge base to maintain the presentation's continuity.

In Chapter 2, we begin by describing human-level agents—the types of intelligent systems we are most interested in supporting. Our desire to build these type of agents leads us to seek methods that can speed their development, one of which is automated validation. Before settling on this approach, however, we explore a number of methods that may help us build human-level agents more efficiently. Our analysis of the pros and cons of each approach leads us to the conclusion that helping to automate knowledge validation is an appropriate goal on which to focus our efforts.

In Chapter 3, we examine prior research that has addressed methods of identifying faults in a system's behavior. Prior research covers approaches that range from completely manual validation techniques to completely automated model-based techniques. We identify a trend in which the ability to support human-level agents decreases as the system's overall autonomy increases. In addition, we posit that a semi-automated approach to detecting behavioral errors will be best suited to reducing the cost of validating complex, human-level systems.

Next, in Chapter 4, we examine behavior at different levels of abstraction, identifying which levels are most appropriate for describing the behavior of both humans and software agents. Based on this representation of behavior, we discuss what it means for behavior to be correct or incorrect. We then explore the manner in which errors manifest in behavior, and we observe that some errors are can be more useful for speeding up the knowledge validation process than others.

Chapters 5 and 6 examine two approaches to behavior comparison. In the first of these chapters, we look at a simple methodology that uses individual observations of two actors' behavior to identify similarities and differences. Based on limitations of this approach, we propose and examine a second methodology, called behavior bounding, in the following chapter. This approach is based upon a model of the agent's aggregate behavior as opposed to a collection of individual instances. In both chapters, we examine analytical properties of the respective approaches and evaluate them empirically in two domains. In addition, in Chapter 6, we provide experimental results that indicate how much the approach can speed up the validation process.

Finally, in Chapter 7, we conclude the thesis by summarizing the results of our study, highlighting the contributions we have made and laying the course for future work.

1.5 Research Contributions

The primary goal of this research is to explore different methods of detecting errors in an agent's behavior. To perform this comparison, we have examined a number of behavior comparison methods at both an empirical and theoretical level. The main contributions of our work are to:

- Identify the relationship between knowledge base validation and machine learning.

- Introduce a method for determining relative salience among different behavior differences.
- Introduce a metric for determining the quality of information in an error detection method's output.
- Describe Behavior Bounding, a novel method to identify differences in human and agent behavior.
- Provide an empirical analysis of Sequence Based Error Detection and Behavior Bounding.
- Identify the limitations of Sequence Based approaches as a general class of comparison metrics.

Chapter 2

Toward Correct Human-Level Behavior

2.1 Human-Level Agents

Intelligent systems have been used in a wide variety of domains, from speech recognition [45] to game playing [25] to simulated aircraft control [20]. Our research addresses a particular subset of these intelligent systems which we refer to as *human-level agents*. Human-level agents are distinguished by the following criteria: first, human experts, as opposed to computers typically perform the task. Second, to be successful, the intelligent system, or agent, must reproduce the expert's externally observable behavior. Additionally, it is often important that the agent reproduce the expert's internal reasoning to some degree (often to ensure that the agent's actions can be explained to other humans). Finally, like humans themselves, human-level agents must interact with an external environment in order to perform many of their tasks.

Like other types of intelligent systems, human-level agents can be viewed as occupying a spectrum of interactivity depending on how often they must sense and react to their environment. At one end of this spectrum are relatively deliberate agents and on the other end are relatively interactive agents. It is interesting to note that human-level agents are rarely purely interactive or purely deliberate. This is because complex tasks typically involve both of these types of behavior, each of which may be required at different points in the problem solving process.

A good example of a relatively interactive human level agent is TacAir-Soar [20]. TacAir-Soar flies virtual military planes as part of a simulated training exercise. Teammates may be other TacAir-Soar agents or human counterparts. Because the agents are intended to model expert-level behavior, it is not acceptable to simply achieve the same final states as the experts (e.g., shooting down enemy planes). Instead, the agents must

generate the same behavior as the experts. Since TacAir-Soar agents are used for both training and simulation, it is also important that their actions can be explained to other humans. By reproducing some aspects of the human's internal behavior, such as the way in which tasks are decomposed into sub-tasks and actions, the motivations behind the agent's behavior is easily communicated to other members in a training exercise or to the observers of a simulation.

In contrast to TacAir-Soar, a relatively deliberate human-level agent is one that performs a significant amount of internal reasoning before executing an externally observable action. This type of behavior can be found in agents that model mathematical problem solving, some types of game playing such as chess, and in certain aspects of more general behavior such as anticipating what another agent in the environment might do [24].

Our goal is to help developers construct human-level agents more efficiently. We are particularly interested in targeting relatively interactive human-level agents because these are the types of intelligent systems we build and work with most frequently. At the same time, however, we want to ensure that our work can also be applied to the largest class of agents possible. Thus we would also like to offer at least partial support for human-level agents that fall more on the deliberate side of the spectrum.

2.2 Obtaining Correct Human-Level Behavior

In Chapter 1, we stated that automating the validation process would be likely to make the process of creating correct human-level agents less costly and more efficient. However, before we commit to this method it behooves us to step back and momentarily examine other ways in which this goal could be achieved. In particular, learning approaches and automated knowledge refinement offer alternative means of generating correct agent behavior. In order to determine the best approach, we will examine each of these methods with respect to following criteria (see Table 2.1):

C1: Human effort As discussed in Chapter 1, the domain expert and knowledge engineer often play a crucial role in knowledge acquisition. The domain expert understands the requirements of the tasks, and is often the gold standard to which the human-level agent's behavior will be compared. The knowledge engineer, on the other hand, has the ability to encode these problem solving methods into a form

Criterion	Description
C1	Human effort
C2	Expected computational resources
C3	Supports human-level behavior
C4	Expected research effort

Table 2.1: Important properties for determining a suitable general approach to obtaining correct human level behavior

that can be executed by the underlying agent architecture. The disadvantages of relying on human effort are twofold: their labor is expensive, and they cannot work continuously. As a result, an ideal approach would obtain correct behavior while minimizing the human's involvement as much as possible.

C2: Expected computational resources To a certain extent, human effort can be reduced by using additional computational resources. For example, this might be done by requiring a computer to search for parameters to correct behavior as opposed to requiring a human to supply these parameters. Some tasks, however, are so complex that they cannot be solved efficiently in this manner. In this case, one of two possibilities results: either the approach becomes infeasible (i.e., its computational resources are impractical); or simplifications must be made, either using heuristic methods or by limiting the scope of the problems that can be addressed. An ideal approach would use relatively low amounts of computation thus ensuring its ability to scale to more complex behavior.

C3: Supports human-level behavior Prior work has shown that there are many ways to obtain the behavior for relatively simple tasks such as classification. However, few of these methods have been demonstrated on the complex, interactive tasks typical of human-level behavior and fewer have gained support outside of academic circles.

C4: Expected research effort If a methodology has not demonstrated the ability to support human-level behavior, bridging this gap is likely to require additional research. This property attempts to give a rough quantification on how many challenges must be faced by a particular approach before it is capable of meeting the needs of human-level tasks.

2.2.1 Automated Learning

Automated learning techniques hold the promise to reduce the cost of developing agents because some (or all) of the agent's knowledge can be acquired by the learning system itself. Approaches may experiment directly within the domain, or they may use positive and negative examples to determine what behavior is most appropriate within a given context. In this respect, learning approaches all attempt to trade computational resources for human effort.

A number of machine-learning approaches (e.g., [13, 44, 48, 60]) have proven useful in both academic and industrial settings. However, these techniques have typically excelled in learning relatively simple behavior to perform such activities as classification tasks. Two approaches that target rich human-level behavior in particular are inductive logic programming [26] (ILP) and van Lent's learning by observation technique [53]. Both approaches have been used to learn sets of rules that can describe an agent's behavior, however both approaches have limitations. ILP relies heavily on large numbers of example data and heuristics for determining how rules should be built. Although this may yield appropriate observable behavior, the underlying motivations for that behavior may be incorrectly represented. Learning by observation has been shown to work in some complex domains, but it has difficulties learning to encode some deliberative tasks even if they are relatively simple (e.g., Towers of Hanoi). Furthermore, neither approach has been incorporated into widespread use.

In general, learning suffers from three significant problems. The first of these is that the agent's problem solving methodology is constructed with little or no human oversight. Any given behavior can be generated in a number of different ways. As we indicated earlier, even if the learning system is able to construct knowledge that will allow the agent to reproduce human behavior, it may do so in ways that are undesirable. For example, it may make rules or data structures that are difficult to read by humans, leading to difficulties with future development. The second potential drawback with a learning approach is that as the complexity of the domain increases, learning becomes significantly more computationally intensive and learning the complex rule sets required by human-level agents has not yet proven to be practical. In addition, when salient features such as internal knowledge structures are hidden from the learning algorithm, appropriate generalization can be difficult, if not impossible. The third difficulty arises

after the learning process has begun. The learning system must evaluate whether the knowledge it has generated will produce the appropriate agent behavior. This, however, is the basic problem of automated knowledge validation, which we know is an open ended research question. Clearly, there are a number of obstacles to a pure learning approach, and thus the expected research effort is high. Although this research effort may be reduced by combining learning with additional domain knowledge, such a change would also increase the human workload.

2.2.2 Automatic Refinement

Another method of generating correct agent behavior is to construct an initial knowledge base by hand and use an automated tool to remove errors. This process is known as knowledge base refinement. In contrast to a learning approach, this method requires more effort from the programmer (to initially encode the agent's knowledge), but has the potential of requiring less computational resources.

Interestingly enough, most refinement systems are still limited by their computational requirements. Typical refinement systems such as KRUST [5] and EITHER [40] use abduction to determine which elements of the agent's knowledge base (or domain theory) must be modified when an error has been identified. Abduction has been shown to be NP-Hard, even when some simplifying assumptions can be made. Not surprisingly, most of the successful demonstrations of this technique have been with relatively small knowledge bases containing few errors. One refinement system, IMPROV [41], avoids abductive reasoning altogether. Although this decreases the computational complexity of the refinement process, IMPROV only has the ability to detect a limited set of errors in the agent's reasoning and behavior.

Automated refinement overcomes one of the major drawbacks of the learning approach by allowing the designer to implement most of the agent's knowledge in the way that she sees most fit. However, past work indicates that computational requirements of automated refinement are still too high to be feasibly used with complex human-level agents. In addition, all refinement systems must be able to determine whether the changes they have made result in more appropriate behavior. An incorrect determination has the potential for introducing new errors, or of incorrectly constraining the agent's behavior. To be successful with human-level agents, a refinement system must overcome

Approach	C1: Human Effort	C2: Comp. Resources	C3: HL Behavior	C4: Research Effort
Learning	Low	High	No	High
Refinement	Mid/High	Mid/High	No	High
Manual	High	Low/None	Yes	None

Table 2.2: Comparison of general approaches for obtaining correct agent behavior

the computational bottleneck and, like learning approaches, the refinement systems must provide a reasonable method of solving the behavior comparison problem.

2.2.3 Manual Knowledge Acquisition

A third method to obtain correct agent behavior is a fully manual approach in which the agent's knowledge base was constructed by human programmers, and then validated and refined using manual techniques (e.g., [20, 59]). This approach requires very little computational power and can ensure that the agent's behavior is implemented in the most appropriate way.

Although the shortcomings of a purely manual approach to agent development are significant, it remains the standard in both industry and research environments [28]. Moreover, this is the only approach that has repeatedly proven itself to be adept for creating human-level agents that meet their operating requirements.

The most significant asset of the manual approach is also its greatest weakness: the reliance on human effort. Because humans assess the quality of the agent's behavior, the manual approach can leverage the tacit knowledge of the domain expert to determine whether the agent is behaving appropriately, circumventing one critical problem suffered by both learning and automatic refinement approaches. Furthermore, because the human implements and changes to the initial knowledge base, errors will be corrected in the most suitable way. However, this reliance on human labor increases the cost and time required to develop complex agents, thus creating a bottleneck of its own.

2.2.4 Summary

Learning and automated knowledge base refinement both seek to make the task of obtaining correct agent knowledge more efficient. However, by so drastically reducing

the human's role in this process they risk being able to detect flaws in behavior and, ultimately, they risk their ability to generate correct agent behavior. On the other hand, while a completely manual approach helps ensure that the agent's knowledge is represented in the most appropriate manner, it suffers from high human effort and high expense. Table 2.2 illustrates that none of these approaches is ideally suited for developing human-level agents because all require significant resources in at least one critical area.

Our research seeks to reduce the cost and time required to develop complex, human-level agents by automating portions of the validation process. In particular, we aim to provide a set of automated tools to help detect errors in an agent's behavior. These tools will help reduce human effort by minimizing the participation of the domain expert in diagnosing problems, and by focusing the knowledge-engineer's effort on specific potions of the knowledge base that are in need of corrections. Moreover, because the problem of detecting errors in agent's behavior is central to all three of the techniques described above, an automated tool may also provide the foundation for improved learning and refinement approaches. In this paper, however we focus on how these tools can be used to aid manual development as this currently the most widely used method of encoding human-level agents.

In the next chapter, we will examine prior research that addresses the central problem of detecting errors in behavior. Based on an analysis of this prior research, we will describe methods for detecting behavior errors that are specifically tailored to interactive human-level agents.

Chapter 3

Related Work

In Chapter 2, we examined potential methods for obtaining correct agent behavior. Our analysis indicated that all of these methods would benefit from research on new approaches for automatically detecting errors in agent behavior. In addition, our analysis also indicated that we should specifically target improving manual knowledge acquisition since this is most commonly used approach for developing the complex, human-level agents we are interested in supporting. In this chapter, we will examine properties of an ideal error detection system. Using these properties, we will evaluate how prior work has addressed the problem of detecting errors in an agent's behavior.

3.1 Properties of an Ideal Error Detection System

An ideal, fully automatic error detection system capable of supporting human-level agents may be an unobtainable goal. Most likely, trade-offs between competing desires will be necessary. Nonetheless, it is instructive to examine what capabilities would be most desirable in such a system. This analysis provides a framework for examining related areas of research as well as a basis for evaluating new systems. Table 3.1 lists the properties of an ideal error detection system. These properties are described in more detail below.

P1: Minimize human oversight As human laborers, the knowledge engineer and domain expert must be paid for their time and cannot work continuously for long periods of time. As a result, a development approach that reduces their involvement has the potential of being both faster and more cost effective. Human oversight is the portion of the development process in which the domain expert and knowledge

Property	Description
P1	Minimize human oversight
P2	Minimize supportive engineering
P3	Minimize reliance on example behavior
P4	Provide a grounded stopping criterion
P5	Adapt to diverse environments
P6	Leverage tacit expert knowledge
P7	Evaluate different aspects of reasoning
P8	Support imprecise specifications of correctness
P9	Identify errors in human or machine behavior

Table 3.1: Properties of an ideal approach to error detection

engineer must evaluate an agent's performance in a number of test scenarios. Because this process may happen relatively frequently (e.g., when changes are made to an agent's knowledge, or a new variation of an agent is created), an ideal error detection mechanism must minimize human oversight as much as possible.

P2: Minimize supportive engineering One way to reduce human oversight in the error detection process is to replace it with a hand crafted tool that is tailored for the particular environment. Because the tool may need to be created only once for each environment, this may be more efficient than relying on a large amount of direct human oversight. However, this approach is not without drawbacks. How can we be sure the new engineered tool is free of errors? Clearly, the answer is that we cannot, and ironically we have created a recursive validation problem. Because supportive engineering adds new layers of costs, an ideal error detection system should avoid creating such tasks whenever possible.

P3: Minimize reliance on example behavior Another way to reduce human oversight in the error detection process is to compare the agent's behavior to instances of captured expert behavior. The benefit of this approach is that expert behavior can be captured once for each task and used to detect errors in many different agents. On the other hand, this approach will suffer if an extremely large number of example behaviors is required to determine whether the agent's behavior is correct. As a result, an ideal error detection method would ensure that the required number of expert examples is minimized.

P4: Provide a grounded stopping criterion Whether error detection is performed fully automatically or via direct human oversight, it is often unclear when the process of detecting errors should be considered finished. Developers may default to using ad hoc methods, or simply continue to look for errors until they have expended their allotted resources. An ideal error detection method would provide three features to help developers identify when they can stop the validation phase. First, it would provide a means of predicting how long validation will take. Second it would inform the developer if it has reached a point in which it will no longer yield useful information. Finally, it would end the validation procedure if it could determine that the agent's behavior was correct within some prescribed tolerance.

P5: Adapt to new environments In some environments, the parameters of correct behavior are very strictly defined. In others, there may be significant freedom in how different tasks are solved. An error detection method must be easily adaptable to support environments with different specifications for what constitutes acceptable behavior.

P6: Leverage tacit expert knowledge One of the greatest assets of a manual approach to error detection is the close involvement of the domain expert. Because the domain expert examines the agent's behavior directly, she can judge whether or not the agent's behavior is correct without having previously externalized the exact parameters for correct behavior. This is extremely useful for diagnosing problems in human-level agents, because obtaining correct and complete specifications for their behavior is notoriously difficult.

P7: Evaluate different aspects of reasoning Errors may occur at many different points in the reasoning process. There are practical limitations to how effective a system can be at identifying errors in all of these locations as we will show in Chapter 4. However, an ideal error detection system should be able to detect errors at all feasible points in the reasoning process.

P8: Support imprecise specifications of correctness As mentioned previously, it is often difficult to get a complete and correct specification for the behavior of a human-level agent. Not only should an ideal error detection method accept imprecise specifications, but it should directly support them by tailoring its classification

scheme to the confidence level of different aspects of the specification.

P9: Identify errors in human or machine behavior An ideal error detection system could be used for more than knowledge base validation. It could also be used to identify errors in novice human behavior.

The properties of an ideal error detection system are outlined in Table 3.1. Note that properties **P1** through **P4** relate to the cost of using the methodology, with **P1** through **P3** describing various sources of human effort. Properties **P5** through **P8**, on the other hand, relate to an error detection methodology's ability to support human-level agents. An ideal error detection method would meet all of these properties we have outlined, and would be able to empirically demonstrate its efficacy.

In the next sections we will review prior research related to our general problem of identifying errors in an agent's behavior. The properties outlined above serve as a basis to analyze how the problem of detecting behavior errors has been addressed in the past, and what areas may benefit from further improvements. We begin our analysis by examining the area of knowledge based verification and validation and then proceed to examine model-based diagnosis and knowledge refinement in the following sections.

3.2 Knowledge Based V & V

Verification and validation have been identified as important parts of the expert system life cycle [21]. As we described in Section 1.1, verification is largely concerned about completeness and consistency. Errors of this type, while important, can often be found using syntactic methods. Behavioral errors that occur at the semantic level, on the other hand, often cannot be identified without inspecting the system's behavior while it is in action; these errors are addressed by validation research.

Within the knowledge base V & V community, validation research typically strives to increase the process's efficiency. This effort is crucial because it has been widely recognized that validation is often an difficult and expensive process [38,52]. This is especially true when the system being validated exhibits complex problem-solving capabilities [61], a quality that is typical of human-level agents. Error detection lies at the heart of the validation problem, and as a result it seems that the V & V community would be an obvious place to find information about automating this process. Validation research

can roughly be divided into two areas: work that addresses theoretical aspects of how the validation process should be conducted; and work that examines tools which help support this task.

3.2.1 Validation Theory

At the theoretical level, prior work has examined a range of validation techniques. These include white-box, or glass-box, validation, which aims to determine whether the system is correctly specified by examining its internal properties such as beliefs or goals, and black-box validation, which has similar aims, but examines only external, observable properties such as the agent's action [21, 52]. Additional methods such as Turing-style tests attempt to validate a system by determining whether its behavior is distinguishable from an expert's behavior [38]. Typically, theoretical studies of validation techniques do not address how the process could be automated. Rather, this work describes what we have previously been referring to as manual error detection.

Manual validation, or error detection, requires significant effort from both the domain expert and the knowledge engineer. This human effort is devoted almost completely to watching the agent's behavior on a large number of test scenarios (**P1**). Because the manual approach detects errors only when they are witnessed by the domain expert or the knowledge engineer, any time a test of the agent's behavior is conducted, one, or possibly both, of the humans must be involved. However, because humans are involved so intimately with detecting errors in the agent's behavior, there is no need to engineer domain dependent methods to identify errors (**P2**), nor is there any reason to collect examples of expert behavior (**P3**). However, cost is increased significantly by the fact that manual validation typically continues until resources are depleted or until some arbitrary number of tests have been successful (**P4**). This means that it is very likely that a fair amount of the human effort poured into observing the agent's behavior will be wasted, or that the agent may complete the validation cycle without a reasonable guarantee of actually being correct. Because manual validation is so costly to begin with, this situation is far from ideal.

At the same time, because manual error detection relies so heavily on active domain expert participation, it has a number of advantages. First, because the domain expert encapsulates knowledge about how the agent should behave, one can quickly adapt a val-

idation procedure to a new environment (**P5**) simply by using a different domain expert. Secondly, because the domain expert can identify errors simply by observing behavior in a test scenario, a manual approach can leverage tacit knowledge that may not have been uncovered during knowledge acquisition (**P6**). This means concepts that are difficult for the domain expert to articulate can still be used to detect errors and this makes it somewhat easier to deal with incomplete or imprecise specifications. In addition, the domain expert can asses how well different aspects of reasoning (such as actions or motivation) match expectations (**P7**). Furthermore, because the domain expert is actively participating in diagnosing behavior, this method is robust even when the parameters for correct behavior are ill-defined (**P8**). Finally, the manual validation techniques are used daily during training sessions and role playing to identify errors in human behavior so they can clearly be used to identify errors in either human or agent behavior (**P9**).

3.2.2 Validation Tools

The second body of work deals explicitly with systems that attempt to provide some form of aid during the validation process. Tools such as Teiresias [6] and KVAT [32] can help track down faults in knowledge bases, thus potentially reducing the effort required by both the domain expert and the knowledge engineer. Unfortunately, these systems do not emphasize automating the error detection process. Instead they focus on helping to automate the process of correcting inappropriate knowledge. These system are similar to a fully manual approach in that they rely on a human to recognize situations in which the agent has reached an invalid conclusion. Indeed, because of this basic reliance, they are identical to a manual approach with respect to properties **P1** through **P9**.

Other systems, such as EXPECT [9], COVADIS [47], and EXPECT-ETM [51] attempt to find faults in an agent's knowledge base before the agent is required to interact with the outside world. These three systems identify errors by examining the agent's internal knowledge directly as opposed to examining instances of behavior. This attribute, coupled with the fact that most of these types of systems identify domain independent errors, means that their main advantage lies in the ability to reduce human effort. Because these tools identify errors without actually examining the agent's behavior, there is no need for human oversight (**P1**) or example behavior (**P3**). In addition, if no knowledge needs to be added to help the system detect errors (as would be the case if only truly

domain independent problems were identified)[1], no supportive engineering is required (**P2**). Finally, because these tools work by examining the knowledge instead of instances of behavior, there is no need for a stopping criterion (**P4**).

Not surprisingly, however, there are also significant disadvantages to reducing human participation in this way and relying only on domain independent errors. First, although the ability to detect domain independent errors may, in fact, speed up the development process, many problems are likely to go unnoticed. Because the parameters of correct behavior will differ between environments, completely domain independent automatic error detection methods (i.e., those that do not require human oversight, supportive engineering, or example behavior) will never be able to adapt the varied circumstances necessary to support human-level agents (**P5**). Second, because the domain expert does not play any role in detecting errors, their tacit knowledge about appropriate behavior can never be exploited (**P6**). In fact, it is questionable how many behavioral errors could be detected without ever seeing the agent perform any tasks. In addition, because these approaches use fixed criteria for determining what constitutes an error, and because the domain expert does not participate in identifying incorrect behavior, they have little ability to deal with the imprecise or incomplete specifications of human-level agents (**P8**). Furthermore, because these systems all rely on processing the agent's knowledge, as opposed to observing or interacting with the agent, it would be difficult if not impossible to use these techniques with humans in place of agents (**P9**). However, because they examine the agent's knowledge base directly, these systems have the potential of identifying errors at any point in the reasoning process; not just those points that have some externally observable manifestation (**P7**).

One of the more interesting tools for aiding the validation process, is provided by the family of Dependency Detection (DD) algorithms described in [14, 15]. Dependency detection does not attempt to explicitly find or correct flaws in the agent's behavior. Its purpose is to provide an abstract representation of the agent's behavior that is easy to examine. In this way, DD strives to make the process of finding and correcting errors simpler than it would be if the agent's internal knowledge representation had to be examined directly.

[1]Some systems (e.g., EXPECT-ETM) are able to detect domain independent errors, but may also be expanded to detect other errors by adding supplemental knowledge. The trade-offs of this approach will be explored in more detail in Section 3.3.

Dependency Detection's abstract behavior representation is a probabilistic state transition diagram in which each state corresponds to a sub-sequence of the the agent's behavior. The diagram as a whole represents the relative probability of the agent performing a given task using different sequences of behavior. By analyzing the states and their transitions, the domain expert or knowledge engineer can identify potential flaws in the agent's knowledge.

Dependency Detection's approach to aiding the agent validation is distinct from the other tools we have examined. Unlike the manual approach and Teiresias Style approaches, DD minimizes human oversight (**P1**) because results of many simulated trials are condensed into the single abstract behavior representation. In addition DD minimizes supportive engineering (**P2**) because the abstract behavior representation is built automatically based on examining sequences of the agent's behavior. However, because it is unclear when DD's representation covers the agent's behavior appropriately, it is also unlikely to be clear how many examples of behavior it should be given (**P3**) as well as when it is appropriate to end the validation phase (**P4**).

Dependency Detection's use of an abstract model of the agent's behavior also makes it reasonably suitable for many complex, human-level tasks. Because the model makes no assumptions about what constitutes correct and incorrect behavior (all error detection is performed by the human expert) it is adaptable to many types of environments (**P5**). This requirement of keeping the domain expert in the loop for error detection also means that, like the manual approach to validation, DD may be able to leverage tacit expert knowledge (**P6**) when errors are detected as well as support imprecise specifications of correctness (**P8**)[2]. Although the work on DD does not explicitly mention that it can be used to identify flaws in different aspects of behavior (e.g. within goals or actions) one could imagine that this would be the possible so long as the system was able to identify what goals the agent was pursing at any given point during task performance (**P7**). Finally, although Howe does not address the idea directly, because Dependency Detection relies on examining sequences of behavior, as opposed to the agent's knowledge base, it should be possible to use this approach to evaluate human or machine behavior (**P9**).

[2]Note, however, that as Dependency Detection's behavior representation becomes increasingly complex, it may be difficult for the domain expert to comprehend the model and map it onto their own experiences. As the domain expert becomes further distanced from the validation process, it will become difficult to leverage their tacit knowledge.

	Cost				Human-Level Agent Support				
Method	P1	P2	P3	P4	P5	P6	P7	P8	P9
Manual	—	√	√	—	√	√	√	√	√
Teiresias Style	—	√	√	—	√	√	√	√	√
COVADIS Style	√	√	√	N/A	—	—	√	—	—
DD	√	√	—	—	√	√	√	√	√

Table 3.2: Properties of v & v approaches

3.2.3 Summary

Table 3.2 summarizes how well different methods from the knowledge base V & V community meet the properties of an ideal error detection system. Both the manual approach and Teiresias style approaches have high support for human-level agents (**P4**)–**P10**). But their methods of detecting errors rely heavily on human support. As that support is removed (e.g., by COVADIS style approaches), the ability to identify the vast range of behavioral problems that could occur in a human-level agent decreases dramatically. Throughout the literature, this trade-off is common.

3.3 Model-Based Diagnosis

The field of model-based diagnosis (e.g., [1, 30, 31, 42]) is concerned with identifying faults in behavior through the use of an explicit internal model of the system being examined. The model-based approach compares the system's behavior to behavior that is predicted by the model. Any discrepancy indicates that some portion of the system is not behaving appropriately.

Often, model-based diagnosis is used to identify errors in systems such as those built with mechanical devices or solid-state components. In these systems it is typically well understood how each component should behave in a given situation and how the components should interact with one another. This property often makes it possible to build a logical model of the system that can be used to predict its overall behavior. Although model-based diagnosis has had success in some domains, its basic premise, that parameters for correct behavior are relatively easy to specify, makes it difficult to apply to the problem of evaluating complex, human-level agent behavior.

Model-based diagnosis, does not require human oversight (**P1**), nor does it require examples of expert behavior (**P3**) to detect errors. Its internal model makes it completely self-reliant. However, if it were simple to make a correct model of the domain expert's behavior, there would be no need to validate the knowledge base to begin with. Thus, building a model that correctly and completely emulates expert behavior could, in the extreme case, be identical to building the knowledge base undergoing validation. This opens up an infinitely recursive validation problem that obviously must be avoided (**P2**). As a result, a feasible implementation of this approach is very likely to require a simplified, or abstract, model of the domain expert's true behavior. Performing this simplification, especially in an undirected fashion, risks the possibility that the model will no longer contain information that is critical for detecting errors. In addition, even if the a useful model were built, this approach, like the others before it, suffers from the fact that, in general, it is unclear how many test scenarios must be exampled before the agent's knowledge base can be confidently called free of errors (**P4**).

The costs of traditional model based diagnosis approaches are likely to prove infeasible for use with human-level agents. Nonetheless, these systems are interesting to study because they do have a fair ability to deal with the some of the problems of supporting human-level agents. In particular, because the specification for correct behavior is stored completely within the system's internal model, it can be modified for use in environments with different requirements (**P5**) (although rebuilding the model is likely to be a time consuming and costly exercise as we already mentioned). In addition, the internal model allows these systems to examine many different aspects of reasoning depending on what level of abstraction at which the model describes behavior (**P7**). Furthermore, so long as the model limits its input requirements to information that can be obtained from a human (e.g., actions, and goals), model-based diagnosis could be applied to novice human behavior as well as machine behavior (**P9**). On the other hand, like many other approaches that completely remove the domain expert from the error detection process, model-based diagnosis suffers because there is no way to leverage the tacit knowledge that for one reason or another has not been incorporated into the internal model (**P6**). Similarly, the complete reliance on the internal model means that properties of behavior that cannot be well specified may be hard if not impossible to incorporate into the model correctly (**P8**)—a serious problem for human-level agents.

	Cost				Human-Level Agent Support				
Method	P1	P2	P3	P4	P5	P6	P7	P8	P9
Diagnosis	√	—	√	—	√	—	√	—	√

Table 3.3: Properties of model-based diagnosis

3.4 Knowledge Base Refinement

The knowledge refinement community is closely tied to V & V and complements it by going one step further—attempting to correct errors once they have been identified [4]. The emphasis in this community is on automating the entire debugging process. This is typically done by observing the agent's behavior as it solves a test problem. When the refinement system detects errors, the agent's knowledge is modified to produce correct behavior. In general, refinement frameworks use solved example problems as a basis for deciding what behavior is acceptable or unacceptable. Somewhat surprisingly, although all refinement systems must address error detection at some level, few do so without making very limiting assumptions.

3.4.1 Traditional Refinement Systems

Traditional refinement approaches rely on solved example problems to determine whether the agent is behaving appropriately. These systems encompass both knowledge base refinement (e.g., [5,10,35]), in which knowledge is encoded in production rules, and theory refinement (e.g., [40]), in which knowledge is encoded in logical formulas. Interestingly, the emphasis in most knowledge refinement work has not been the problem of identifying when an error has occurred, but rather the problem of identifying what piece of knowledge is at fault given that an problem occurred.

Because the focus of this community is automation, traditional knowledge refinement tools seek to minimize involvement of both the domain expert and the knowledge engineer. As a result, they rely on example expert behavior (**P3**) rather than direct human oversight (**P1**) or an engineered solution (**P2**). The desire to automate refinement often also gives rise to a bias toward using domain independent methods of error detection. This means that traditional refinement systems have many of the same trade-offs as COVADIS style approaches to knowledge base validation (see Section 3.2.2). However, because refinement

tools also seek to maximize the use of solved problems or example behavior, they have the potential to identify behavioral errors more effectively than automated V & V tools.

Traditional knowledge refinement tools do however have some significant obstacles to overcome before they can be said to truly support human-level behavior. First among these is that as with a manual approach to validation, it is often unclear how many test scenarios should be conducted. There is no indication whether the refinement system will be able to make additional improvements to the agent's knowledge simply by analyzing additional examples. This leads to problems determining when the refinement phase should be over (**P4**). Furthermore, because these systems rely on domain independent methods for detecting errors, they, like COVADIS style approaches are not very adaptable to environments with different specifications about what is and what is not acceptable behavior (**P5**). It is somewhat surprising that refinement tools do not, in general, have a better approach for detecting behavioral errors. However, this is likely to be a result of the fact that most research has been done on classification type tasks in which errors manifest simply as a misclassification.

Because they have been applied mainly to classification type tasks, refinement systems do not meet many of the requirements listed in Table 3.1. By removing humans from the error detection process altogether, they cannot leverage tacit expert knowledge (**P6**). Furthermore, because error identification typically occurs only after reasoning has ended and a classification has been made, they have not shown an ability to evaluate different aspects of the reasoning process (**P7**). Because refinement techniques classify all behavior as either correct or incorrect, and do not involve the domain expert in determining validity of behavior, they do not support imprecise specifications for what should and shouldn't be considered appropriate (**P8**). Finally, although the refinement process as a whole could not be used to help train a human, the methods these systems use to detect errors (i.e., detecting simple mismatches in behavior) certainly can be used to identify errors in human behavior (**P9**).

3.4.2 CLIPS-R

CLIPS-R [35] stands apart from traditional knowledge refinement systems because it allows developers to specify domain dependent constraints on the agent's behavior. CLIPS-R uses solved example problems with detailed specifications about the constraints

that must initially be met during execution and at task completion. Initial and final state constraints specify what the agent's working memory (fact base) should and should not contain as well as bindings for the agent's input (sensor) functions. Goal constraints specify the constraints on the agent's working memory when the goal is completed. Execution constraints, on the other hand, are somewhat more complicated. These are represented as a finite state machine (FSM) indicating the valid sequence(s) of actions that may be observed by the system as the agent proceeds from initial state to its goal.

Although the representation offers a significant degree of flexibility, it also requires significant effort to construct. Not only must the knowledge engineer and domain expert construct test scenarios, and define state constraints, but they must also build up a potentially complex finite state machine representing parameters of correct expert behavior (**P2**). Although Murphy and Pazanni do not describe the process by which knowledge is acquired to build the finite state machine, presumably it requires examining the expert's behavior on a number of example scenarios (**P3**). As in model-based diagnosis, however, the benefit of this engineering effort is a model of behavior that can be used to identify errors in the agent's problem solving process thereby eliminating the need for human oversight (**P1**). Because the purpose of the finite state machine and state constraints is limited to detecting errors, if a knowledge base has been built correctly, any effort devoted to constructing the FSM will have been wasted. For these reasons, CLIPS-R does not minimize human effort effectively.

In its ability to support human level behavior, CLIPS-R is relatively similar to traditional knowledge refinement approaches. Significant differences arise because its explicit use of domain dependent information in the FSM make it adaptable to environments with different requirements (**P5**), although it is unclear if CLIPS-R makes the process of adaption simple. The FSM allows the domain expert to specify constraints that must be upheld *during* reasoning, so CLIPS-R has an ability to identify errors at different points in the process (**P7**). In addition, the basic method of error detection used by CLIPS-R could be turned around to identify errors in a novice human participant with little or no modifications to the basic process (**P9**). On the other hand, like most other automated methods, CLIPS-R does not leverage tacit expert knowledge in any way (**P6**). Moreover, building the FSM requires very intimate knowledge of exactly how behavior should unfold, which means that it would be difficult to make use of the imprecise specifications

	Cost				Human-Level Agent Support				
Method	P1	P2	P3	P4	P5	P6	P7	P8	P9
Traditional	√	√	—	—	—	—	—	—	√
CLIPS-R	√	—	—	—	√	—	√	—	√
IMPROV Style	√	√	√	—	—	—	√	—	√

Table 3.4: Properties of KR approaches

typical of human-level agents (**P8**).

3.4.3 IMPROV & EXPO

Although the majority of knowledge refinement systems, including CLIPS-R, use deviations from solved example problems to detect errors, some systems take a different approach. IMPROV [41] and EXPO [8] use information directly from the agent's underlying architecture and the external environment to determine whether the agent has the knowledge needed to achieve its goals. Like other approaches that do not rely on solved problems, or example expert behavior to determine the parameters of acceptable behavior, these systems are limited to looking for weak, domain independent clues that an error has occurred. In IMPROV, these clues are either the fact that the agent cannot plan a way to solve a problem, or that the agent's actions do not produced the effects anticipated when the plan was formed. Either of these situations indicates that knowledge needs to be modified.

IMPROV style systems perform similarly to traditional knowledge refinement systems, with only a few exceptions. Because they do not require direct human oversight (**P1**), additional engineering (**P2**), or any examples of expert behavior (**P3**) to detect errors, they require minimal human overhead. However, because they use domain independent methods to identify errors, they also suffer from many of the same drawbacks as traditional knowledge refinement approaches. IMPROV style approaches do, on the other hand, have an increased ability to identify errors at different points in the reasoning process (**P7**), because they look for errors during the agent's planning stage and execution stage. As a result, they can find flaws that relate to knowledge about how tasks should be solved, or about how actions modify the world.

	Cost				Human-Level Agent Support				
Method	P1	P2	P3	P4	P5	P6	P7	P8	P9
Manual	—	√	√	—	√	√	√	√	√
Teiresias Style	—	√	√	—	√	√	√	√	√
COVADIS Style	√	√	√	N/A	—	—	√	—	—
DD	√	√	—	—	√	√	√	√	√
Diagnosis	√	—	√	—	√	—	√	—	√
Traditional	√	√	—	—	—	—	—	—	√
CLIPS-R	√	—	—	—	√	—	√	—	√
IMPROV Style	√	√	√	—	—	—	√	—	√

Table 3.5: Summary of the properties in prior work

3.4.4 Summary

Table 3.4 summarizes the properties of error detection methods used by the knowledge refinement techniques described in the previous sections. For the most part, this table follows the anticipated trend in which methods that rely less on human involvement have fewer traits that will be useful for supporting human-level agents. One interesting exception is CLIPS-R, which although it requires significantly more upfront human effort than the other approaches, does not benefit significantly.

3.5 Discussion

Our survey of related work indicates that to date, no approaches meet the requirements for an ideal error detection system set forth in Section 3.1. In particular there is an obvious trend in which techniques that do not rely on human oversight or significant initial human effort have additional difficulties identifying the myriad potential errors that might be exhibited by a complex human-level agent.

In the upcoming chapters we will explore methods of detecting errors in agent behavior. Common to all of the approaches we will explore, is the desire to reduce human the effort of the domain expert and the knowledge engineer, while still allowing them to participate somewhat in the error detection process. We believe that this will give the best trade-off in terms of satisfying all of the constraints on an ideal error detection method. To this end, we focus on approaches that examine and compare expert behavior to agent behavior and yield a report as to what suspicious behavior has been detected.

Using this report, the knowledge engineer can focus his effort on examining isolated areas of the knowledge base that are likely to be faulty. Equally importantly, the domain expert maintains some involvement in the process. This process of analyzing the summary report allows the domain expert to provide the final determination about which of the identified discrepancies are actually errors.

In Chapters 5 and 6, we will examine two methods of detecting errors and producing summary information. Before we begin to examine these approaches, however, a deeper exploration of behavior is necessary.

Chapter 4

Behavior

In Chapter 2, we stated that the original motivation for our research was to increase efficiency and reduce the cost of creating agents that generate correct human-level behavior. Based on an analysis of potential approaches for achieving this goal, we determined that examining methods to automatically compare human and agent behavior may be a suitable way to achieve a significant cost savings. Before we can begin our investigation, however, we must have a thorough understanding of where behavior comes from, what it is, and what it means to be correct.

In the situations we are interested in examining, behavior is generated by an agent as it interacts with an external environment. An agent can be viewed as an entity with a body and a mind. The body serves as the agent's interface to the external world. Its sensors, perhaps eyes and ears, perceive the world and how it changes over time. Its effectors, perhaps arms and legs, allow the agent to interact with, or manipulate, parts of the world. In contrast, the agent's mind serves to guide how the body is used. The mind makes decisions about what actions are likely to accomplish the agent's goals by applying the agent's knowledge to its perception of the current world state. Thus, by changing the agent's knowledge, one can modify the agent's actions and the internal reasoning process that it uses to solve problems. As we have described it, an agent could be implemented completely in software but it could also be a robot or a human. Because the term agent has become entangled with software agent, we will use it mainly in this sense throughout the paper. We will use the term *actor* when we want to emphasize that the entity satisfying the criteria above may be either a human or a software agent.

An actor's behavior is the result of its mind using knowledge to achieve goals. At its most abstract, one might consider behavior to be concisely represented by the environ-

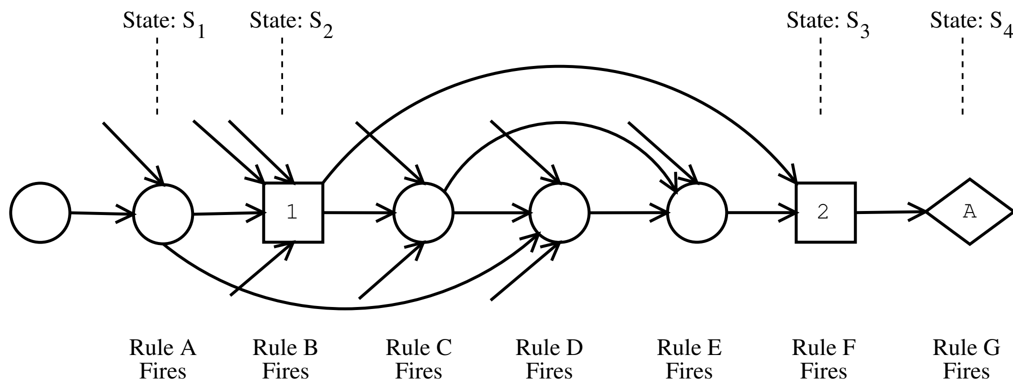

Figure 4.1: A chain of reasoning resulting in action A.

ment's change from the time the actor begins pursuing a goal to the time it is finished. This change is described by the initial and final environmental state pair, (s_i, s_f). In an environment with no exogenous events, this state pair describes the actor's net effect on the environment and thus reflects how the actor has modified the world to obtain its goals. In more complex environments, however, the state pair may include effects that are not a direct result of the agent's behavior. In these situations, describing behavior at such an abstract level is unlikely to be sufficient to determine whether the agent performed its task appropriately. Some details can be added by listing the sequence of actions that were pursued during task performance. We could go further by describing aspects of the actor's internal behavior such as the set of goals that are being pursued during the course of performing a task. In its most specific form, behavior describes the actor's entire mental state and the process by which it decides what goals and actions to pursue.

In the following section, we will examine behavior at different levels of abstraction. We will identify how the level of abstraction impacts the types of behavioral deviations that can be detected, and we will identify a range of abstractions that are appropriate for describing the behavior of both humans and software actors. Then, for the remainder of the chapter, we discuss the distinction between correct and incorrect behavior and how an error detection method can be evaluated.

4.1 The Chain of Reasoning

The process of problem solving can be viewed as a series of steps in which information from the environment, as well as internal information maintained by the actor, is used to select goals and actions to complete the specified task. We call this process a chain of reasoning and it describes one instance of the actor's behavior.

The chain of reasoning is particularly well illustrated by agents implemented in the Soar architecture [23]. In this rule-based system, problem solving is broken into small slices called decision cycles. In each decision cycle, rules from the agent's knowledge base are matched against the current contents of the agent's working memory. Rules that match are fired in parallel, resulting in one or more modifications to working memory. These modifications create or destroy symbolic structures that represent goals or actions the agent is pursuing or intermediate symbolic structures that might be used to make problem solving more efficient or might be used to guide future reasoning.

Figure 4.1 illustrates a portion of an actor's chain of reasoning. This figure depicts how the actor's cognitive architecture, in concert with the actor's knowledge, generates a sequence of symbols over the course of some time period represented by changes to the environmental state (illustrated in the upper portion of the figure). The sequence illustrates modifications to intermediary symbolic structures (circular nodes), goal structures (square nodes) and finally an action (triangular node). Each node indicates a change to a particular symbol structure; in Soar we can think of this as a change to the content's of the agent's working memory. The progression of environmental states indicates the correspondence between the actor's reasoning process and what occurs in the outside world. Note that we can associate nodes with specific times as opposed to relative times by ensuring that a global clock is a part of the state structure.

For simplicity, we have illustrated a chain of reasoning in which symbolic structures are modified completely sequentially. However, it is also possible to imagine tree-shaped chains in which multiple changes may occur simultaneously. Arcs between nodes represent causal relationships between different structures as defined by the actor's knowledge. So, for example, the arc between node 1 and node 2 in the figure indicates that node 1 enables the creation of node 2. Arcs in the figure that are only connected at their head (like three of the four arcs coming into node 1) indicate causal relationships with symbols created earlier in the chain of reasoning and not included in this illustration. Finally, the

figure identifies the knowledge and architectural processes responsible for each symbolic modification. In the figure, as in Soar, rules manipulate structures. Thus the symbol represented by node 2 is created because `Rule F` matches against the symbol represented by node 1 and the symbol created by firing `Rule E`.

The chain of reasoning contains a complete description of all aspects of the actor's behavior. Given the actor's internal state at the beginning of a task, and the chain of reasoning up to task completion, we can identify exactly how the actor's internal state changes as the task is pursued. Although Soar's decision cycle provides a conceptually simple mapping on to the chain of reasoning, it is by no means the only architecture that can be viewed as undergoing problem solving in this manner. Production systems such as CLIPS [3] and OPS-5 [2], BDI architectures [46] such as UM-PRS [27] and JAM [16], as well as some connectionist systems such as those described by Simen [50] can be viewed as using a chain of reasoning to create, modify and destroy symbols that describe sequences of goals or actions (see, for example [56] which describes building similar goal based agents in Soar and CLIPS).

Comparing behavior is straight forward if we have access to two complete chains of reasoning. Simply by examining the ordering and contents of the nodes, we can often identify behavioral deviations even before they manifest as observably distinct actions. Unfortunately, as we describe below, obtaining complete chains of reasoning is unlikely to be feasible in many situations. As components from the complete chain are removed, the description of behavior becomes more abstract. The benefit of increased abstraction is that the behavior comparison becomes less implementation dependent and information required for the comparison becomes easier to acquire. On the other hand, as details about the actor's behavior are removed from the description, it becomes increasingly difficult to pinpoint the location in the actor's reasoning process that is responsible for a particular deviation, thus making it potentially more difficult to repair errors.

4.1.1 Complete Chains of Reasoning

Complete chains of reasoning have three important features. First, they identify exactly how symbolic structures are modified while an actor performs its task. Second, they identify the causal relationship between these modifications. Finally, they identify what knowledge is used to make each modification.

Given this information, we can:

- Identify *any* differences in the sequence of symbols generated during task performance. This allows us to ensure an extremely tight correspondence between the two actors, both in terms of their externally observable behavior and in terms of their internal behavior. Furthermore, because we know the correspondence between elements in the symbol sequence and changes in the environment's state, we can identify differences in timing, even if the relative positions of elements stays the same.

- Identify the origins of deviations. This allows us to isolate exactly where a problem occurred. Since we also know what knowledge is used to make each modification to the symbol sequence, it is also possible to isolate faulty knowledge when a deviation is detected.

- Identify cascading deviations. An actor's behavior is often partially dependent on what it has done in the past. Thus, a deviation that occurs at time t increases the probability that another deviation will occur at time $t'|t' > t$. Because the complete chain of reasoning indicates the causal relationships between symbolic structures, we can identify instances in which a deviation at time t' is a direct result of a deviation at time t. This ability helps to prioritize the order in which behavioral deviations should be examined and resolved.

Examining behavior at this level of detail is extremely useful if a designer wants to identify how an actor's knowledge might be modified in order to change its behavior in a specific way. However, in many situations, obtaining all of the information to describe behavior at this level of detail is not feasible. In some cognitive architectures, for example, the causal relationship between different symbolic structures may not be explicit or well understood. Moreover, it may be difficult to determine exactly what knowledge is used to progress through each step in the chain of reasoning.

Because a complete chain of reasoning may be impossible to obtain from an actor that uses an arbitrary cognitive architecture, this level of representation is unsuited for the general problem of comparing two actors' behavior. However, as we increase the level of abstraction, it will become increasingly difficult to determine the reason why one actor behaves differently from another.

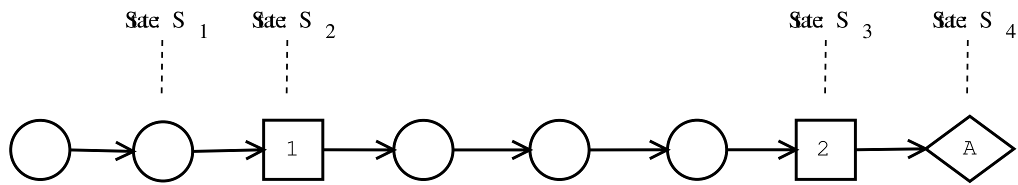

Figure 4.2: A chain of reasoning containing a sequence of symbol modifications.

4.1.2 Complete Symbol Chains

Removing causal links and annotations about what knowledge was used to modify the actor's symbol stream yields a chain similar to the one illustrated in Figure 4.2. This chain of reasoning simply depicts the sequence of symbol modifications that occur during the course of problem solving.

Complete symbol chains retain some of the valuable detection properties of complete chains of reasoning. For example, using complete symbol chains, we can identify behavioral deviations at their origin. Moreover, because we know immediately when an incorrect step in the symbol sequence occurs, we may be able to isolate faults to a relatively small portion of the actor's knowledge base. However the abstraction does come at a cost. Because there is no information about causal relationships between different steps in the symbol sequence, we can no longer determine from the symbol chain how behavior deviations occurring early in the sequence affect later portions of the sequence.

Although many agent architectures may be able to generate a sequence of all the modifications to an agent's symbol structure over the course of problem solving, it is clearly not feasible to ask humans to articulate this same information. Moreover, at this level of abstraction, the description of behavior is still very closely tied to how knowledge is represented. For example, differences in intermediary data structures may occur simply because actors operationalize their knowledge differently. This would make it difficult to compare two software agents implemented by different engineers within distinct agent architectures. As a result, describing behavior at this level of detail is still not appropriate for our purposes.

4.1.3 Chains of Goals and Actions

Removing intermediate symbolic structures from the chain of reasoning leads to a much more concise representation of behavior. At this level of description, we can identify

why an actor is performing specific actions in the world based on the its current goals. However, we can no longer identify all of the internal steps leading up to the selection of goals or actions.

Without intermediate symbolic structures, the chain of reasoning describes behavior in a relatively implementation independent way. Differences in how knowledge is represented or operationalized will go unnoticed so long as the actors still pursue the same goals and perform the same actions. Moving away from a description of the system's internal behavior that had been captured by more detailed chains of reasoning, a chain of goals and actions describes the actor's behavior at the knowledge level [36].

As Newell describes it, a knowledge-level system is located within an environment and performs a series of actions to obtain its goals; it selects between potential actions by using all of its relevant knowledge [36, pp. 50]. Typically, the knowledge level is used to predict an actor's behavior, for instance by stating the expectation that a pilot will pull back on the stick just prior to landing because he knows that this action will decrease his downward velocity, and thus help to achieve his goal of a safe (and soft) landing. For our use, however, we use the knowledge level in the reverse sense. Our aim is to validate the actor's knowledge, or more generally, test for knowledge equivalency. By comparing chains of goals and actions between two actors, we can make a reasonable hypothesis about the similarity of their underlying knowledge even though some aspects of their reasoning cannot be examined.

In addition to supporting a knowledge-level description of an actor and its behavior, by maintaining goal symbols in the chain of reasoning, we can ensure that two actors who behave identically would also explain the motivation behind their actions identically (i.e., using the same goals as justification). This is of pivotal importance for actors involved in training or other pedagogical exercises. Often, such actors will need to explain their goals in order to help students learn appropriate ways to act in a given situation. Recall from Chapter 1 that validation requires the agent (typically a software agent) to generate the same outputs as a gold standard (typically the human expert). Thus, in the situations described above, it is crucial that the behavior representation incorporate goals if a properly validated knowledge base is desired.

Even in situations where an actor's goals will never be communicated to the outside world, ensuring that they are incorporated in the actor's behavior representation is likely

to be beneficial. In contrast to intermediate symbols that can serve many purposes, goals directly and necessarily constrain the types of lower-level behavior that are appropriate within the actor's current context. By comparing chains of goals and actions, we can determine whether two actors constrain their lower level behavior in the same way. Ensuring that this structural organization of goals and actions is consistent between two actors can increase a designer's confidence that both actors will continue to behave identically even in situations that have not been observed.

Unlike more detailed descriptions of behavior, information about goals and actions can be obtained from any willing participant. Because goals play a direct role in determining appropriate actions, humans are likely to be cognizant about what goals they are trying to achieve while performing a task. To externalize the motivations behind their actions, humans can articulate their goals while performing a task or annotate their behavior after the fact. Software agents can simply be programmed to store the sequence of goals and actions they pursue during problem solving.

4.1.4 Chains of Actions

A chain containing only a sequence of actions abstracts away completely from the actors' reasoning. Without information about the actors' current goals, fewer symbols are compared, and thus there are fewer opportunities to detect a deviation.

Examining behavior at this level of abstraction can ensure that two actors' externally observable behavior is identical. However, it suffers from the problem that there may be some cases when an actor performs the correct action, but does so for the wrong reason. For example, if a novice pilot pulls up on the stick just prior to landing because he is trying to allay his fear of approaching the ground, the actions may be acceptable, but the motivation is undesirable. Because the relationship between the actor's goals and external actions is completely hidden, it is particularly difficult to form reasonable hypothesis about how the actor may perform in new situations.

Although actions sequences are not nearly as useful as sequences that include information about goals, one of their main assets is that they can be obtained simply by observing the actor's behavior as it performs the specified task. This means that the actor does not need to directly participate in constructing the action sequence. They can be generated without the actor's knowledge simply by unobtrusively observing their

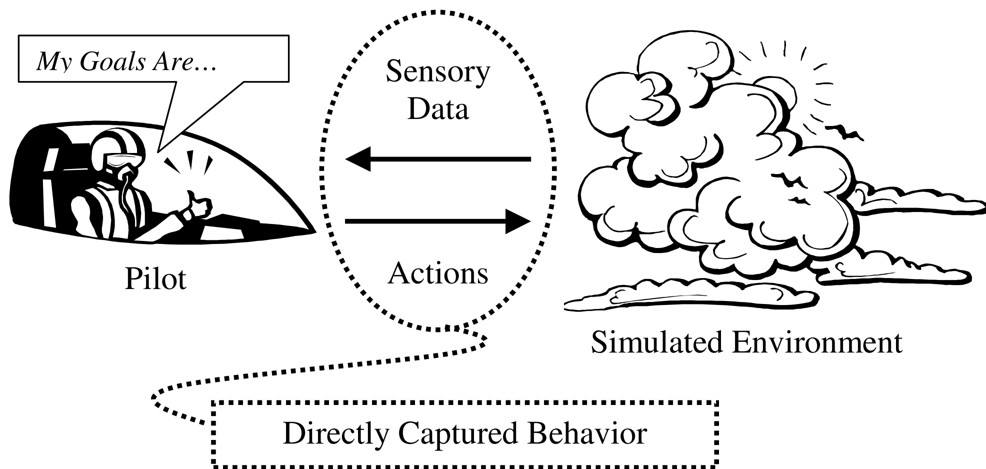

Figure 4.3: Capturing a behavior trace using a simulator

interaction with the environment.

4.1.5 Appropriate Abstractions for Human-Level Agents

As we described in Section 2.1, human-level agents are designed to reproduce the behavior of a human expert within a domain. Although we may wish to compare human-level agents to one another, we also need the ability to compare the software agent's behavior to that of their human counterpart. In this way, behavioral comparison could be used for validation or to objectively score a Turing test. Because we are limited by the human's cognitive architecture, many details encoded by the complete chain of reasoning are unavailable. As a result, our work focuses on techniques that compare behavior using no more information than would be available from chains of goals and actions.

4.2 Behavior Traces

In the previous section, we argued that when dealing with human-level actors, the most appropriate level of abstraction for defining behavior is as a sequence of goals and actions. We call these sequences *behavior-traces*, since each trace corresponds to a single instance of the actor's behavior.

Capturing a behavior trace is a relatively straight forward process if an environmental simulator appropriate for the specified task is available [53]. Consider a human pilot per-

forming flight maneuvers in a simulator (see Figure 4.3). For the purposes of generating behavior traces, the simulator has two important features.

First, it maintains relevant information about the external environment such as wind speed and geographic data. It presents this data to the human through some sort of interface, perhaps an instrument panel and a virtual cockpit window. This interface makes it is very easy to identify what the pilot knows about the external world at any point in time because all this information must be generated by the simulator and must be presented by the interface. A well built simulator provides the human with all of the information about the environment that is required to perform the task appropriately[1]t follows that if the simulator fails to provide relevant information, it is not well designed. This is clearly the case because such a simulator would not allow the actor to base its decisions on the same information that would be used to make decisions in the real-world.. As a result, one can collect all of these potentially salient features simply by recording what information is presented to the pilot.

The simulator's second important feature is that it provides an interactive experience controlled in part by the human's actions. This means that the simulator must be aware of all actions the human performs during a task. For a pilot, this means actions such as pushing or pulling on the stick, and raising or lowering the landing gear. By recording the output of the pilot's controls we can pair what the human does (its actions) with their perception of the environmental state (its sensory input).

The simulator thus allows us to easily capture state-action pairs (s, a). To construct goal-action sequences, however, we must also identify what goals the pilot pursues at each point in time. This can be done in two ways. During task performance, the pilot can articulate what goals are being pursued and how they change over time. Alternatively, the pilot can examine his own behavior after the task is completed and provide goal annotations post-hoc. Either way, this goal information completes the behavior trace, creating a sequence of tuples $B = ((s, G, a)_0, (s, G, a)_1 \ldots (s, G, a)_n)$.

Constructing a behavior trace for a software agent follows similar lines as the procedure above. The only significant difference arises while recording the actor's goals during task performance. To make goal annotations, the software agent can simply record the mapping between environmental states and actions allowing the sequence of $(s, G, a)_i$ to

[1]I

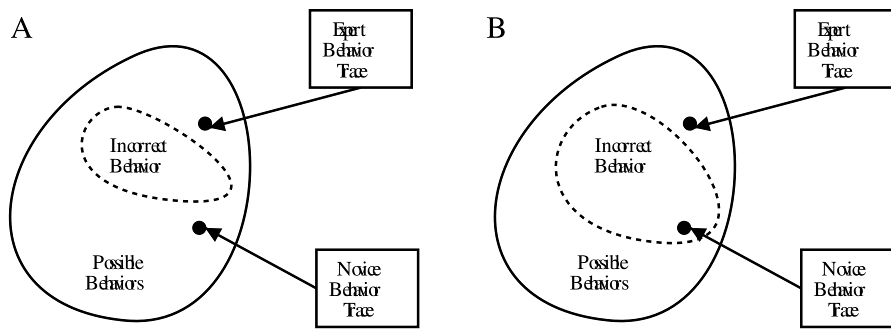

Figure 4.4: A space of possible behaviors

be easily constructed.

4.3 Correctness

The behavior traces describe an actor's behavior in terms of goals and actions. Given two such behavior traces, it is clear that we can identify how these traces differ simply by comparing each of the elements in the traces. We can think of the space of possible behaviors as being divided into distinct regions: areas of correct behavior and areas of incorrect behavior. Each behavior trace corresponds to a distinct point in this space. Two traces that cover exactly the same point in the behavior space are identical. Two traces that cover different points deviate from one another. If the goal of the behavior comparison is simply to determine whether two instances of behavior differ from one another, all that must be done is to determine whether these instances occupy the same point the behavior space. However, if we wish to go a step further by distinguishing acceptable deviations from errors, the problem is more difficult.

To distinguish errors from acceptable deviations, we begin by partitioning the behavior space into regions of correct and incorrect behavior. If two traces both fall within areas of correct behavior, the deviations are insignificant; they do not correspond to errors in the actor's knowledge. On the other hand, if one behavior trace falls within the area of correct behavior and the other does not, then these differences indicate errors in the novice actor's behavior and corresponding problems in its knowledge.

Clearly, determining the boundary between correct and incorrect behavior is the most crucial aspect of the error detection process. In the domains of human-level agents, formal specification alone is often not enough to identify this boundary. Instead, we must rely

on examples of behavior provided by the domain expert. Unfortunately, because the range of acceptable behavior varies widely across domains, it may not be possible to be completely confident about the boundaries between correct and incorrect behavior unless we observe all the possible ways in which the expert might perform her task. Thus, even if two traces have very few deviations and thus lie close to one another in the space of all possible behaviors, it may be difficult to distinguish between the two situations illustrated in Figure 4.4: A and B. In particular, determining whether a deviation is simply an inconsequential difference, or an error is difficult for the following reasons:

- **Differences between two behaviors do not necessarily imply an error.** Human-level agents are meant to operate in high fidelity simulations and in real-world environments. These environments have a large potential goal and action space, and as the size of this space increases, there is an increased likelihood that problems will have multiple solutions. This means that the simple observation that a novice and an expert performed different actions or pursed different goals is not, by itself, enough information to determine whether an error has occurred.

- **Criteria for determining correctness may change during problem solving.** Complex tasks require solving many simple problems. Each subproblem may be viewed as a task in itself, and as such, the criteria used to judge whether it is correct may be unique within the scope of the overall problem. For example, air speed may be relatively unimportant during many points in a pilot's flight, but may nonetheless be crucial for correct behavior when the pilot is landing his aircraft. This means that error detection methods must be adaptable to suit a number of different situations. Otherwise, their use will be severely limited.

- **Criteria for determining correctness may depend on higher level task goals.** In some situations, primitive actions may be the wrong level of abstraction for determining whether a particular behavior is correct. In such cases, it may make more sense to leave the exact implementation open ended and perform an evaluation based on the motivation for performing each action, or goal. In the case of a military pilot, for example, a dogfight may be considered correctly executed so long as the enemy is destroyed even though the exact sequence of actions used to accomplish this goal may differ from observations of an expert.

- **The context of the task affects which solutions are acceptable.** In contrast to the previous criteria, the context of the task may not be represented in the agent's goals explicitly. Instead it may simply be an aspect of higher level assumptions about the environments or circumstances that the agent will encounter. For example, there may be significant flexibility in choosing a tactic of engagement when only a small number of planes are involved, but if the attack is assumed to be part of a larger, coordinated effort, tactics that would otherwise be acceptable may be considered inappropriate in this situation. These assumptions need to be articulated to ensure that the performance metric used to judge the agent's behavior is appropriate.

- **Only a fraction of available information determines whether an error has occurred.** In complex, real-world environments the state space is often very large if not infinite. In order to deal with these worlds, actors must efficiently abstract their sensory information. As a result, only a fraction of the information is required to decide what action should be pursued at any given point or whether the actor is behaving appropriately. Thus, as the environment grows more complex, error detection becomes more difficult because detection methods must correctly differentiate between features of the environment that are important for determining whether the actor is behaving correctly, and features that should be ignored.

- **Problem solving strategies may be flexible and diverse.** In nondeterministic environments, actions may not always have their intended effect. As a result, a procedure may fail unexpectedly even though it is the correct thing to do. To deal with such situations effectively, actors are likely to benefit from a flexible and diverse set of strategies that accomplish the task using different means. This complicates error detection because a method that overly constrains the actor's behavior (for example, by ensuring that the actor's actions are identical to the expert's actions) may adversely affect the actor's performance when unexpected situations arise.

When comparing two behavior traces, none of these difficulties will be surmountable without knowing a good deal about how the space of correct behavior should be specified. One obvious way of obtaining this knowledge is simply to acquire more information about the scope of expert behavior. In this way, the set of expert behavior traces themselves can

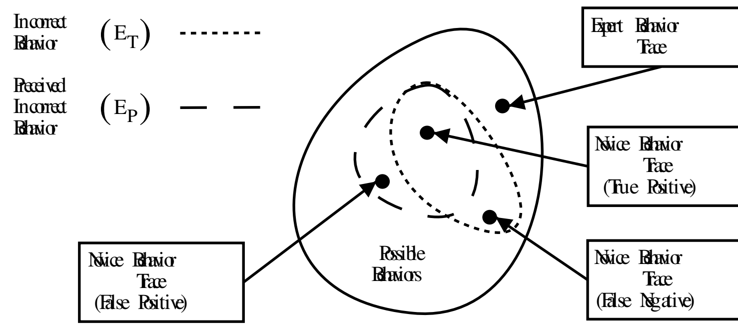

Figure 4.5: An incorrectly identified boundary between correct and incorrect behavior

be used to determine the range of acceptable behavior and how this range may change while performing different aspects of a task. However, because it is unlikely that a complete set of behavior traces can be acquired for all but the simplest or most constrained tasks, it is likely that some uncertainty as to how errors should be distinguished from acceptable deviations will remain.

Given the difficulty of identifying where the boundary between correct and incorrect behavior lies, it is reasonable to wonder what effect an incorrect specification would have. Figure 4.5 illustrates a situation in which this boundary has been incorrectly perceived by the error detection system. Recall that expert behavior is the gold standard against which the novice will be compared. Thus by definition, expert behavior must fall within the the region of correct behavior as it does in the illustration. Novice behavior, on the other hand, may fall in any part of the behavior space.

Figure 4.5 illustrates two problems that arise when the perceived area of incorrect behavior (E_P) does not match against the true area of incorrect behavior (E_T). In the first situation (the point on the left side), the novice's behavior falls within E_P and is classified as an error. In reality, however, the novice has not committed an error (the behavior is not in E_T), so the error detection method's inaccurate classification is a false positive. In the second situation (the lower point on the right side of the figure), the novice's behavior falls outside of the region perceived to be incorrect behavior and is classified as acceptable. However, the novice's behavior actually falls within E_T, so the error detection system's classification is again incorrect; this is a false negative. Finally, the third point, located in both E_T and E_P, illustrates an example of a correctly detected error, a true positive.

	Predicted Negative	Predicted Positive
Actual Negative	True Negatives	False Positives
Actual Positive	False Negatives	True Positives

Figure 4.6: Confusion matrix

A matrix can be formed by analyzing the similarities and differences between the perceived classification and actual classification. This matrix is sometimes referred to as the confusion matrix [22]. When classification distinguishes between two groups (as in our situation) the matrix contains four elements, true and false negatives and positives, illustrated in Figure 4.6.

At times, the performance of classifiers is based largely on a single metric: accuracy. Accuracy measures how many classifications were correct.

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$

In our evaluation, the benefit of this measurement would be primarily to describe how well the perceived boundary between correct and incorrect behavior matches the actual boundary with a single, concise figure. The disadvantage is that this measurement obscures the role played by each type of misclassification (i.e., false positives and false negatives) in achieving sub-optimal performance. For our purposes, we prefer measurements that explicitly distinguish between the effects of false positives and negatives. Two such measurements are sensitivity and specificity.

Sensitivity is the ratio of true positives (actual errors that are also perceived to be errors) to the sum of true positives and false negatives (actual errors that are perceived to be acceptable behavior).

$$Sensitivity = \frac{TP}{TP + FN}$$

For finite behavior spaces, as sensitivity goes to zero, the area of the behavior space perceived to be incorrect, E_P, actually covers less and less of the space of truly incorrect behavior E_T, thus increasing the number of false negative, and decreasing true positives. For finite behavior spaces, at zero sensitivity, $E_P \bigcap E_T = \emptyset$. As sensitivity goes to one,

the area of perceived incorrect behavior covers more and more behavior that is in fact incorrect until at one, $E_T \subseteq E_P$. Notice that sensitivity remains one even if the error detection method's perception of incorrect behavior is too large (i.e., covers the bounds of actual incorrect behavior). Specificity, the ratio of true negatives to the sum of true negatives and false positives, measures this tendency.

$$Specificity = \frac{TN}{TN + FP}$$

As specificity goes to zero the perceived area of correct behavior covers less and less area that is actually correct. So, at zero, $\neg E_P \bigcap \neg E_T = \emptyset$. As specificity goes toward one, the area of behavior perceived to be correct covers more and more behavior that is in fact correct until at one, $\neg E_T \subseteq \neg E_P$. Thus an error detection method whose classification scheme has a sensitivity and specificity of one perceives the boundary between correct and incorrect behavior to match the actual boundary.

Clearly, both false negatives and false positives are undesirable. Unfortunately, efforts to reduce these artifacts are likely to be at odds with one another. By being more selective about what deviations constitute an error, it is straightforward to decrease the number of possible behaviors that are perceived to be incorrect. Although this will help reduce false positives and thereby increase specificity, it has the potential of increasing false negatives and thus decreasing sensitivity. Likewise, decreasing selectivity about what deviations constitute an error will reduce the number of negatives, but will also likely increase the number of false positives identified.

We can make an informed decision about the appropriate balance between sensitivity and specificity by examining how incorrect results from the effects of false positives and false negatives in more detail. False negatives indicate that errors go undetected, and thus undermine a main motivation for this work, the prospect of automating knowledge base validation. However, if we can classify the types of errors that are hard to identify, or the situations in which false negatives are likely to occur, we can go a long way to reducing the impact of this deficiency.

Although intuitively less problematic, false positives also reduce the efficacy of an automated error detection system. Most obviously, false positives result in an incorrect assessment of the novice's ability to reproduce the expert's behavior. But because the information reported by the error detection method is likely to be analyzed by a human,

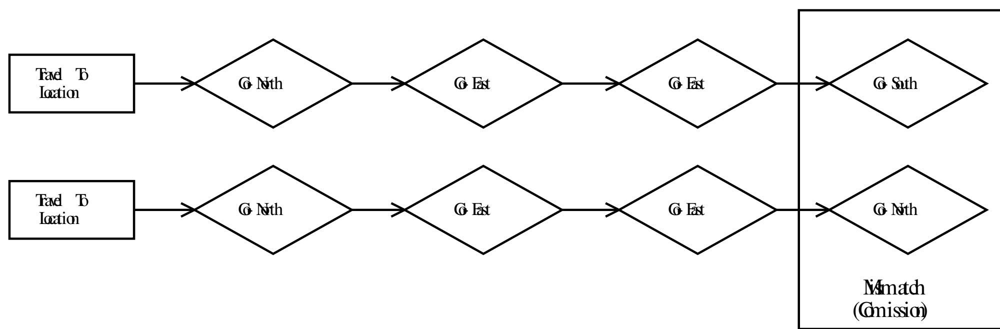

Figure 4.7: Mismatches in the behavior trace indicate errors

it is important to note that false positives also require significant human effort to examine and follow-up before they can be dismissed. If a report is laden with false positives, it is unlikely that anyone would actually care to read it. On the other hand, if the report produces a large amount of good information, an occasional red herring is likely to be accepted. A final measurement, Report Density, which is described in Section 4.6, is an alternate method to measure the impact of false positives and indicates the tendency of a report to be dominated by spurious information.

4.4 Errors

In the previous section, we noted that behavior errors are a subclass of more general behavior deviations. Here we examine the ways in which these errors manifest in behavior traces. As we will show, some basic properties of the behavior deviations are constant and allow us to categorize errors into distinct classes. These classes serve two purposes. First and most importantly, they provide a framework for evaluating an error detection method's performance. Second, they aid in designing experiments by enumerating types of errors that should be investigated.

At the simplest level, all errors can be identified by a single discrepancy between two particular symbols in the behavior traces (see Figure 4.7). Such a mismatch can occur in one of three ways.

Commission If the novice's behavior trace and the expert's behavior trace both contain a goal or action symbol at the specified location, but these goals or actions are inconsistent, an error of commission has occurred.

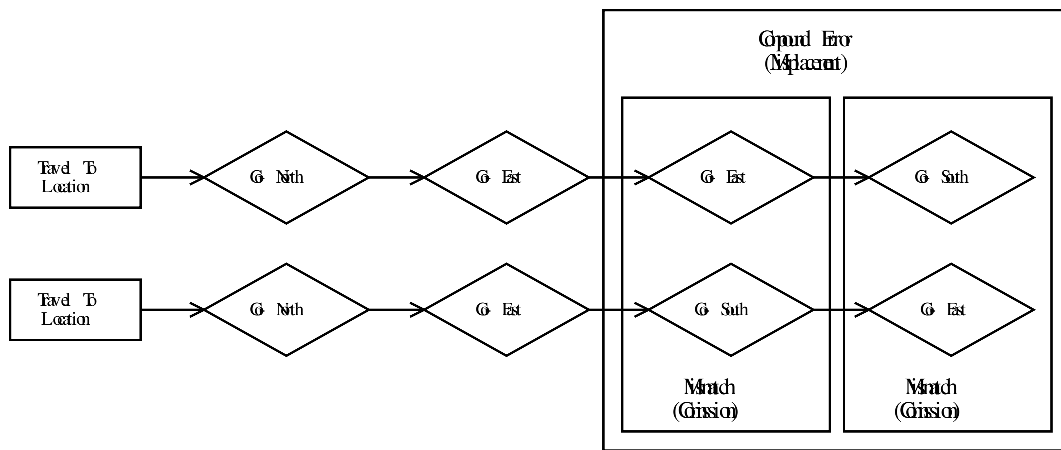

Figure 4.8: Multiple simple errors may form a compound error

Omission If the expert's behavior trace contains a goal or action symbol where there is no corresponding symbol in the novice behavior trace, this error is an omission.

Intrusion The final simple error type, intrusion, is identical to omission except that the goal or action symbol occurs in the novice's behavior trace but not in the expert's behavior trace.

Typically, it is relatively straightforward to classify errors into these three categories, especially considering that we can compare the novice-level agent's knowledge to the expert-level agent's knowledge. However, in some situations there are enough differences between the two actors behavior that it can be difficult to determine whether a deviation is a commission, or one of the other forms. In such situations, we mark the error as belonging to either category and consider it acceptable for an error detection method to identify it as either form.

When we consider more than one element in the expert and novice's behavior trace at a time, it is possible to identify relationships between multiple errors. Because these compound errors concisely describe multiple simple errors as well as the relationships between them, uncovering a single compound error is preferable to identifying many simple errors. We identify two types of compound errors.

Misplacement Two simple errors can indicate that a particular symbol is misplaced in the behavior trace. The typical manifestation of this error begins with a simple error in which the expert behavior trace specifies symbol m_i to occur, but the novice

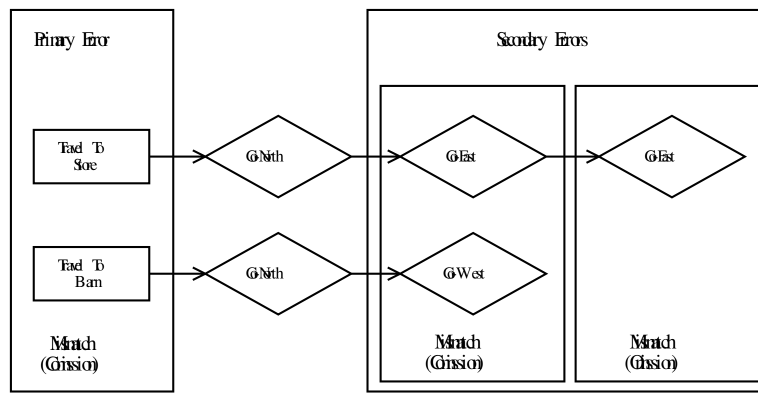

Figure 4.9: A cascade of errors follows an incorrect goal selection

behavior trace indicates that another symbols, m_j, has been pursued. If later in the behavior trace another simple error occurs in which the symbol m_j in the expert's behavior trace is matched against the symbol m_i in the novice's behavior trace, a misplacement can be identified (see Figure 4.8). Note that if the error only involves one symbol being out of place, then it will be identified simply as an omission, commission or intrusion.

Duplication Duplications occur when the novice behavior trace contains a sequence of errors in which one or more symbols inappropriately reoccurs. For example, if the expert's behavior trace contains n instances of symbol m_i, while the novice's behavior trace contains $n + 1$ or more instances of this symbol, a duplication error has occurred.

We have described simple and compound errors that occur at the level of individual symbols (e.g., two symbols being misplaced). However, errors can also occur among subsequences in the behavior trace (see Figure 4.9). This typically happens after the novice begins to pursue an incorrect goal. In such a situation, there is a causal relationship between the initial error and the sequence of errors that follows. We define two more error forms based on these attributes.

Primary Error A primary error is the first in a causally linked sequence of errors.

Secondary Error A secondary error occurs as a direct result of the primary error. Although it is a problem in and of itself, it can be corrected simply by correcting the primary error.

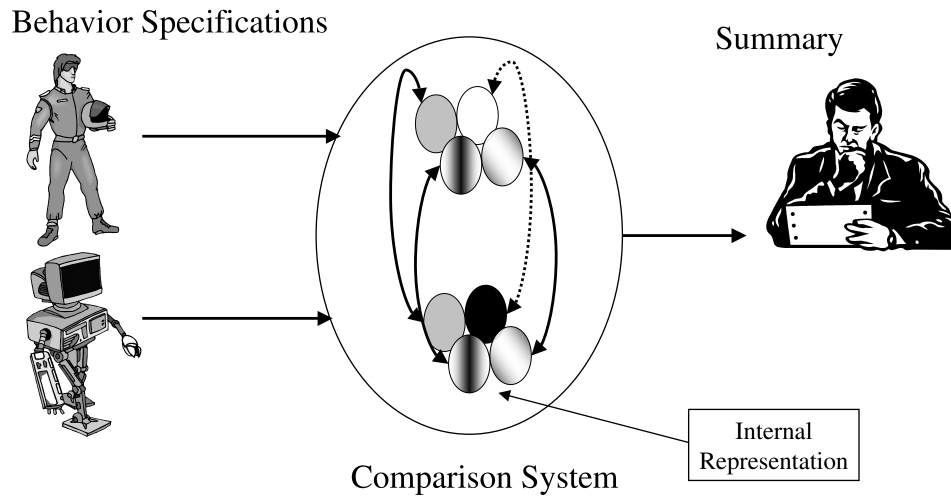

Figure 4.10: The framework of a behavior comparison or error detection method

Just as compound errors are more salient than simple errors because they concisely describe multiple simple errors as well as the interactions between them, a primary error is more salient that the secondary errors that follow. Note that since a single error can act as both a primary and secondary error (if a hierarchy of cascading errors occurs), the primary/secondary relationship creates a corresponding salience hierarchy. Errors toward the top of this hierarchy are more salient, with the most salient being the initial primary error. Errors toward the bottom are less salient because they can be corrected by fixing any preceding error in the causal chain.

4.5 Behavior Comparison & Error Detection Methods

In the previous sections, we examined basic definitions of behavior, correctness and errors, but we have yet to specify the basic framework of the behavior comparison methods that we will examine in the subsequent chapters. We can think of a behavior comparison framework as a simple input/output system as illustrated in Figure 4.10.

This simple system must accept two specifications of behavior as input, one for each of the actors involved in the comparison (illustrated in the figure by the arrows on the left hand side entering the oval). The instantiations of this framework that we will examine in the upcoming chapters rely on two sets of behavior traces for this input. Given these

specifications, the comparison system builds an internal representation of both actors' behavior. This representation is illustrated by the small shaded shapes in the interior of central oval. The internal behavior representations may be created by generalizing, merging or modifying the original specifications, or they may simply be a duplication of the original specifications themselves.

The critical aspect of these internal behavior representations is that they are the data structures that will be compared to one another, not the input specifications themselves. This results in two significant properties. First, the actual comparison process will vary according to the properties of these internal structures. For self-similar structures such as sequences or hierarchical forms, it is likely that the gross comparison can be decomposed recursively into a set of simpler comparisons between individual components of the data structures such as elements in the sequence or nodes in the hierarchies. Second, the types of errors that can be detected will be influenced by properties of the internal representation. This means that determining what internal representation to use may have a significant impact on the efficacy of the comparison. If selection of the internal representation is constrained by resource or efficiency concerns, for example, it may not be possible to achieve maximum efficacy.

Once the comparison is completed, the system outputs a summary of the similarities and differences in the actors' behavior and this summary is analyzed and interpreted by the domain expert and knowledge engineer. Note that this report describes how the internal representations of behavior differ. If the comparison is being used to validate an agent's knowledge then the ultimate goal is to use this summary to identify portions of the agent's knowledge base that require correction. This means that the knowledge engineer must interpret the summary to determine what modifications are required. Thus the internal data structures used to represent the actors' behavior impact the efficacy of an error detection method both directly, because it is possible that only a subset of errors will be identifiable, and indirectly, because the representation will determine what information appears in the summary and this may be either simple or difficult to map onto changes that must be made in the agent's knowledge base.

4.6 Evaluating an Error Detection Method

In the previous sections, we illustrated three components that play a central role in determining how useful an error detection method will be. The first of these is the ability to discriminate between correct and incorrect behavior. The second component is the type or types of errors that can be identified. Finally, the choice of internal representation affects both what errors can be detected and how easy it is to correct an agent's knowledge based on information contained in the comparison method's report. Recalling the properties of an ideal error detection system outlined in Chapter 3, it is clear that both of these components play a direct role in determining how much human effort is required to make use of the automatically generated report. In this section, we present our approach for examining the performance of different error detection methods.

Ideally, an empirical evaluation would directly examine how much human effort is saved by using each automated error detection method during the development of a number of complex human-level agents. One way we might do this is by implementing multiple agents in parallel, using conventional validation methods on one branch and distinct automated error detection methods on the other branches. All branches would begin at the same point (a non-existent agent) and end at the same point (correct agents, all of which generate identical behavior). By examining how much time was spent on each of the development branches, we could measure the effectiveness of each error detection method and validation approach. In addition, we could further analyze each method's ability to deal with different types of errors by investigating what errors occurred during development and the relative speed at which they were fixed. Finally, given some knowledge about how likely it is for each type of error to occur during construction of other human-level agents, we could make a reasonable hypothesis about how much savings each error detection method might make in the general case.

Unfortunately, as we have already noted, developing human-level agents is an extremely time consuming process, often requiring many person-years. The cost of developing these agents is already a serious roadblock in their deployment, and increasing this cost by even a factor of two or three is a significant additional impediment. As a result, such approaches are not feasible for this study. Nonetheless, there is still a good deal to learn about the relative effectiveness of automatic error detection methods using other experimental approaches.

1. Acquire a specification of correct (expert) behavior.
2. Construct a set of flawed novice agents.
3. Identify general differences by comparing the expert's and the novice's knowledge.
4. Acquire suitable behavior traces from the expert and novice.
5. Manually catalog errors in each novice behavior trace.
6. Evaluate how well each error detection method identifies the cataloged errors.

Figure 4.11: An overview of the steps in our evaluation process

The approach we've selected identifies the effectiveness of error detection methods without directly examining development time. Like the approach described above, we evaluate the effectiveness of each error detection method by examining its ability to identify different types of errors in development versions (novice versions) of a particular agent. By examining the number of true errors detected, as well as false negatives and false positives, we can obtain a confident measure of the relative strengths and weakness of each approach without directly examining how development time is impacted in a ongoing project. Our evaluation process is described by six high level steps outlined in Figure 4.11 and described in detail below.

Our evaluation begins with a specification of correct behavior. Under normal development circumstances, the specification of correctness would be the domain expert's behavior. For our experiments, however, we replace the domain expert with a correctly specified *expert-level agent*. The idea of replacing the human expert with a validated software agent may initially seem counterintuitive. After all, our research seeks, in large part, to make it easier to create agents that reproduce human behavior, not the behavior of other software agents. However, this approach offers significant advantages over other evaluations methods.

The first advantage gained by replacing the human domain expert with an expert-level agent is that we can ensure that both the expert-level agent and the novice agent (the agent that is being validated) represent their knowledge in a similar manner. This provides an additional means of determining how the expert and novice behavior differ than might otherwise be available: not only can we examine instances of the actors'

behavior to determine differences, but we can also directly compare the knowledge that guides their behavior. This can be be an important asset to help ensure that appropriate behavior traces are collected, and to ensure that the our expectations about what errors should be detectable are in fact correct.

The second advantage gained by replacing the human expert with a software agent is that we can test an error detection method's efficacy without being influenced by the complications of the knowledge acquisition process. Moreover, since we ultimately believe that many aspects of human-level behavior can be duplicated by software agents, replacing the human expert with an expert-level software agent should not change the generality measurements. On the other hand, by examining behavior that is already encoded in the software agent's knowledge, there is the potential that this methodology will bias us toward examining behaviors that are easy to encode in software as opposed to the complete breadth of human behavior.

Given the expert-level agent, we begin the second step by constructing novice agents which are partially correct versions of the final desired behavior. The novices are only partially correct since they pursue different sequences of goals and actions than the expert-level agent. These difference arise because the novice-level agents do not have knowledge bases that are identical to the expert-level agent. Instead, some portion of their knowledge base has been incorrectly produced or corrupted.

Novices can be constructed in a number of different ways, but we focus on novices that are generated by introducing random changes into the expert-level agent. Introducing random changes helps to ensure that we examine a wide range of possible errors and that we minimize the potential to bias the experiments' results. Moreover, by effectively maintaining a large body of shared knowledge between the expert and the novice agents, it is straightforward to map the novice agent's correct knowledge onto the expert's knowledge, and also to isolate problematic knowledge to a specific portion of the novice's knowledge base. This allows us to take maximum advantage of the fact that we are using an expert-level agent as opposed to a human domain expert.

The major drawback of constructing novice-level agents in this fashion is that it is unclear whether the manner in which we manipulate the agent's knowledge base in order to introduce errors is representative of flaws that would occur naturally during the development process. If our comparison systems examined the novice-level agent's knowledge

base directly, this would indeed be a serious concern. However, in all of the systems we address, errors are identified phenomenologically—by examining the agent's behavior. As a result, the main concern should be that the novice-level agents we construct generate the same types of observable errors as development version of these agents. Our novice-level agents create flaws that cover all the error types we identified in Section 4.4. Thus, we should have a high degree of confidence that the changes we introduced do represent many of the observable errors we would expect to see in a development version of an intelligent agent.

Once we have constructed a set of novice-level agents, we must determine the exact set of behavioral errors they are capable of producing. This third step requires careful manual examination of the knowledge used by, and the behavior produced by, both the novice and the expert. We begin the process of documenting errors by analyzing how the novice's knowledge differs from the expert's knowledge. Based on this analysis, we can often identify general situations in which the novice's behavior will diverge from the expert's behavior. These general situations provide a high level description about the errors that will arise. For example, we might be able to determine that the novice will fail to perform a specific action when trying to accomplish a particular goal, or that it might pursue a goal on inappropriate occasions. However, if we consider how difficult it can be to predict the behavior of an intelligent agent simply by examining its knowledge, it is not surprising that in many cases it is hard to determine the exact forms in which each of these general errors may manifest using information about the differences in the agents' knowledge alone.

The fourth step is to acquire concrete examples of both the expert's and novice's behavior by gathering the behavior traces that will be used to compare the agents' behavior. In most situations, human-level agents will be capable of performing their specified task in many different ways. In order to examine a significant range of these behaviors, traces are selected randomly from this pool of possible behaviors and then examined to ensure that two properties hold. The first property we want to ensure is that no two behavior traces are identical. The second property we want to maintain is that all of the predicted errors actually occur in at least one behavior trace. While we are examining behavior traces to ensure that the second property holds, we can also perform the fifth step in our process by cataloging the specific form or forms in which each error

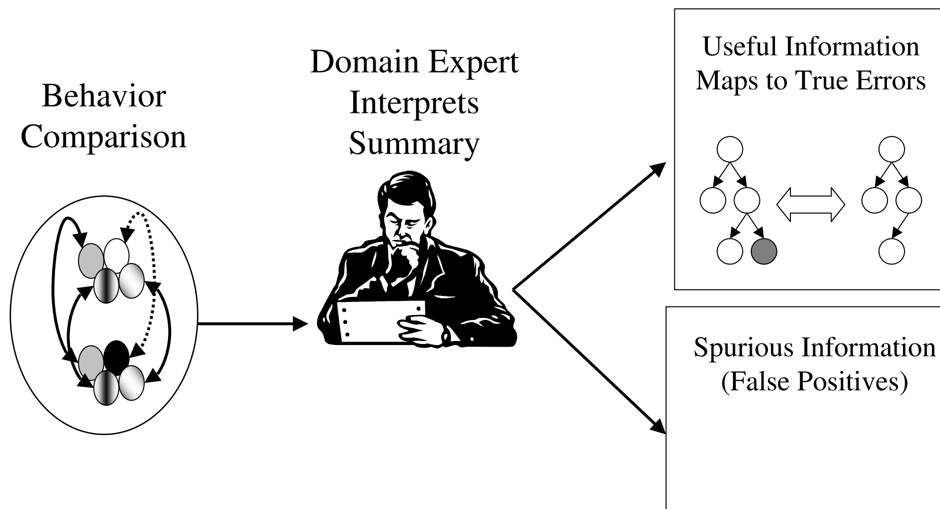

Figure 4.12: Interpreting summary information from an error detection method.

manifests. In this way, we annotate all of the attributes of the error (e.g., whether it is primary or secondary, omission or commission). The information cataloged during this process will be used later to determine the set of errors that were and were not detected by a particular approach.

Finally, at this point we are ready to begin evaluating each of the individual error detection methods. It is important to recall that any error detection method that relies on examining examples of behavior suffers from the potential problem that unless an error manifests in the examples that are being examined, it cannot be detected. Thus, the goal of our experiments is to determine how many of the errors that occur in the novice behavior traces can be identified by a particular error detection method. In most cases, the traces being examined will contain all the errors the novice is capable of producing. However, in certain situations, such as when only a small subset of the available traces are being examined, it will only be possible to detect a similarly reduced number of errors.

Given two sets of behavior traces, one corresponding to the expert-level agent, and the other corresponding to the novice agents, the automated error detection method examines these instances and prepares a report indicating similarities and differences in the behaviors. This report will be more or less useful depending on how well the error detection method performs. By definition the expert-level agent is the standard of correct behavior, so any true differences are instances of inappropriate behavior or

errors. By examining the information in the report, we determine whether any of the information in its summary maps on to error forms identified in the manual examination of the novice's behavior traces. If so, these are instances of true positives (correctly detected errors) that improve the error detection method's performance score. At the same time we also want to identify how many true negatives as well as false positives and false negatives have been identified. Figure 4.12 illustrates the process described above. Used in a real validation setting, as opposed to an evaluation setting, the process would be much the same. The critical difference is that determining whether information in the summary maps to true errors or to false positives would be likely to require some amount of investigation (either by manually examining some examples of behavior, or examining the novice agent's knowledge base).

Because the error forms identified in Section 4.4 do not form sets with mutually exclusive membership, and because some forms are more salient than others, we must be careful how true and false positives and negatives are calculated. Consider, for example a high level error description such as *The pilot does not always contact the control tower prior to initiating a landing*. Suppose that this error manifests in two ways: by the pilot failing to contact the control tower completely, or by the pilot contacting the control tower after the landing had been initiated. Depending on the circumstances, these manifestations may take the form of an omission in the first case, and as an omission plus an intrusion in the second case. In addition, since the second case involves an action being moved to an inappropriate location in the agent's behavior sequence, it is also an instance of a misplacement error. This means that depending on the set of behavior traces being examined, the high level error may manifest as just a single simple error (perhaps an omission), or as a set of three errors (two simple errors and a misplacement). Exactly how the values in the confusion matrix are calculated depends both on what errors manifest in the behavior traces, and what errors are detected by the automated system.

In the first case, where the pilot simply fails to contact the control tower, counting is simple: either the omission is identified or it is not, there is nothing more to it. If the error is identified, the number of true positives is incremented by one, otherwise the number of false negatives is incremented by one.

The second case, in which the pilot belatedly contacts the control tower, is not so

simple however. Here, the errors may be identified in one of a number of ways. The first of these ways occurs if only one of the simple errors is found; for example identifying that the pilot failed to contact the tower before landing. In this case, it is clear that at least one false negative occurred, because the detection method did not identify the error of intrusion when the pilot belatedly contacted the tower. However, as we mentioned previously, the error of misplacement carries more information that the sum of two simple errors: it also identifies a relationship between these errors. So this situation is more appropriately reflected by counting two false negatives.

Similarly, if both simple errors are detected, but the compound error goes unidentified, one false negative should be counted. And, if all three errors are correctly identified, clearly three true positives occur and zero false negatives. Two of the more interesting cases arise, however, when the compound error is detected, but the report makes no explicit mention of either of the simple errors and when no errors are identified at all.

When only the compound error is detected, determining the most appropriate way to count the elements in the confusion matrix requires more thought. We could count this as two false negatives (the simple errors) and one true positive (the compound error). However, this approach is unappealing because the compound error encapsulates the constituent simple errors and thus a knowledge engineer that corrects the compound error will necessarily correct the simple errors. As a result, we count zero false negatives in this situation. Similarly, counting only one true positive may bias our measurements in undesirable ways. In particular, it will favor error detection methods that identify many simple errors over those that identify fewer compound errors. In order to reduce human effort as much as possible, however, we should favor reports that concisely describe differences between the actors using compound errors in lieu of multiple simple errors whenever possible. This helps to ensure that the error reports are simple to examine, understand, and put to use because little time is wasted identifying and filtering through extraneous information. Based on this analysis, we consider all of the low-level errors as detected if a high-level error encapsulating the low-level errors is identified. Thus in the situations described above, we would count three true positives even though only one high-level error was explicitly identified.

We face a similar counting problem if the error detection method does not identify any of the three errors that occurred in the behavior trace. In this case, we must de-

termine how many false negatives should be counted. One possibility is to count three false negatives, but this is also unappealing because the compound error encapsulates information in the the two other simple errors. Instead, following similar arguments as with the counting of true positives, we count a single false negative since all three errors would be considered identified by the single compound error.

Our approach to counting elements in the confusion matrix can be generalized by the following rules:

- If only simple errors are detected, count each as a true or false positive depending on whether they correspond to actual errors in the novice's behavior.
- If compound errors are detected, count true positives for the compound error and all of the simple errors that comprise the compound error. If the compound error is detected incorrectly the count it as a false positive.
- If a primary error is detected, count true positives for the primary error and all of the secondary errors that are causally linked to it.
- False negatives are counted first by finding the set of errors that were not identified by the error detection method. The count is then incremented by the minimum additional number of errors that it would have been necessary to identify for the true positive count to be maximized.

One of the effects of our counting method is that the number of errors reported (RP) by an error detection method may no longer be the sum of $FP + TP$. Instead, one piece of information in the report can map to multiple true positives, thus $TP \geq RP - FP$. To illustrate differences of brevity between reports that identify similar numbers of true positives, we introduce a final performance metric, *Report Density*.

$$Report\ Density = \frac{TP}{RP}$$

In the following chapters we examine two methods of comparing two actors' behavior. In Chapter 5, we describe a relatively straightforward approach based on sequences extracted from behavior traces. Then, in Chapter 6 we describe our novel comparison method based on a concise representation of the behavior encoded in one or more traces.

Chapter 5

Sequential Approach to Comparing Behavior

5.1 A Basic Comparison Method

In the previous chapter, we argued that when dealing with humans, it is likely that we will need to limit representations of behavior to sequences that contain goals and actions, i.e., behavior traces. Based on this, we can define a simple approach to comparing two actors' behavior in the following manner:

1. Acquire behavior traces from the human domain expert and software agent.
2. Extract relevant features from the behavior traces.
3. Compare the extracted features between human and software agent.
4. Report differences.

These four steps create a framework for a number of distinct comparison approaches, and can be considered a generalization of the techniques we first presented in [57]. We refer to all of the techniques derived from this simple framework as sequential comparison methods.

5.1.1 Acquire

The first step in this comparison method is to acquire examples of behavior. As described earlier, this can be done relatively simply by capturing a human's or software agent's interactions with a simulator. During this first phase, we acquire two sets of behavior traces, H and A that represent instances of the human's behavior and the software agent's behavior respectively.

In complex environments, we expect that both the human domain expert and the software agent will be able to perform their task in a variety of different ways. This variability may be a response to differences in the environment (e.g., because it is non-deterministic) or it may arise because the actor has opportunities to choose among more than one equally acceptable goal or action at different points in their reasoning process. As the complexity of the domain increases and task performance becomes increasingly variable, a larger number of behavior traces will be needed to get an accurate overview of the actor's behavior. Exactly how many traces will be required is a difficult question to answer, and we will examine it in more detail in Section 5.2. For now, we will limit ourselves to outlining desirable traits of the acquired behavior:

- In general, more traces are better than fewer traces.
- Each new trace should be distinct from any of the previously collected traces.
- Traces should be representative of the entire breadth of the actor's behavior. One approach to help ensure that this occurs is to to generate traces randomly (independently from one another) presuming that is viable.

5.1.2 Extract

After behavior traces have been acquired from the human and the software agent, the second step begins. Although we have argued that behavior traces are an appropriate level of abstraction for most human-level tasks, there may be situations in which some information contained in the trace is not relevant for detecting meaningful differences. To reduce the number of spurious deviations (false positives), we extract features from the behavior traces that are directly relevant for a meaningful behavior comparison. This process yields two new sets H^* and A^* each containing sequences that are generalization of those in H and A. We consider two main approaches for constructing these generalized sequences.

Symbol Modification

The most obvious way to transform H and A is to remove those features from the state or features from goal and action structures that we know are irrelevant for detecting errors. For example, consider the partial behavior trace, H_j illustrated in Figure 5.1

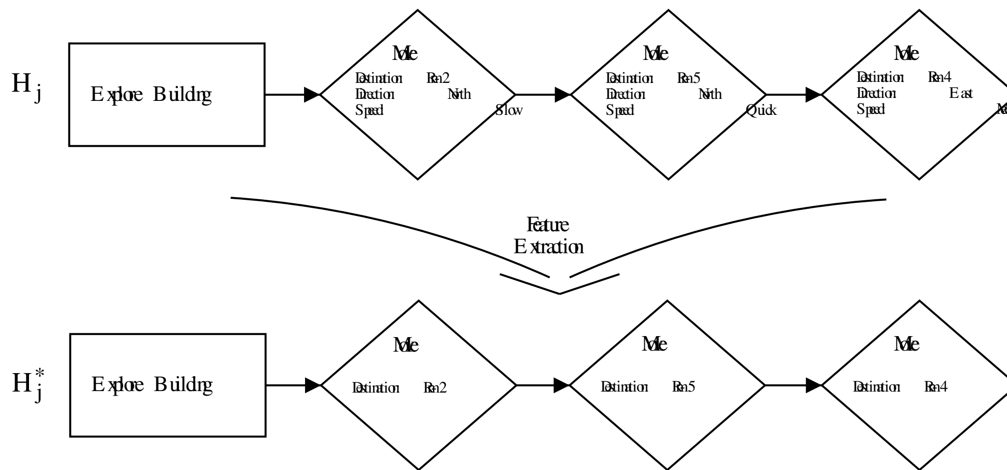

Figure 5.1: A behavior trace that may profit from feature removal

that represents a relatively unstructured activity—exploring an abandoned building. Although this may be an important part of some human-level tasks such as search and rescue, it may also be the case that the correctness of this behavior is based solely on whether the actor is moving through new areas of the building. In this situation many details as how exactly the actor moves (e.g., direction and speed features augmenting the action symbols) may be irrelevant and theoretically could be removed when H^* and A^* and formed, as in the sequence illustrated in the lower potion of the figure, H_j^*. The difficult aspect of this procedure is to correctly determine the largest set of features that can be removed without impacting the ability to detect errors; exactly where this information comes from is somewhat unclear, but we will examine a few possibilities below.

An alternative method of transforming H and A is based on more more general equivalence classes of goal or actions symbols. This approach goes a step beyond simply removing features from the state or from action or goal symbols by making it possible to describe arbitrary equivalence relationships among different symbols. Sequences in H^* and A^* would be comprised of canonical forms of these equivalence classes, thus reducing the detectable differences between two sequences. This technique may be most useful if different equivalence classes can be used at different points in the behavior sequence. For example, if there were multiple movement actions, such as `walk`, `run`, `crawl`, they could all be grouped into a single equivalence class when the most recent goal symbol was `explore-building`. Moreover, this method would also allow us to treat these symbols as distinct during other parts of the task.

The freedom to influence how the sequences in H^* and A^* are formed from sequences in H and A is a double-edged sword. Although it allows the ability to reduce the number of spurious errors reported by the comparisons method, in practice, it may also be relatively difficult to implement correctly. This difficulty stems from the fact that it may be unclear how to obtain information about the set of features that should be abstracted from H and A, and moreover, it may be unclear how reliable this information is.

In order to identify which features are salient for detecting errors, one may be tempted to look for answers in the agent's knowledge base. However, this is clearly an unacceptable strategy if we are attempting to validate this same knowledge. Alternatively, one might attempt to obtain this information directly from the domain expert herself. This, however, simply amounts to re-performing some of the previous knowledge acquisition, a process that we have already noted is both error prone and tedious. A third approach would be to statistically classify what parameters are salient for identifying correct expert behavior. This method, however, would only work in the presence of a large corpus of observations about expert behavior, something that may not be reasonable to require.

From the analysis above, it is unclear whether it would be possible to determine what features of the behavior trace are important to detect errors without negatively impacting one of our primary goals (e.g., minimizing human effort). In the cases where this information can be successfully obtained, however, the exact form of the knowledge may play an additional role in determining properties of the overall error detection method.

Sampling Method

Another way to affect how the sequences H^* and A^* are formed from H and A is to change the way in which the original sequences are sampled (see Figure 5.2). Recall that the behavior traces contain state, goal and action tuples, (s, G, a). Instead of ensuring that there is a one to one correlation between tuples in H^* and tuples in H, we could sample the original behavior trace at intermittent points and thus create generalizations of the original. There are a wide variety of ways to sample the original trace, but we consider three particularly natural methods.

Time based sampling can construct sequences that closely resemble those stored in the original behavior trace. Here, however, a new element in the sequence is formed after some specified period of task-performance time has elapsed. At its finest resolution,

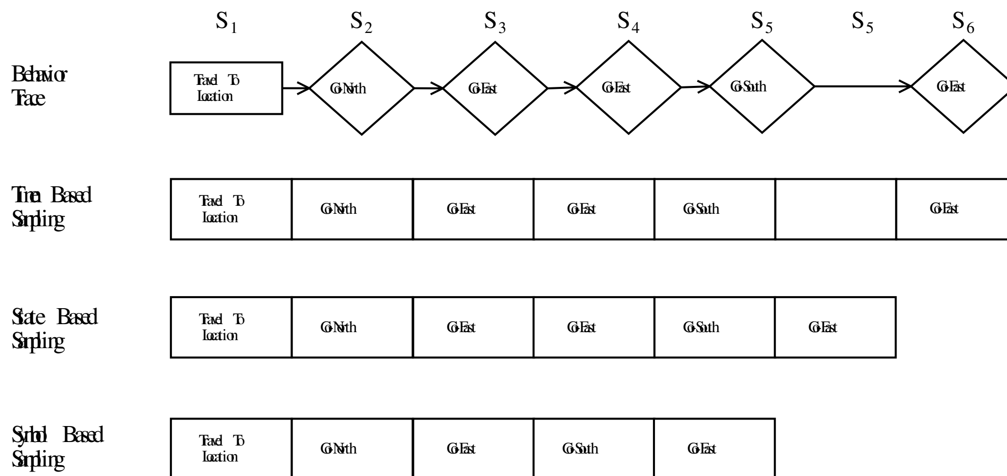

Figure 5.2: Effects of different sampling methods

time based sequencing will exactly duplicate information stored in the behavior trace. Although it would be possible to sample the behavior trace at a lower rate, it is unclear what advantage this would provide. At a low sampling rate, actions or goals that occurred in between the sampling points would be lost. Although we would still be able to detect deviations between two sequences, spurious deviations could easily be introduced if the streams became misaligned by a single time step. Moreover, because symbols between sampling points would be lost, it would be unclear when these spurious errors were introduced by the sampling method.

Similar to time based sampling, state based sampling also retains most of the information that is already stored in the behavior traces. Using this approach, a new element in the sequence is formed whenever there are two adjacent elements in the behavior trace such that $(s, G, a)_i \neq (s, G, a)_{i+1}$. In contrast to time based sampling, state based sampling will collapse periods when the behavior trace record the flow of events at a higher resolution than that at which the environment operates. As a result, state based sampling can be useful for representing relatively event-driven behavior. For example, an observation of an agent waiting for a phone call may contain a series of `wait` or `idle` symbols representing the agent's inaction before the phone rings. Once the phone does ring, a new goal such as `pick-up-phone` or an action such as `move-right-hand` would be produced. Creating such a sequence with state based sampling hides differences between agents that wait 30 seconds to pick up the phone and agents that wait only 10 seconds

to pick up the phone. In many situations, this would be a much more desirable sequence to use in a comparison than the original behavior trace.

The final approach, symbol based sampling, results in the shortest sequences. Using this method, strings of identical goal or actions symbols are collapsed regardless of whether they change the world state. More formally, a new element in H^* or A^* is formed only when there are two elements in the behavior trace, $(s, G, a)_i, (s, G, a)_{i+1}$, such that $a_i \neq a_{i+1}$ or $G_i \neq G_{i+1}$.

Clearly, one's choice of how the behavior trace should be sampled has a significant impact on what differences will be detected during the comparison phase. Using a method that extracts information from the behavior traces less frequently, such as symbol based sampling, results in shorter sequences and thus fewer opportunities to detect differences between two sequences. This can reduce the number of spurious errors, or false positives, that are detected, but if it is used indiscriminately, it can also result in some errors being overlooked (i.e., false negatives). It is also important to note the importance of sampling behavior from both actors in an identical manner. For example, a meaningful comparison could not be performed if the human's behavior traces were sampled using a time based approach, while the software agent's behavior traces were sampled using a state based approach.

Analysis

We can interpret each of the sequences in H and A as strings whose length corresponds to the number of elements in the individual sequence h_i and a_i. Characters in these strings are selected from two (potentially very large) alphabets Λ_H and Λ_A respectively. The two approaches to making generalizations of H and A discussed above modify the properties of these strings differently. The first method, symbol modification, effectively reduces the size of the alphabets so that $|\Lambda_H^*| \leq |\Lambda_H|$ and $|\Lambda_A^*| \leq |\Lambda_A|$. The second method, which varies the way the original sequences are sampled, leaves the alphabets intact but reduces the length of the strings.

5.1.3 Compare

In the third step, we compare each sequence $a \in A^*$ to the contents of H^*. A successful comparisons hinges on three important processes: appropriately aligning the

symbols in two sequences; identifying a particular $h \in H^*$ against which we will compare a; and identifying actual differences between a and h.

Alignment of two sequences, regardless of whether they come from the sets A^* or H^*, is simplified by the fact that these sequences all describe an actor's behavior while performing a complete task. As a result, natural alignment points exist at the beginning and end of each sequence. The beginning of the sequence identifies the point at which the actor obtains the task goals and begins the reasoning necessary to perform it. The end of the sequence corresponds to the point of task completion, that is, the point at which the task goals have been met. The fact that the ends of each sequence serve as fixed points means that alignment is limited to the interiors of the sequences and the window in which symbols can be shifted is limited by the differences in sequence lengths. Often this also means that the differences between the lengths of different sequences is relatively limited because we are not considering situations in which one sequences corresponds to only a small portion of the task behavior in another sequences. This greatly simplifies the alignment problem and reduces the complexity of the comparison.

Our method compares pairs of sequences as opposed to entire sets, so for each $a \in A^*$, we must identify an appropriate sequence $h \in H^*$ upon which to base our comparison; this is the second part of the comparison step. The method used to select these sequence pairs, (a, h) will play a significant role in what and how many errors are detected. In order to determine the best approach, we consider the extreme case, in which the software agent exhibits no errors, and the general case in which the quality of the agent's behavior is unknown.

Recall that a wide range of behaviors may be contained in both A^* and H^*. In the ideal case, in which the software agent's behavior has no errors, $A^* \subseteq H^*$. In order to determine that this relationship holds, the sequence pairs, (a, h) used in the comparison must be selected such that $a = h$. This can be done relatively easily by comparing each $a \in A^*$ to each of the $h \in H^*$ and examining each of these sequence pairs for equality.

In the more general case, when the quality of the agent's behavior is unknown, we want to use the same basic approach to choosing the sequence pairs (a, h) that will be used to determine how the agent's behavior corresponds to the expert's behavior. Again, we need to examine each $h \in H^*$ in the hopes of identifying one such that $a = h$. In the absence of an exact match, we want to identify h such that it is the closest approximate

match to a. We refer to this as the closest concept to a in H^*, or $CC(a, H^*)$.

Choosing the closest concept can have an impact on the number of differences identified in a, however in some cases, efficiency concerns may prohibit a complete comparison between a and each $h \in H^*$. In these cases, a heuristic may be used to identify the closest concept, so that the complete comparison can be performed on only a single pair of behaviors $(a, CC(a, H^*))$. One efficient heuristic is the generalized Hamming distance between a and each $h \in H^*$. This function chooses $CC(a, H^*)$ such that there are minimal discrepancies between the contents of its slots and the slots in a. Moreover, the cost of identifying the closest concept in this fashion is limited to $\sum_{h \in H^*} \min(|a|, |h|)$ where $|h|$ and $|a|$ are the number of slots in the sequences h and a respectively.

Once we've determined the closest concept, we can perform the comparison between the two behavior sequences. Since we are now comparing only a single set of sequences, it is reasonable to make use of a more computationally expensive method than we would use to identify $CC(a, H^*)$. One useful method that remains relatively inexpensive is minimum edit distance [29, 55]. Assuming that the sequences in H^* and A^* have been generated using the same sampling technique, we can use the minimum edit distance to identify how a can be modified to recreate the expert's behavior ($CC(a, H^*)$) by deleting, changing or inserting the minimal number of slots. These edit operations map directly onto individual errors of omission, commission and intrusion. As each sequence in A^* is examined, errors are detected and recorded for the final phase of the comparison.

5.1.4 Report

The last step in the comparison is to report the overall differences between the software agent's behavior and the human expert's behavior. Many differences that were observed during the comparisons, such as performing `use-left-hand` instead of `use-right-hand`, may occur repeatedly throughout the agent's behavior. These identical differences are all likely to stem from a single error, such as a different preferences about which hand to use. By consolidating identical errors, we can reduce the amount of redundant information in the error report, making it easier to examine.

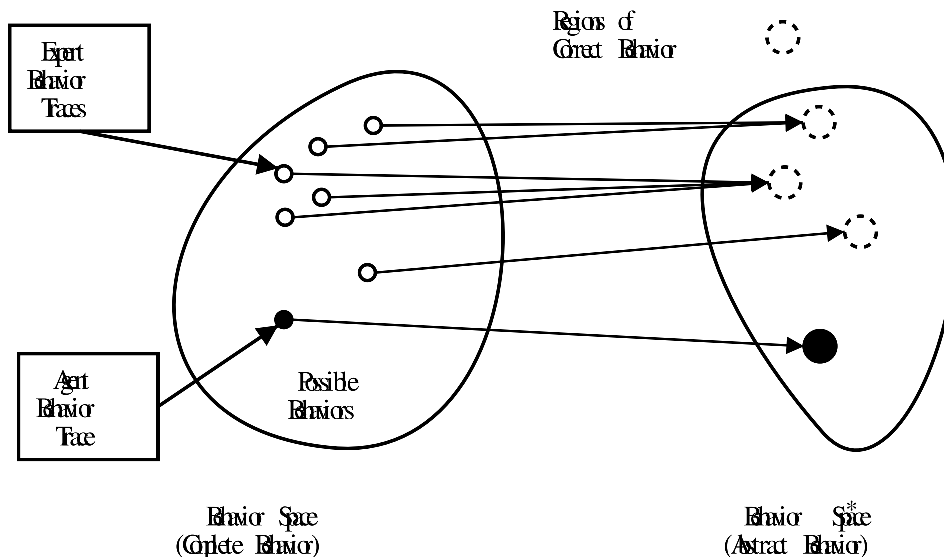

Figure 5.3: Mapping between complete and generalized behavior spaces

5.2 Analysis

Before we begin an empirical investigation of the sequential approach to behavior comparison, we want to examine some of its properties in more detail. In particular, we are interested in understanding more completely how the behavior space is used by the comparison approach. This involves examining how the partitions between correct and incorrect behavior are identified as well as examining how the generalization procedure (the extract phase of the process) changes the effective size of the behavior space. Secondly, we would like to gain a better understanding about how many behavior traces might be required to adequately describe an actor's behavior.

5.2.1 Properties of the Behavior Space

To examine how the sequential approach to behavior comparison distinguishes between regions of correct and incorrect behavior, we begin with an overview of the comparison process. At a basic level, the agent's behavior a is compared to each observation of the expert's behavior. If a matches one observation of expert behavior, no errors are detected. This means that the regions of behavior space corresponding to correct behavior can be viewed as a number of points occupying a portion of the abstract behavior space (B^*-Space).

Although the regions of correct behavior are points in B*-Space, they correspond to potentially larger regions in the unabstracted, or complete, behavior space (B-Space). The size of each of these regions depends on how much information in H was generalized to form H^*. When all of the features in the behavior trace are relevant for identifying errors, there is no difference between the sequences in H and those in H^*. In this case, there is a one-to-one correspondence between points on B*-Space and points in B-Space. Thus, even in the complete behavior space, only $|H^*|$ points will be considered within the region of correct behavior. As the amount of information relevant for identifying errors decreases, sequences in H^* become increasingly general compared to those in H. As a result, points in B*-Space correspond to one or more points in B-Space. Similarly, members of H^* now correspond to one or more of the behavior traces in H. Figure 5.3 illustrates this situation in which some generalization occurs between B-Space and B*-Space; the area covered by the B-Space and B*-Space as well as the size of "points" in these regions illustrate the effects of generalization.

Given that the sequential approach defines regions of correct behavior by enumerating points in abstracted behavior space, we would like to know how effective our generalization procedures are at limiting the size of this space. To perform this analysis, we begin by examining the size of the complete behavior space, $|B|$. Let Λ_H be the set from which (non-null) symbols are selected to compose the sequence representing the expert's behavior. Assume that the number of symbols in the sequences of expert behavior is limited to the finite range (H_l, H_u). We can construct the space of possibly correct behaviors, i.e., the complete behavior space, by forming all sequences within the bounds of H_l and H_u containing symbols selected from Λ_H.

$$|B| = \sum_{i=H_l}^{H_u} |\Lambda_H|^i \tag{5.1}$$

$$|\Lambda_H|^{H_u} \leq |B| \leq |\Lambda_H + 1|^{H_u}$$

The size of the abstracted behavior space, $|B^*|$ is defined similarly and is affected by the type of generalization operations used during the extract phase of the comparison process. For example, using symbol modification to remove a factor of r features when transforming H into H^* where $0 < r < 1$, the size of the abstract space is given by

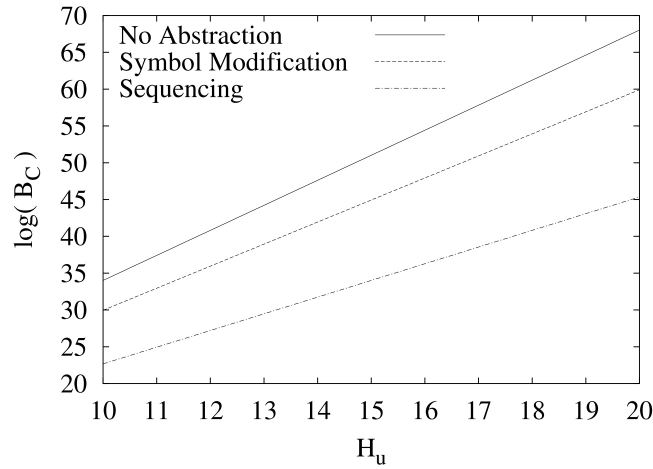

Figure 5.4: Effects of two extraction methods

$$|B^*| \leq \sum_{i=H_l}^{H_u} ((1-r)|\Lambda_H|)^i \tag{5.2}$$

Alternatively, if sampling was used to generalize the behavior traces, reducing the length of each sequence by a factor of r where $0 < r < 1$, then

$$|B^*| \leq \sum_{i=H_l}^{H_u} |\Lambda_H|^{(1-r)i} \tag{5.3}$$

Figure 5.4 illustrates the relative power of these generalizations for a modest sized alphabet containing 30 elements where $r = 2/3$. For identical values of r, the improvement made by sampling is more dramatic. However, appropriate ranges for r depend heavily on properties of the domain and are also likely to vary significantly based on the generalization approach that is used (either symbol modification or sampling). Moreover, it is important to realize that in either case, the abstracted behavior space still grows exponentially. Since the boundary of correct behavior is specified by enumerating elements from the abstract behavior space, this is will lead to some difficulties, especially in complex domains, as we will show in the following sections.

5.2.2 Sample Complexity

As we noted previously, one problem with using behavior traces as a basis for performing our comparison is that as the complexity of the domain increases, it is likely that

more and more examples will need to be acquired. Sample complexity is a measurement used to establish computational bounds on the process of learning particular concept classes. We can use this same measurement here to determine how many behavior traces are required to establish the boundary between correct and incorrect behavior.

In computational learning theory, the following equation, where $|Y|$ indicates the size of the hypothesis space, can be used to identify a bound on the sample complexity for consistent learners [12, 34].

$$m \geq \frac{1}{\epsilon}\left(\ln(|Y|) + \ln\left(\frac{1}{\delta}\right)\right) \tag{5.4}$$

A learning algorithm is consistent if it outputs hypotheses that perfectly fit its training data. For these learners, m represents the lower bound on the number of training examples that is required to learn a hypothesis that is probably (with probability $1-\delta$), approximately (within error ϵ) correct (i.e., the actual target concept).

When we examine a behavior comparison method, we begin by mapping the terminology we have been using throughout this paper onto the definition above. The comparison method's perception of the boundary between correct and incorrect behavior is the hypothesis being learned. The actual boundary, on the other hand, is the target concept (i.e., the hypothesis we hope to learn). Finally, behavior traces acquired from the expert fill the role of training examples. Although the sequential comparison does not have a sophisticated learning algorithm by any means, its ability to memorize (i.e., to add new generalized behavior traces to H^*) fills this same role. Because the memorization procedure retains all distinct training examples, this procedure is completely consistent and the equation 5.4 is applicable.

To identify the sample complexity of the sequential approach to behavior comparison, we must identify the size of the hypothesis space in which our learning procedure will operate. Since the hypothesis is the set of sequences in H^*, the size of the hypothesis space is equal to the number of all possible sets of points in B^*. From equations 5.3 and 5.2, we know that the number of points in B^* is less than (or equal to) the number of points in the possible behavior space, but regardless of the generalization method we use the number of points still grows exponentially as a function of H_u. For simplicity, we use the lower bound from equation 5.3 as a lower bound on the number of sequences in H^*. This means that the size of the hypothesis space, i.e., the number of different ways

in which H^* *could* be specified, is given by: $|Y| \geq 2^{|\Lambda_H|^{(1-r)H_u}}$. Substituting this back into Equation 5.4.

$$m \geq \frac{1}{\epsilon}\left(|\Lambda_H|^{(1-r)H_u}\ln(2) + \ln\left(\frac{1}{\delta}\right)\right)$$

We find that the sample complexity grows exponentially as a function of the sequence length. This means that this method is computationally intractable, and unless the length of the behavior traces and the overall complexity of behavior is closely controlled, there is little hope that this method will identify the actual boundary between correct and incorrect behavior. Short of the necessary number of expert behavior traces, H^* will lack examples of correct behavior, thus increasing false positives and decreasing specificity, however false negatives should be unaffected. In complex environments, where we may not be able to gather the appropriate number of examples of expert behavior, the information in the error summaries is likely to become dominated by these false positives. In these situations, the summaries may become effectively useless.

5.3 Empirical Evaluation

To examine the empirical efficacy of the sequential approach to behavior comparison, we evaluated its performance in two domains. The first of these was a relatively simple object retrieval environment that was constructed as part of our research testbed. The second domain, the MOUT environment, represents a significant step up in complexity. In addition, it is important to note that the MOUT environment was developed completely independently from our project, thus removing the possibility that the environment was tailored to specific properties of the behavior comparison method.

Both environments are interfaced to the Soar agent architecture [23], and in all of the following experiments agents were implemented in Soar. In each domain, we examined two distinct and orthogonal methods of building sequences. The first of these methods built sequences that contained only actions and action parameters using change based sampling. The second method built similar sequences but extracted only goal and goal parameter symbols. In both approaches, state information that was not connected to the goal or action symbols was ignored. The idea behind this decision was that it would allow us to construct relatively general representations of behavior while still maintaining state

information implicitly in the ordering of elements in each sequence. We consider the goal and actions sequences orthogonal because they contain distinct symbol sets. As a result, it is reasonable to consider using them in combination to identify errors in either actions or goals. In the object retrieval domain, we examine the performance of both individual approaches as well as the combined method.

5.3.1 The Object Retrieval Environment

The first environment we examine is a purpose-built simulated object retrieval domain. The world consists of a map representing a restricted city-space, with buildings and roads. The interiors of buildings consist of rooms and hallways each of which may contain various typed objects. The agent's basic task is to retrieve an object of some prespecified class (e.g., a bag or a book) from within a specific target location (i.e., a particular building). Task execution thus requires three phases: traveling to the target area, searching for the desired object, and exiting the target area.

We began the experiments by constructing an initial expert-level agent that solves the task in an extremely constrained manner. That is, across different attempts, the agent will complete the task using identical behavior so long as it is provided identical initial states and so long as the environment responds identically to its behavior. Given the nature of the task, we make the assumption that the agent has access to a road map describing the overall layout of the city-space, but that it does not know the exact layout of a building's interior. Thus a complete path to the desired object cannot be identified without some exploring.

Our initial expert-level agent begins the task by leveraging knowledge about the city's layout to plan an optimal route from its initial location to the target area. Once this route is established, the agent examines the nearby area for potential methods of transportation. Using internal preferences over different modes of transportation, the expert-level agent selects one and then travels along its planned route to the target location. Because the layout of individual buildings is not known a-priori, the agent must explore the building to locate the desired object. Once the object has been identified and retrieved, the agent must exit the target area to complete the task.

Given this initial expert-level agent, we performed a number of modifications on its knowledge-base by randomly removing rules that determine preferences between com-

Experiment	Agent Traces	Expert Traces
1	a_0	h_0
2	a_1	h_0
3	a_2	h_0
4	$a_{0,1,2}$	h_0

Table 5.1: Individual experiments in family one

peting goals and actions. The results of the modifications are a set of new agents that complete the task successfully in the traditional sense (i.e., they reach the same final state), but which have increased flexibility in terms of the sequence of goals and actions they use to achieve that final state.

Each family of experiments begins by selecting two agents, e and n, such that n is a modified (more flexible) version of e. We designate e as the expert, and n as the novice. Because n is more flexible than e, it will behave in certain ways that are not consistent with expert behavior—these are errors.

After the expert and novice have been selected, they are individually incorporated into a simulation so their behavior can be observed. We then gather between 10 and 15 behavior traces of the actors performing their task, ensuring that no two behavior traces are identical[1]. These traces form the sets BT_E and BT_N for the expert and novice respectively. Finally, as described in Chapter 4, each behavior trace in BT_N is examined manually to catalog the errors it contains. This catalog serves as the gold standard against which we will measure performance.

The captured behavior traces are then split into a number of subsets: $A_i \subset BT_N$ and $H_i \subset BT_E$. The A_i and H_i are subsets of the behavior traces that were collected from the novice-level agent and expert-level agents respectively. These sets correspond to the sets A and H described earlier in the chapter. A single experiment is performed by examining each comparison method's performance on a pair of these subsets (A_i and H_j). A *family* of experiments contains the experiments that compare all A_i to all H_j for a particular novice/expert pair. Thus comparing four expert/novice pairs results in four experiment families although the total number of individual experiments may be much larger.

[1] In some cases, the expert and novice do not perform the task in 10 distinct ways. In these situations, all distinct methods of task performance are gathered.

Table 5.1 illustrates the individual experiments within the first experiment family of the object retrieval testbed. In this family, the expert-level agent has maximally constrained behavior, which is covered by an individual behavior trace (h_0). The novice-level agent, on the other hand, has somewhat less constrained behavior that is covered by three behavior traces (a_0, a_1, a_2). The four experiments in this family examine the effects of comparing different subsets of these expert-level and novice-level behavior traces. When more flexible agents are used, more behavior traces are collected to represent the agents' behavior and the number of experiments performed within each family increases accordingly. By constructing experiment families in the way, we are able to examine the impact of different observational data more easily.

Results

For each experiment, we compare the sequences in A_i^* to those in H_j^*. This yields a summary of similarities and differences between the novice-level agent's behavior and the expert-level agent's behavior. Following the protocol laid forth in Section 4.6, we evaluated the performance of behavior comparisons using goal sequences and action sequences to represent the agents' behavior. For this presentation, we have averaged these values for each experiment family. The following four figures illustrate the performance of the approaches within the object-retrieval domain, as well as the net effect of using these methods in combination.

Figure 5.5 illustrate the sensitivity across each experiment family in the object retrieval domain and illustrates three phenomena. First, it shows the expected effect that in general the action sequence will detect different errors than the goal sequence. This is illustrated by the fact that sensitivity profile across experiment families (in these cases largely driven by the number of correctly detected errors, or true positives) is different for the action sequence and the goal sequence. This lends credence to the idea of using these comparison methods in combination as is illustrated by the relatively high sensitivity of the combined approach.

Second, the figure shows a somewhat higher performance for the comparison approach that relies on action sequences as opposed to goal sequences. Although some of this effect may be attributable to the particular distribution of errors in our test set, this effect can also be attributed at least in part to a basic property of this and many other domains.

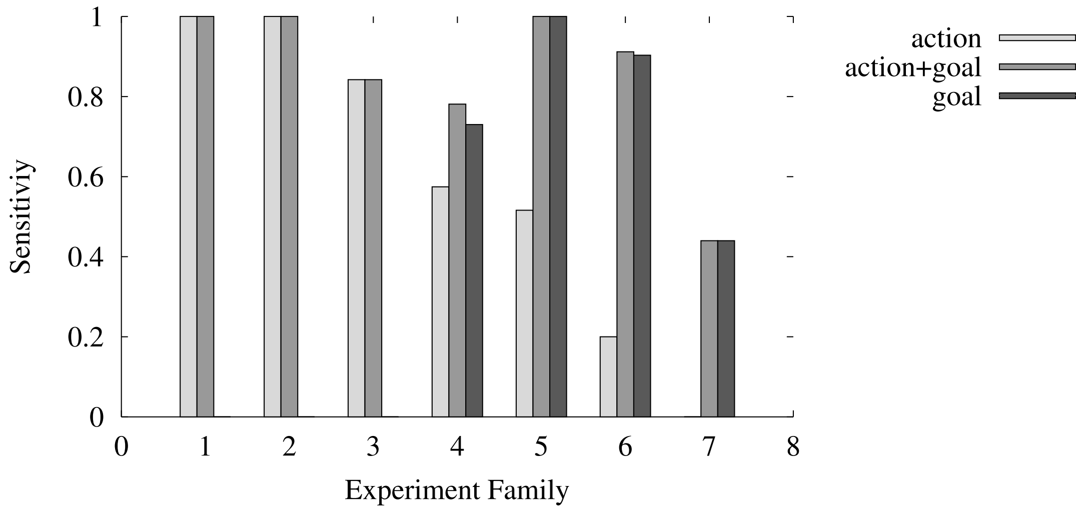

Figure 5.5: Sequential approach: sensitivity in the object retrieval domain

In the object retrieval domain, the agent's knowledge is organized hierarchically into methods for accomplishing different aspects of the overall task. Toward the top of this hierarchy are abstract goals. Moving toward the bottom, these goals are progressively refined until primitive actions are reached at the bottom most level. In the object retrieval domain, errors that originate at higher levels of this hierarchy (i.e., via the selection of an inappropriate goal) are likely to result in a cascade of lower, action level errors at some point in the future. Thus using action sequences to compare behavior should allow some true positives to be detected even when errors originate in the goals. In contrast, this effect is not as prominent in the reverse sense. As we mentioned previously, in the object retrieval domain there are many ways to correctly perform the task in the classical sense (when the agent's final state is the only criteria for success). This applies not only to the task, but also to individual goals and subgoals. As a result, an error that originates at the level of primitive actions (e.g., an error of commission in which one action is substituted for a similar but incorrect action) is less likely to affect the sequence of goals the agent pursues during later stages of task performance. As a result, many errors that originate at the primitive action level will be impossible to detect by examining a sequence of goal symbols alone.

Finally, it is important to note that although the action and goal sequences do perform significantly better in combination than alone, even used jointly, they do not achieve

maximum sensitivity across all experiment families. Intuitively, it seems sequences of goals and actions should be extremely sensitive, and should mainly be at risk of low specificity (due to false positives). False negatives can arise for two reasons.

First, these sequential methods do not identify compound errors very effectively. Recall that each novice trace in A^* is compared against an element in H^* in a single pass, and mismatched members in the two sequences are stored as errors. This means, that during examination, only low level errors are identified. Compound errors are considered only during the summarization process after all the sequences in A^* have been examined. It is likely that these errors would be detected more effectively if they were sought earlier in the comparison process, while the representations of the agent's behavior were still available to be examined.

The second manner in which false negatives can be introduced is due to incorrect alignment of the expert's and novice's behavior sequences. Recall that the beginning and end of the agent's behavior sequences served as fixed alignment points, and for these experiments a minimum edit distance algorithm is used to align the interior elements in the sequences. Although it is possible that there will be a single alignment that produces the minimal edit distance, as the sequences get increasingly long and increasingly complicated, it becomes more and more likely that multiple minimal cost alignments exist. Since the comparison method has no way to prefer one alignment over another, one of these alignments is selected at random. This choice has no effect on the number of errors that are reported, but it may mean the difference between reporting information that can be used to identify an error that actually occurred (true positives) and reporting unproductive information that allows the real errors to go undiagnosed (false negatives).

The standard counterpart to sensitivity is specificity, illustrated in Figure 5.6. Across all experiment families specificity is extremely high, nearing its maximum value in all cases. However, for the sequential approach to error detection, this really only tells part of the story. To understand the actual impact of this value, we must examine this calculation in more detail. Recall that specificity is the ratio: $TN/(TN+FP)$. But what exactly is a true negative? When describing the evaluation process in Section 4.6, we described how the comparison process can be viewed as a set of classification problems in which different aspects of a behavior representation (in this case symbol sequences) are evaluated to determine their similarities and differences. Each one of these classifications

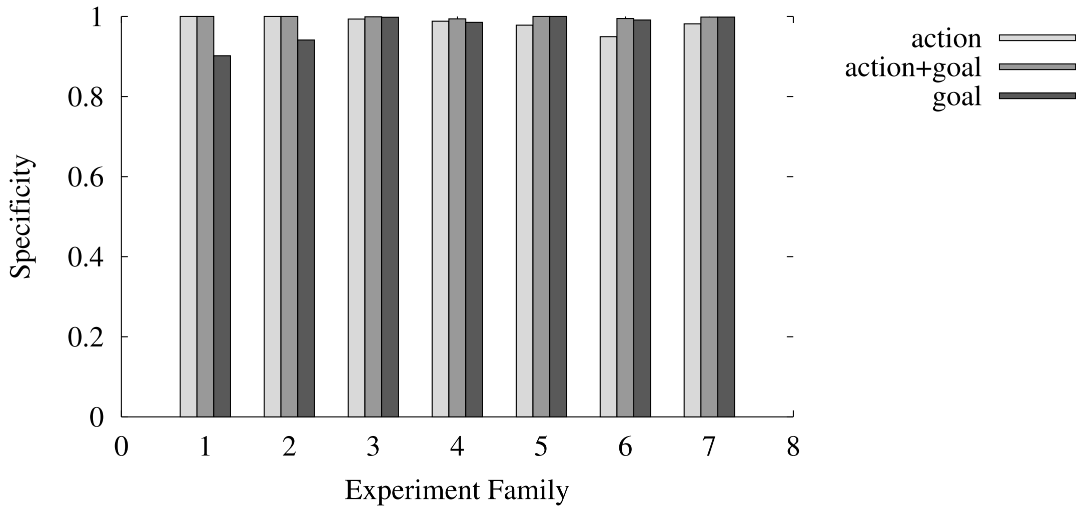

Figure 5.6: Sequential approach: specificity in the object retrieval domain

that correctly indicates that the aspects of the model being compared are in fact the same is a true negative. In the sequential approach, there is at least one such classification event for each element in each sequence of A^*. This number can grow very large. In this relatively simple domain, the number of elements in an individual action sequence is around 14. So, if an agent's behavior is described by 10 of these sequences, there are 140 classifications to be made. Because the agents being examined are already close enough in their behavior to achieve the same final states after performing their tasks, much of what they do is often very similar. Thus, we should expect that the vast majority of these classifications will not indicate that an error has occurred. Moreover, when so many classifications are performed and the prior probability of an error is so low, maintaining a high specificity is crucial. If specificity were to drop to .5, for example, it would indicate that for every true negative, there was one false positive. False positives would then number on the order of 100, and completely dominate the error report. The next measurement we examine, report density, takes this problem into account.

Figure 5.7 illustrates report density for each of the behavior comparison methods. Essentially this value (described in Section 4.6) measures the average information content in each piece of information provided by the behavior comparison's report. Whereas high sensitivity indicates a low number of false negatives, high report density indicates a relatively low prevalence of false positives. In many ways, report density is a more

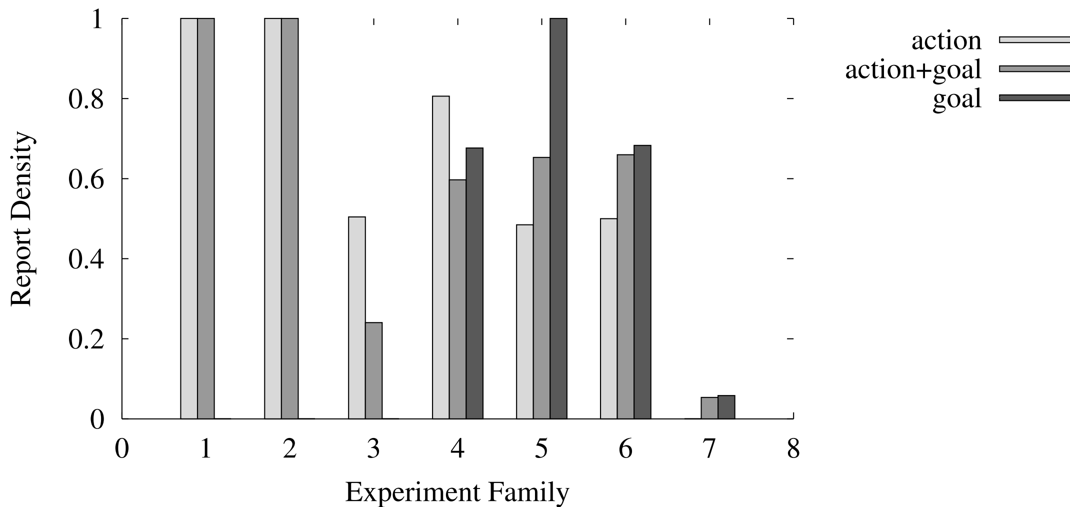

Figure 5.7: Sequential approach: report density in the object retrieval domain

useful way to measure the deleterious effects of false positives than specificity. In all of the behavior comparison method we are examining, it is expected that the error report provided by these methods will be interpreted by the domain expert or knowledge engineer to make a final diagnosis and identify problems in the agent's knowledge base. The quality of a report, however, is unaffected by the number of classifications in which errors should not, and are not detected (i.e., true negatives). Instead, the report presents information that is either a true positive or a false positive. The role of the false positives then is not so much to reduce the ability to discriminate aspects of behavior that are correct, but rather to contaminate the otherwise useful information about how the agents are in fact different. Like sensitivity and specificity, a higher report density is more valuable. Unlike these other measurements, however, report density can increase above 1.0 if compound or primary errors are concisely represented in the error report. This allows us to explicitly measure (and favor) methods that generate concise, high quality, error reports. A report density value of 1.0 indicates that there is, on average one piece of information in the error report for every error that can be identified using this information. Figure 5.7 indicates a reasonable report density across the majority of experiment families with values typically at or above .5 for the combined approach using both goal and action sequences[2]. This means that on average, a domain expert or

[2]Across the board, false positives were limited to an average 2 or less in all experiment families. Thus low report densities are the result of few or no true positives being detected.

knowledge engineer analyzing the reports will need to weed out one red herring for each useful piece of information in the report—performance that may not be ideal, but is nonetheless respectable.

Our experiments in the object retrieval domain, indicate that although the sequential methods are by no means perfect, they are able to identify a number of errors in these simple agents, especially when used in combination. Moreover, although there is some overhead associated with identifying what information in the reports is actually useful, reports are rarely dominated by noise. In the next subsection, we will examine how these properties scale in more complicated environments.

5.3.2 The MOUT Environment

In contrast to the object retrieval domain, the MOUT Environment represents a significant increase in overall complexity. The environment is built on top of Unreal, a commercial 3-D video game. Equally important to the added complexity of this environment, is the fact that MOUT was built independently from our research into behavior comparison techniques. Thus, it provides an important reference point for judging the overall effectiveness of our techniques.

Our basic experimental approach in the MOUT environment remains very similar to that which was used in the object retrieval domain, however some important differences do exist. Instead of building our own expert-level agent and using it as a basis for novice-level agents, we began with the default MOUT agent and made successive modifications to its knowledge base to generate a set of partially flawed, novice-level agents. The MOUT agent's knowledge base is very large (near 750 rules) compared to the expert-level object retrieval agent which contained only 78 rules. Because of the scale of the agent's knowledge base, it is unreasonable to expect that simply performing random modifications to the rules would result in behavior changes that manifest reasonably frequently while simultaneously ensuring that the changes were not severe enough to completely prevent the agent from performing its task. As a result, we made a series of random modifications, including rule additions and deletions as well as condition modification to a restricted subset of the agent's knowledge. This knowledge describes the agent's behavior with respect to middle-level goals, and, as desired, modifications to these rules resulted in behavior deviations at both the level of goals and the level of

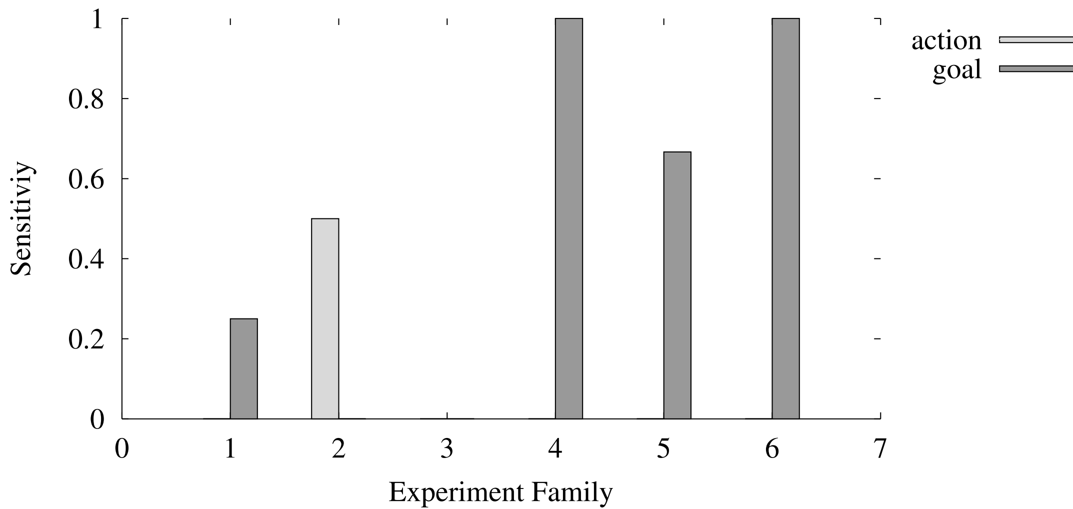

Figure 5.8: Sequential approach: sensitivity in the MOUT domain

primitive actions.

Figure 5.8 illustrates the sensitivity of comparison methods using sequences of goals and actions to represent the agent's behavior. By using the goal sequence to represent the agent's behavior, it was possible to identify all of the errors in the agent's behavior in two of the test scenarios. However, as we will see, even if these methods approached perfect sensitivity across all experiment families, they would nonetheless have little use in an environment of this complexity.

Figures 5.9 and 5.10 complete the story. The report density for these experiments peaks at an amazingly low peak value near 0.015. Moreover, the number of false positives soars to a high of nearly 600. In essence, the majority of reports in this set of experiments would be next to useless in a real world setting where a domain expert or knowledge engineer needed to allocate time to discern true positives from false positives. Interpreting these reports was exceedingly difficult even in the experimental setting with a priori knowledge about what errors occurred in the behavior and could therefore be represented in the reports. At the heart of the problem are two issues that reinforce earlier concerns about the scalability of this approach.

The first issue that makes the sequential approach to behavior comparison relatively useless in complex environments is its relatively weak ability to generalize behavior traces. Even using sequences that ignored many of the state features, and are therefore much

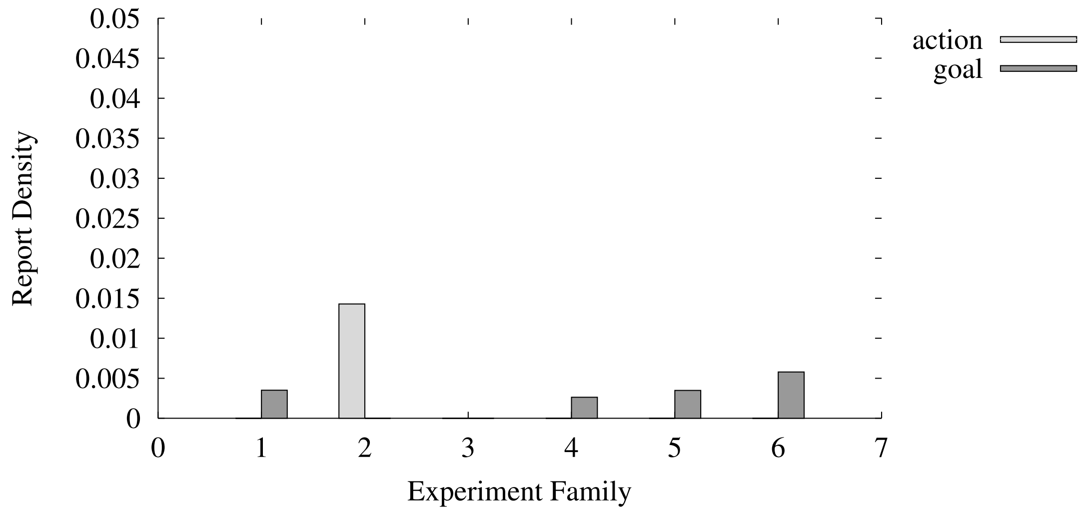

Figure 5.9: Sequential approach: report density in the MOUT domain

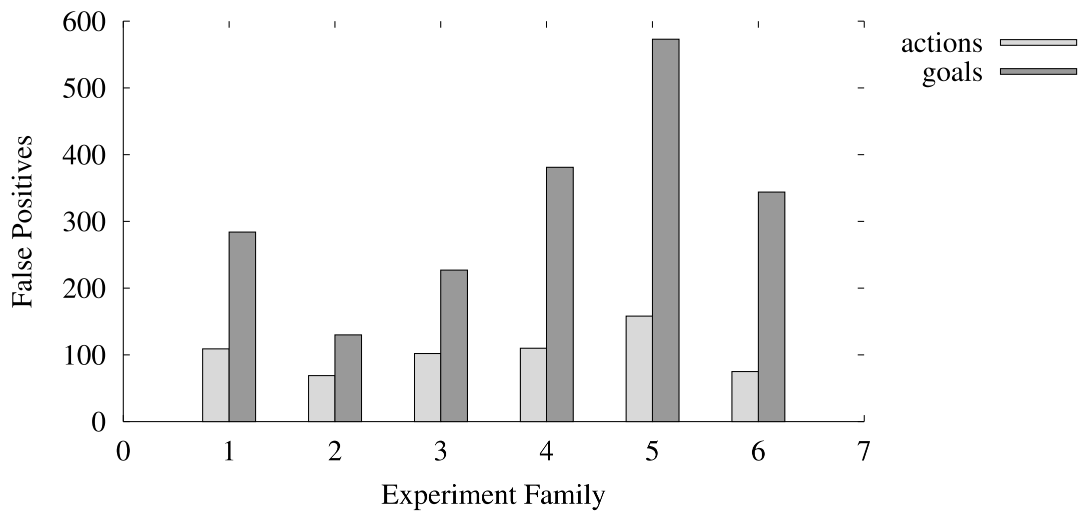

Figure 5.10: Sequential approach: false positives in the MOUT domain

	Cost				Human-Level Agent Support				
Method	P1	P2	P3	P4	P5	P6	P7	P8	P9
Sequential Approach	√	√	—	—	√	√	√	**Some**	√
Manual Validation	—	√	√	—	√	√	√	√	√

Table 5.2: Properties of the sequential approach

less likely to generate false positives than more complete sequential representations of behavior, error reports were still dominated by spurious information. Although it may have been possible to further reduce the amount of information in the goal and action sequences, for example, by defining equivalence classes between different actions or goals, we found it very difficult to determine how these classes should be formed so that fewer false positives would be detected while still maintaining minimal false negatives.

The second difficulty stems from the variability in the lengths of the behavior representations. In the MOUT environment, behavior representations often contained more than three hundred symbols. Worse, the lengths of sequences could vary dramatically for a particular agent's behavior, often by a factor of three. As we described earlier, this makes the alignment problem much more difficult, and exposes one of the key weaknesses of the sequential method.

5.4 Satisfying Properties of an Ideal Approach

In Chapter 3, we identified nine properties that would be met by an ideal error detection system. Here, we examine how well the sequential approach to error detection measures up to this ideal.

The first four properties of an ideal error detection method describe the overall effort required to use or implement the approach. We began our investigation with the premise of minimizing human effort, and to a certain extent, the sequential approach we examined in this chapter do meet these criteria. In particular, because the sequential approaches perform the behavior comparison automatically, they minimize human oversight (**P1**) by relying on it only during at the end of the process, to interpret the information in the report. In addition, the sequential approach does minimize supportive engineering (**P2**) because domain knowledge is not strictly required (although if it is available, it can and

should be leveraged to help reduce the size of H^* and A^*). However, because size of the hypothesis space grows exponentially, in complex domains, a large amount of example behavior will be required to get acceptable figures for sensitivity and report density, thus violating **P3**. Furthermore, because of this same property, it will be difficult to ensure that an adequate number of observations of the novice's behavior have been collected to appropriately reflect the breadth of the novice's behavior. This fact will make it difficult to determine an appropriate time to end the validation phase, thus violating **P4**.

The remaining five properties of an ideal error detection method describe its suitability for complex, human-level domains. In this respect, the sequential approach fulfills all of our original requirements. Because it is a completely general approach to detecting differences in agents' behavior, it can easily adapt to diverse environments (**P5**). By ensuring that humans retain some degree of participation in the validation process (by analyzing and interpreting the information in the error report), this method is able to leverage tacit expert knowledge (**P6**) without requiring nearly the same degree of human oversight as a fully manual approach to validation. Although the sequential approach to error detection does not directly support imprecise specifications of correctness by changing the way it evaluates different types of deviations in the novice's behavior, it does partially meet this requirement by allowing the final diagnosis to be performed by the domain expert or knowledge engineer by analyzing the error summary (**P8**). Finally, because this approach can utilize any information in the behavior trace, and because it requires only information in these traces, the sequential approach can be applied to different aspects of the agents' reasoning process as we illustrated in the goal-based and action-based comparisons (**P7**). Furthermore this same trait allows the sequential method to be applied to both human and agent behavior (**P9**). These properties are summarized and compared against the manual approach to validation in Table 5.2.

Chapter 6

Behavior Bounding

Although we initially had relatively high expectations for the sequential approach to behavior comparison, our evaluation in Chapter 5 indicates that these expectations were ill-founded. In retrospect, we believe that in terms of efficacy, the sequential approach's biggest liability is its poor method of generalizing behavior traces. Even when sampling and symbol modification can both be applied to the problem, which we found it hard to do in practice, the hypothesis space still grows too quickly. To find a better approach, we must start by re-examining the representation of behavior itself.

In the sequence-based approach to comparison, an actor's behavior is represented as a set of instances. Although these instances may themselves be generalizations of actual observations, the fact that this method cannot construct a concise representation of aggregate behavior is the cause of significant problems. In hopes of finding a better representation, we return to the model-based approaches in which a single (but not necessarily concise) model describes the entire scope of an actor's behavior.

Recall from Chapter 3 that previous model-based approaches (i.e., model-based diagnosis and the CLIPS-R approach to knowledge refinement) fail to meet a number of the properties of an ideal error detection system. Three properties that we want to pay particular attention to when designing our new behavior comparison system are:

P2: Minimize supportive engineering Because the models are typically constructed by hand, they require additional engineering effort before any validation/comparison can begin. If the results of the testing are negative (i.e., no differences between the two test subjects can be found) then the effort used to construct these models is effectively wasted.

P4: Provide a grounded stopping criterion Like most other methods for behavior

comparison or validation, none of the model-based approaches we have examined address the problem of when the evaluation should be terminated. By leaving this question open-ended, the agent's design team is put in the difficult position of deciding when the validation phase should end without a theoretically grounded basis.

P8: Support imprecise specifications of correctness None of the automated comparison methods we examined deals well with the inherent imprecision that occurs when specifying parameters of correct human-level behavior. In our review of related work, the only methods that did satisfy this property were those that lacked any degree of autonomy and were driven completely by human oversight. Although we expect that by allowing the human domain expert to review a report of potential deviations, we will retain some abilities to deal with imprecise specifications, we believe that a comparison method would benefit from integrating this characteristic into the detection framework, instead of relying completely on human oversight.

A foundation for building a model that meets all these criteria can be found within the structure of the agent's knowledge itself. In goal-oriented tasks, behavior is often specified by describing how high-level goals decompose into lower-level goals and finally into primitive actions. Our approach, called behavior bounding [58], exploits the structural relationships between goals, sub-goals and primitive actions to describe an actor's behavior in a concise format. In the remaining sections of this chapter, we will examine this model in more detail and present a theoretical and empirical analysis of the approach as we did in Chapter 5.

6.1 The Hierarchical Model

The advantages of behavior bounding stem from its representation of behavior. Behavior bounding is based on the structural relationships implicitly and explicitly encoded within the actor's knowledge base. It is also inspired by the hierarchical representations used in AND/OR trees, HTN planning [7] and GOMS modeling [19] to describe the variety of ways in which particular tasks can be accomplished.

Behavior bounding's hierarchical behavior representation (HBR) is illustrated in Figure 6.1:A. The hierarchy is an AND/OR tree with binary temporal constraints repre-

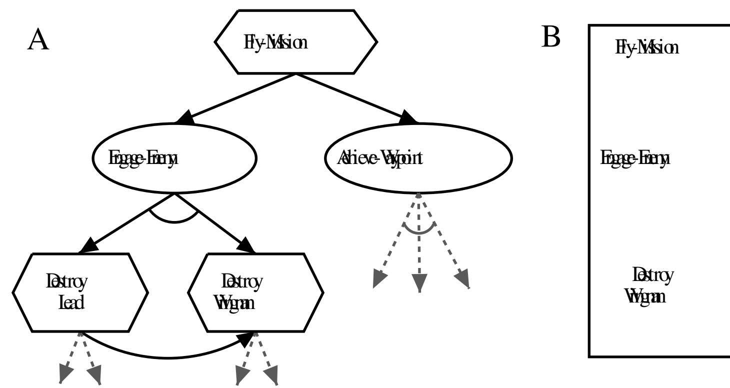

Figure 6.1: Hierarchical behavior representation & goal stack

```
Set goal:  Fly-Mission
Set goal parameter:  (altitude 30000)
Set goal parameter:  (patrol-speed 800)
    Set goal:  Achieve-Waypoint
    Set goal parameter:  (waypoint AZ-12)
    Set goal parameter:  (threat-level low)
    Set goal parameter:  (ETA 10 minutes)
        Action:  (set-altitude 30000)
        Action:  (compute-heading AZ-12)
        Action:  (set-heading)
    Set goal:  Return-to-Base
    ...
```

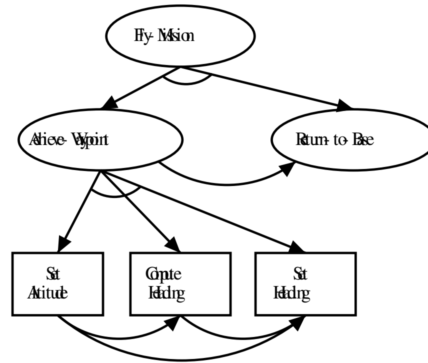

Figure 6.2: Constructing the hierarchical behavior representation from a behavior trace

senting the relationships between the actor's goals and actions. In this representation, internal nodes correspond to goals and leaves correspond to primitive actions. A node's children indicates the set of sub-goals or primitive actions that are relevant to accomplishing the specified goal. For example, in Figure 6.1:A, the sub-goals `Destroy-Lead` and `Destroy-Wingman` are relevant for completing their parent goal, `Engage-Enemy`. The manner in which sub-goals should be used to achieve their parent goal is encoded by the parent's node-type constraint (AND vs OR) and the ordering constraints between sub-goals. In Figure 6.1:A, AND and OR nodes are represented with ovals and hexagons respectively. Binary temporal constraints are represented with arrows between siblings. Thus, the hierarchy specifies that `Engage-Enemy` may be correctly accomplished by first accomplishing `Destroy-Lead` and then accomplishing `Destroy-Wingman`.

Each behavior trace can be represented as a constrained hierarchical structure similar to the one illustrated in Figure 6.1. Typically, however, a single behavior trace results

in a maximally constrained hierarchy. For example, consider the partial behavior trace on the left-hand side of Figure 6.2. The trace is processed in a single pass, reading from beginning to end. As new goals and actions are encountered, nodes are added to the hierarchical representation. The hierarchy of goals the actor is currently pursuing is indicated in this behavior trace by each line's level of indentation. In this example, the goal stack is generated incrementally beginning with the selection of a top level goal that is decomposed into a lower level goal before again begin decomposed into a series of primitive actions. A goal is considered completed when it is no longer a member of the actor's goal stack. For example, in Figure 6.2, the goal `Achieve-Waypoint` is completed when the actor commits to performing a new goal at the same level of abstraction (i.e., when the goal `Return-to-Base` is selected). As the behavior trace is read, the requirements for goal completion are tracked including the subgoals necessary to accomplish the current goal and their ordering as well as the parameters of the goal and its respective subgoals. These requirements are represented as the descendants in the hierarchy and the constraints between them. The algorithm used to construct the hierarchical behavior representation is presented as pseudo-code in Figure 6.3.

6.1.1 The CREATE-HIERARCHY Algorithm

The CREATE-HIERARCHY algorithm can be used to construct representations of both instance and aggregate behavior. By calling it with a single behavior trace, B, and $H \leftarrow$NIL, a hierarchical representation of a single behavior trace is generated in the method mentioned above. By iteratively calling CREATE-HIERARCHY with different behavior traces, the representation of behavior, H, is augmented and generalized until it covers all of the example traces. This algorithm can be executed in $O(lN^2)$ time where l is the length of the behavior trace and N is the number of nodes in the goal hierarchy. Details of the execution and its cost can be found in Appendix A.

The model constructed with the CREATE-HIERARCHY algorithm and illustrated in Figure 6.3 is clearly a much less complex representation of behavior than the agent's underlying knowledge base. Behavior bounding abstracts away internal data-structures the agent may use in problem solving that cannot be represented by the constraints in the hierarchy. Some basic tasks, such as depth first search, often rely on internal data structures (e.g., an open list) to discriminate between alternative behaviors (e.g.,

```
CREATE-HIERARCHY(B, H)
 1  W ← empty tree
 2  lastStk ← NIL  // previous goal/action stack
 3  for each (s, G, a) in B
 4  do
 5      for i = 0 to length[lastStk]
 6      do
 7          if GOAL-COMPLETED(lastStk[i])
 8            then h_g ← FIND-NODE(H, lastStk[i])
 9                 if h_g = NIL
10                   then
11                        ADD-SUBTREE(H, PARENT(lastStk[i]), lastStk[i])
12                   else
13                        GENERALIZE(H, h_g, W, lastStk[i])
14      for each g_i in [G, a]
15      do
16          p_g ← PARENT(g_i)
17          w_g ← FIND-NODE(W, p_g, g_i)
18          if w_g = NIL
19            then
20                 w_g ← ADD-NODE(W, p_g, g_i)
21                 CONSTRAIN-CHILDREN(W, p_g)
22            else
23                 if OUT-OF-ORDER(W, p_g, w_g)
24                   then UPDATE-CONSTRAINTS(W, p_g, w_g)
25                 GENERALIZE(w_g, g_i)
26      lastStk ← [G, a]
27  return H
```

Figure 6.3: The CREATE-HIERARCHY algorithm

`Expand` *X* v.s. `Expand` *Y*). As a result, these tasks would be difficult to perform correctly using information stored in the HBR alone unless data captured by these internal data structures was pushed into the goal hierarchy itself, for example by explicitly identifying distinct `Expand` subgoals for various items in the search tree.

The representational limitations of the HBR begs us to ask: if the agent's behavior can be represented using such a simple structure, why was it not programmed in this representation to begin with? The hypothesis here is not that this representation is sufficient to *completely* capture the agent's behavior. Most human-level agents do rely on intermediate data-structures that are not available through the environment or through the structure of the goal hierarchy. However, our hypothesis is that the representation provided by behavior bounding is sufficient to identify a large class of possible errors in agent behavior without sacrificing efficiency. Moreover, we believe that behavior bounding can also help identify potential problem spots in the agent's knowledge (e.g., a specific goal) even if an exact error cannot be identified.

6.1.2 Leveraging Assumptions

In contrast to the behavior representations used for the sequential comparison methods described in Chapter 5, the HBR makes two strong assumptions about the organization of the actors' knowledge and the effects this will have on their behavior. These assumptions increase the efficiency and efficacy of error detection for certain types of human-level agents.

The first assumption used by the behavior bounding approach is that the actor's goals are organized hierarchically, with more abstract goals placed toward the top of the tree. We also assume that at any point in the problem solving process, the actor pursues a set of goals belonging to different levels in the hierarchy. This set, referred to as the goal stack, corresponds to a path in the hierarchy beginning at the top node and descending to the most concrete sub-goal that is currently being pursued by the actor. Figure 6.1:B illustrates a possible goal stack maintained by the actor whose behavior is represented in Figure 6.1:A.

The second assumption leveraged by behavior bounding relates to the independence of goals. Temporal constraints can only be formed between sibling nodes, and And/Or classification determines which of a node's children must be performed for a particular

task. This makes it easy to constrain the ways in which a particular goal is achieved, but difficult to represent constraints between arbitrary parts of the hierarchy. Although this may cause problems with some agent implementations, this property has significant benefits. Most importantly, it decreases the number of observations that are required. Consider a task that requires completing two goals, each of which could be fulfilled in four distinct ways. A sequential representation that makes no assumptions about goal independence (such as the one described in Chapter 5) would construct sequences of length 4 and require sixteen distinct observations to cover the acceptable behavior space where as behavior bounding would only require four observations to cover this same behavior space. This significant impact on efficiency is the direct result of leveraging the assumption about how goals are likely to add regular structure to an actor's behavior.

6.2 Identifying Errors

In Chapters 4 and 5, we viewed a behavior comparison method as an algorithm that divides the space of possible behaviors into two distinct regions: those that are perceived to be correct (i.e., consistent with the expert's behavior) and those that are perceived to be incorrect (i.e., inconsistent with the expert's behavior). The sequential comparison method described in Chapter 5 performs this division by memorizing examples of expert behavior and drawing a black and white distinction between consistent and inconsistent behaviors when it examines observations of the novice's behavior. In contrast, the constrained hierarchical representation used by behavior bounding breaks the space of possible behaviors into more refined regions.

To demonstrate this process, we begin by noting that the constrained hierarchical representation allows us to impose order on the space of behavior representations. In particular, we can define an ordering from specific to general over the behavior hierarchies, by starting with a maximally constrained hierarchy (at the top) and iteratively removing constraints until none remain. Constructing a representation of an expert's behavior performs this same generalization, but most often stops before all constraints have been removed. Figure 6.4, in which each node represents a behavior hierarchy, illustrates this ordering.

Once we have created a representation for the expert's behavior using the algorithm described in Section 6.1, we can identify the node it occupies in this ordered space (call

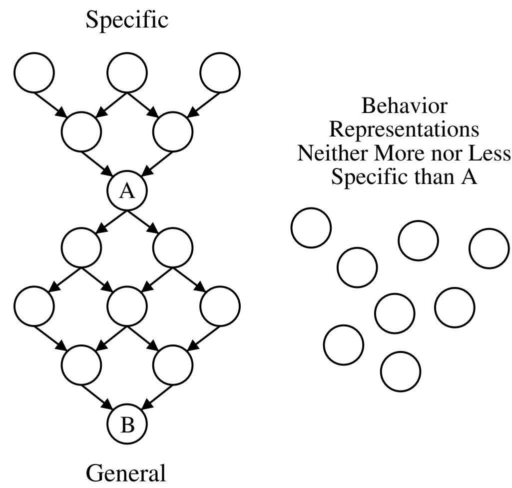

Figure 6.4: Imposing order on the behavior space

this node `A` in Figure 6.4). This node (the upper boundary node) allows us to easily determine if the agent's behavior is likely to be correct. By definition, correct behavior must be consistent with expert behavior. An agent whose behavior representation is a specialization of the expert's (i.e., lies above `A` in the generalization lattice) exhibits behavior that is consistent with the expert's and is therefore likely to be correct. As in the sequential approach to behavior comparison, the upper boundary node allows us to partition the behavior space into two regions: correct and incorrect.

A second partition is formed by the node representing the completely unconstrained version of the expert's goal hierarchy. This node is illustrated at the bottom of Figure 6.4 (labeled `B`). It contains the most basic specification for what may constitute acceptable agent behavior and as a result could be used to identify behavior representations that are known to be incorrect (because the agent's behavior hierarchy is topologically inconsistent with the expert's behavior hierarchy). Such representations would have a goal decomposition structure that was inconsistent with (i.e., contained different parent/child relationships than) this lower boundary (nodes in the right side of Figure 6.4).

Together, the upper and lower boundaries create three regions in the behavior space. Nodes that are a specialization of the expert's behavior (the upper boundary node) correspond to behavior that is very likely to be correct. Nodes that are not a specialization of the unconstrained version of the expert's goal hierarchy (the lower boundary node) correspond to behaviors that are known to be incorrect. The region in between the upper

and lower boundary nodes corresponds to behavior that is likely to be incorrect, but perhaps not with as high a probability[1] as the region that is not a specialization of the lower boundary node. We draw the distinction between these regions (known-incorrect and likely-incorrect) based on the assumption that examining additional examples of expert behavior is more likely to result in generalizations to the upper boundary node as opposed to structural changes in the goal decompositions themselves. In Section 6.6 we examine an alternative method of specifying the HBR that corresponds to the lower boundary node which further justifies the distinction between these regions in the behavior space.

Similar to the S-Set and G-Set in Mitchell's version-spaces [33], the upper and lower boundary nodes allow behavior bounding to delineate three regions inside the behavior space without enumerating their contents. Once these boundaries have been established, a hierarchical representation of the novice agent's behavior is generated. Performing a node-by-node comparison of the novice's behavior model with the boundary nodes, it is a simple matter to determine whether this representation is a specialization of either the upper or lower boundary nodes. This analysis, which can clearly be done in polynomial time with respect to the number of nodes in the hierarchy determines the region within which the novice's behavior lies. In this same process, the comparison identifies aspects of the novice's behavior model that are consistent and inconsistent with the models represented by the boundary nodes. These inconsistencies form the basis of behavior bounding's error report, and can be displayed in either a standard text format or visually using a GUI.

6.3 Analysis

In this section, as we did in Section 5.2 for the sequential method of behavior comparison, we examine some of behavior bounding's properties in greater detail. In particular, we want to examine the computational requirements for learning the HBR that corresponds to the upper boundary node. As part of this task, we will examine the sample complexity of the hypothesis space, thus allowing us to make some comparisons between behavior bounding's generalization procedure and the generalization procedure used in

[1]Here, we assume that it is easier to ensure that the HBR reflects the correct agent topology than it is to ensure constraints on the upper boundary node's HBR are adequately generalized. In practice, the degree to which this assumption holds will depend on properties of the agent and how the HBR corresponding to the lower boundary node was formed (See 6.6 for an alternative method).

the sequential approach. Secondly, we will examine the ability of the representation corresponding to the lower boundary node to act as a filter for behavior that is known to be inconsistent with examples of the expert. Finally, before evaluating behavior bounding's performance in our testbed domains, we will determine the degree to which it has met the three requirements outlined at the start of this chapter.

6.3.1 Learnability

In this section, we examine two aspects of behavior bounding's hierarchical representation: the effort required to create and maintain it, and its ability to represent behavior efficiently. Both of these requirements are addressed by the overall learnability of the representation. That is, if the representation can be learned from observations (as we have suggested), then it requires human effort only to initiate the learning process. If the learning procedure is efficient, and the data structure's growth is limited, we can further say that the hierarchy represents behavior efficiently.

The learning procedure for constructing the HBR extracts goal stacks and actions from a behavior trace, forming a hierarchical structure such as the one illustrated in the previous section. After processing the first behavior trace, the hierarchy contains the maximum number of constraints (i.e., And/Or constraints on the goals and binary temporal constraints between siblings) that are consistent with the behavior in the trace. So, if each goal in the hierarchy is pursued only once while performing the task, all internal node-types are And (maximally constrained) and all sibling internal nodes are totally ordered (again, maximally constrained).[2] Upon examining subsequent behavior traces, the hierarchy is generalized in such a way that it remains maximally constrained with respect to all of the behavior traces it has processed.

In Section 6.1 we illustrated the Create-Hierarchy algorithm with time complexity $O(lN^2)$, where l is the length of the behavior trace and N is the size of the goal hierarchy. In most cases, it is reasonable to assume that one property of expert-quality behavior is that the completion of the task will occur within a number of steps proportional to N. When this assumption holds, we can say that this algorithm is bounded by $O(N^3)$ with respect to the size of the input (i.e., the length of the behavior trace). Because this complexity is a low-order polynomial of N, the hierarchy is efficient when encoding an

[2] The leaves, representing primitive actions, will only be totally ordered if each action was used only a single time to achieve its parent goal.

instance of behavior.

We can also classify the sample complexity of our hierarchical representation. We can think of our representation as an ordered tuple $P = (p_0, p_1, \ldots, p_N)$ where each p_i refers to one of the N nodes in the hierarchy. Each p_i is itself a tuple containing the type of the node referred to by p_i (either AND or OR), as well as a list $L = (l_0, l_1, \ldots, l_b)$ (where b is the branching factor of the hierarchy) that describes the ordering constraints between the node referred to by p_i and its siblings. In the degenerate case, the length of the list L would be at most length N, but because ordering constraints only occur between siblings, the length of L would typically be much shorter. The size of this hypothesis space is bounded by 2^{N+N^2}. Using Haussler's equation [12], the number of training examples, m, required to learn the appropriate behavior representation is bounded by:

$$m \geq \frac{1}{\epsilon}\left((N^2 + N)\ln(2) + \ln\left(\frac{1}{\delta}\right)\right) \tag{6.1}$$

This indicates that the required sample size is polynomial with respect to the number of goals in the hierarchy (N). This, together with the fact that the time required to incorporate a new behavior trace into the learned HBR is also polynomial in N, shows that our representation is PAC-Learnable. This means that the HBR efficiently represents aggregate behavior as well an individual instance of behavior, thus meeting our fourth requirement.

6.3.2 Efficacy of the Lower Boundary Node

The efficacy of behavior bounding is addressed by two components. The first of these is how well the unconstrained hierarchical representation (the lower boundary) identifies behavior that is known to be incorrect. The second component is how well the expert's representation (the upper boundary) distinguishes between potentially correct and incorrect behavior.

At first glance, it is not obvious how much behavior can be filtered by the lower boundary. However, its effectiveness as a filter is quite surprising. Consider an unconstrained behavior representation with branching factor b and depth d. Without loss of generality, assume that the nodes are uniquely labeled. For simplicity, also assume that at any level in this hierarchy, the actor completes its current goal before starting the next goal. Then, we could define an actor's behavior as a sequence of symbols chosen

from the lowest level of the unconstrained hierarchy. For behavior sequences of length b^d, in which no symbol is repeated, the number of possible sequences that are consistent with the goal decomposition of the unconstrained hierarchy is given by the recurrence $S(b,d) = b!S(b,d-1)^b$. In closed form, $S(b,d) = b!^r | r = \sum_{j=0}^{d-1} b^j$. In contrast, there are $b^d!$ sequences in which the symbols may be placed without necessarily conforming to the unconstrained hierarchy. For a hierarchical structure of depth 4 and branching factor 2, only 1 in approximately $6.4 \cdot 10^8$ of the length 16 sequences are consistent with the goal decomposition specified by the unconstrained hierarchy. This illustrates the potential power of the lower behavior boundary to discriminate between behavior that is potentially correct and the large collection of behavior that is inconsistent with the expert's goal decomposition structure, and thus known to be incorrect.

6.3.3 Expressivity Limitations

One of the most significant aspects of behavior bounding is that the size of the behavior representation is fixed by the size of the agent's goal hierarchy. This property ensures that the representation will not grow unbounded and more importantly ensures that the size of the hypothesis space remains small enough to be PAC-Learnable. However, as we alluded earlier, this property does not come without some loss of expressivity. In particular, behavior bounding does not build multiple nodes to represent distinct goal decompositions. That is, the CREATE-HIERARCHY algorithm identifies nodes to add or generalize based solely on the name of the goal and its relative location in the hierarchy. This creates two potential problems for performing behavior comparison between certain goal based agents.

Duplication Errors

First, behavior bounding's HBR cannot explicitly represent duplication errors. Consider the illustration in Figure 6.5. On the left side of the figure, we see an example of a behavior trace in which the agent accomplishes goal `P` by pursuing first goal `C` and then goal `B`. This results in the obvious hierarchical behavior representation that covers exactly this one behavior trace. However, consider the HBR built from the behavior trace on the right side of the figure. In this case, the agent accomplishes `P` by pursuing `C` twice before beginning `B`. The traces are identical except for this additional instance of

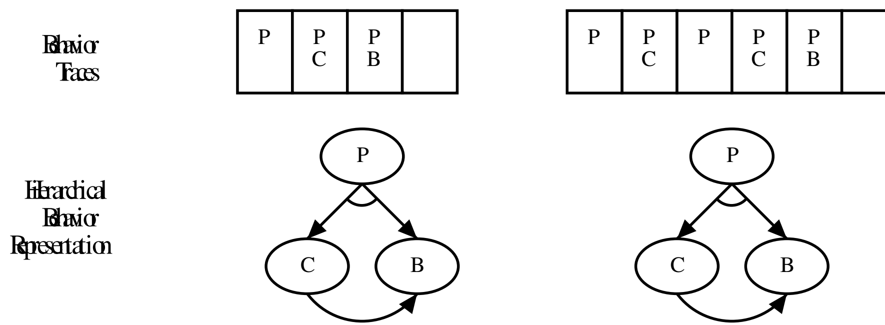

Figure 6.5: Duplications cannot be directly identified by the HBR

`C`. But because the goals in the agent's behavior trace are mapped into the hierarchy by name and lineage, the HBR contains only a single node corresponding to `C`. This node is generalized so that its parameters cover the parameters observed on both instances of `C`, but because there is only one node, there is no explicit method of determining how many times `C` was completed before moving on to `B`.

In this sense, behavior bounding's HBR does not explicitly represent properties necessary to identify duplication errors. However, because the node corresponding to a particular goal is generalized each time the goal is pursued, duplication errors may be implicitly represented by differences in the generality of nodes in the novice's behavior representation and in the expert's behavior representation. If ordering constraints are generalized, this implicit representation can be fairly obvious (in Figure 6.5, this does not occur). Unfortunately, we have not yet identified any straightforward way in which to capture the relationships needed to identify duplication errors explicitly while simultaneously maintaining the overall learnability of the representation. In some situations, however, it might be possible to identify potential duplication errors simply by tracking upper and lower bounds on the number of times that a particular goal (or action) was pursued in order to accomplish its parent. Although this won't fix the underlying representational problem, it may be sufficient to identify errors in many situations, especially if the knowledge engineers take this constraint into account at design time.

XOr **Type Decompositions**

The second limitation of behavior bounding's HBR is that it is not ideally suited to represent some types of goal decompositions. Consider, for example, a goal P that

is best defined as an XOR goal. That is, P is correctly accomplished by performing exactly one of its two descendants C_1 or C_2. Although the behavior traces will indicate this relationship (i.e., P is accomplished on some occasions using C_1, on other occasions using C_2, but never using C_1 and C_2), P will be represented in the HBR as an OR node with two children C_1 and C_2. This obfuscates the fact that these subgoals may be in conflict with one another.

Although this representational issue may be addressed at least in part by deliberately engineering the agent's goal decomposition, it might be addressed more effectively by changing the type constraint of the node itself. Instead of differentiating simply between AND nodes and OR nodes, we might expand the representation by adding an explicit XOR type. With this representation, the generality ordering would be given by the following relations (where $>_g$ identifies the *more general than* relation): OR $>_g$ AND; OR $>_g$ XOR; AND $\not>_g$ XOR; XOR $\not>_g$ AND.

Initially, goals would be classified as either AND or XOR depending on whether a single sub-goal or multiple sub-goals were required to accomplish the goal being classified. On successive attempts this designation would change to OR if a different set of subgoals was pursued (in the case of an AND node), or if more than one subgoal was required to achieve the goal being classified (in the case of an XOR node). Of course, such a classification scheme would still require some amount of deliberation during the design of the agent's goal decomposition structure to avoid the situations in which `P` decomposed into either `A` or (`B`, `C`).

6.3.4 Comparison to other Model-Based Approaches

At the beginning of this chapter, we identified three properties that we wanted to ensure our novel approach met: minimizing engineering; providing a stopping criterion; and supporting imprecise specifications of behavior. Here we examine to what extent behavior bounding satisfies these properties.

As we have shown, behavior bounding's hierarchical representation can be learned efficiently from examples of expert behavior. Assuming that a simulator is already available so that behavior traces can be captured, this method requires no additional engineering. As a result, we can safely say that this method does in fact minimize supportive engineering as desired. However, it is also possible that behavior bounding's performance can be

improved if some engineering is allowed. In particular, as we discuss in Section 6.6.1, hand crafting the hierarchical behavior representation that corresponds to the lower boundary node may be useful to circumvent some of behavior bounding's current shortcomings.

The second property we must examine, is behavior bounding's ability to provide an indication of when the validation phase should be considered completed—a grounded stopping criterion. Recall from Chapter 3 that none of the methods we examined satisfied this property. Behavior bounding, however, does provide this ability to a significant degree. Because the hierarchical behavior representation is PAC-Learnable, with a given size sample of expert behavior, we can learn a representation that is probably, approximately correct. This of course assumes two important criteria, first that the actual behavior representation is expressible in the language provided by behavior bounding, and secondly that it is possible to gather independent and randomly drawn examples of expert behavior. Given than these constraints are met for both the expert and the novice actors, we can identify a theoretically grounded point to stop the comparison (i.e., when the required number of behavior traces to appropriately represent each actor's behavior have been gathered).

Finally, behavior bounding supports imprecise specifications of correctness in two distinct ways. The most obvious of these is that behavior bounding does not actually require any such specification. Instead, the specification is inferred by examining observations of the expert's behavior. Thus, the difficulty of articulating parameters of correct behavior is relatively unimportant. All that is needed is for an expert to be able to perform the task a number of times to build a set of examples. The one place in which a specification of correct behavior can be used to aid the error detection process, however, is in the manual specification of the hierarchical behavior representation corresponding to the lower boundary node as discussed in Section 6.6.1. Here too, an imprecise specification is acceptable because all that is needed is a lower bound on correct behavior. Thus we do not need to know all of the parameters of correct agent behavior, but rather only a subset of them.

Unlike past approaches, behavior bounding supports all three of our target criteria. However, we have yet to examine its efficacy in use. In the remainder of this chapter, we will address this question and then return to the complete set of criteria set forth in Chapter 3 to assess its overall performance.

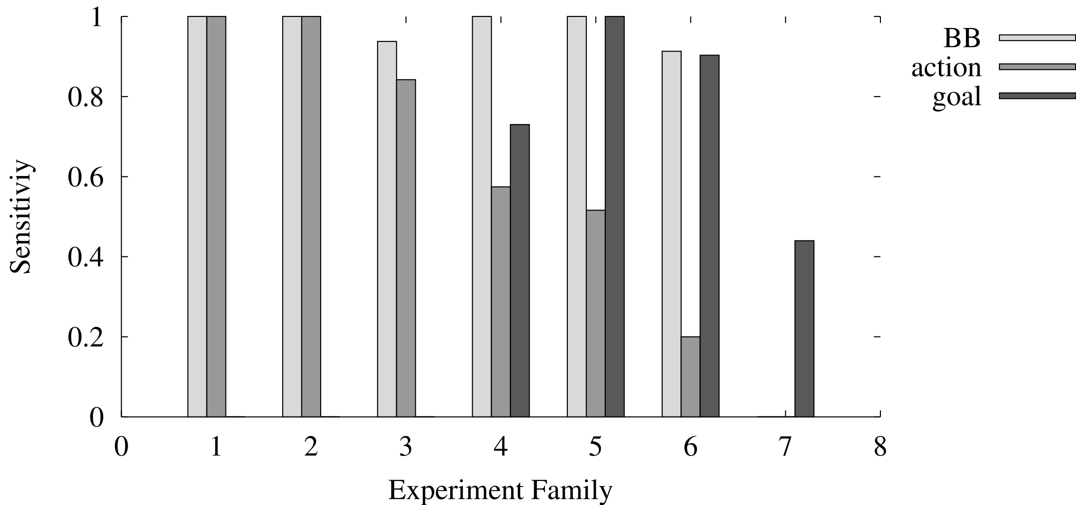

Figure 6.6: Behavior bounding: sensitivity in the object retrieval domain

6.4 Empirical Evaluation

To empirically evaluate behavior bounding, we examined its performance in both the object-retrieval and MOUT environments discussed in Section 5.3. To facilitate a direct comparison between behavior bounding and the sequential approaches, we ran our tests using the same experimental setup as before.

6.4.1 The Object Retrieval Environment

Figure 6.6 illustrates the sensitivity across each experiment family in the object retrieval domain. The figure illustrates two main phenomena. The first and most obvious is that overall, behavior bounding is better at identifying behavior errors than either the goal or action based sequential comparison methods. In fact, behavior bounding equals or betters the sensitivity of the combined action and goal sequence described in Chapter 5 on all but the final experiment family. The poor performance on this final experiment family is due to limitations of the hierarchical representation itself which we discuss below.

In the seventh experiment family, the expert's behavior contains traces in which a particular goal is decomposed in two ways. For simplicity, we'll call this problematic goal P. The first way the expert completes P is by pursuing two subgoals, call these A and B, in the following sequence: A, B, A. The second decomposition is performed by pursuing

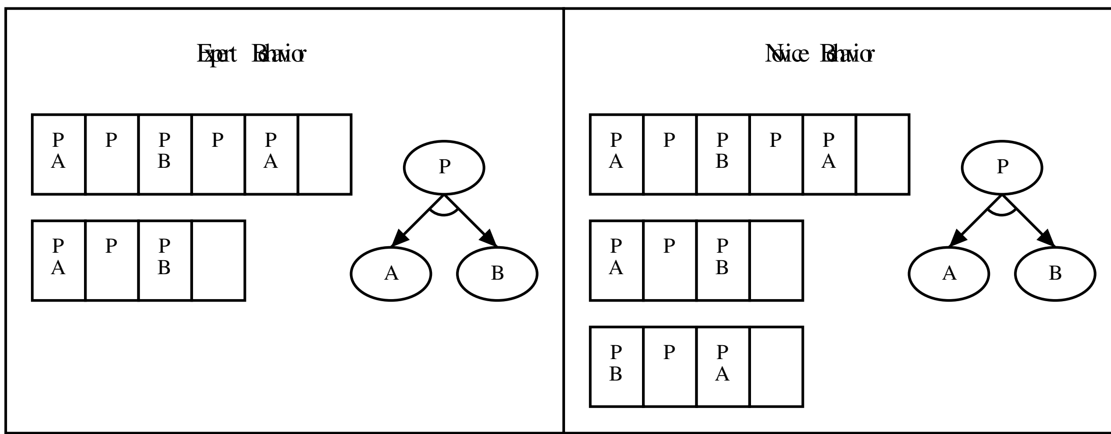

Figure 6.7: Limitations of behavior bounding's HBR in experiment family seven

these same subgoals but in the simplified sequence: A, B. Importantly, the expert will never attempt the following decomposition: $P \rightarrow B, A$. However, when the first behavior trace is processed to form the hierarchical behavior representation, over-generalization occurs. As discussed previously, the HBR contains only a single node to represent each instance of identically named goals with the same lineage. Thus when the first trace, containing the decomposition $P \rightarrow A, B, A$, is processed, only three nodes are formed—one for P, A, and B respectively. To accommodate the fact that A is observed to occur both before and after B, temporal constraints are completely generalized between these two nodes. This situation is illustrated on the left hand side of Figure 6.7. Unfortunately, this behavior representation fails to capture the fact that the expert would never perform $P \rightarrow B, A$. Thus, when the novice's behavior traces are processed (illustrated on the right hand side of Figure 6.7), it is of little surprise that the same HBR is produced and no differences are detected between the expert and the novice. Although there are ways to address this particular problem, we have yet to identify a general method that maintains the concise representation of behavior that we desire.

Behavior bounding's ability to detect errors while maintaining very concise reports is illustrated by its relatively high report density (see Figure 6.8). Recall that report density measures the amount of useful information in an error detection method's summary; scores less than but near to one indicate that on average one error could be detected for each discrepancy indicated in the summary, whereas scores less than one indicate some amount of superfluous information (false positives) is contained in the summary. Report

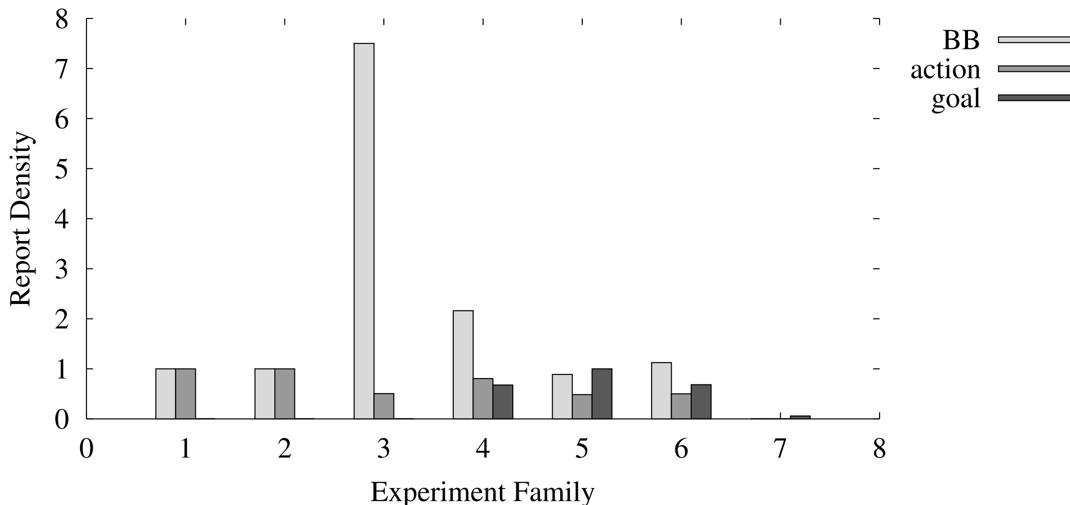

Figure 6.8: Behavior bounding: report density in the object retrieval domain

density scores higher than one are also possible, but only when reports remain exceedingly concise and identify high-level errors that correspond to multiple low-level errors. Because of behavior bounding's ability to concisely represent relationships between goals via decomposition and ordering constraints, it is well suited to identifying misplacement errors. Moreover, because the structures being compared are relatively small (compared to the set of sequences being compared in the sequential approach) behavior bounding can easily maintain a relatively low false positive count.

Behavior bounding's performance in the object-retrieval environment is encouraging. Overall, it performs well against the sequential comparison approaches we examined in Chapter 5 even though it's internal representation of behavior is constrained by our desires to maintain efficiency across environments of differing complexity.

6.4.2 The MOUT Environment

In Chapter 5, the increased complexity of the MOUT environment exhibited clear evidence that the straightforward sequential method of identifying differences in two actor's behavior is unlikely to scale well beyond relatively simple domains. Figure 6.9 illustrates behavior bounding's sensitivity compared to that of the sequential approaches. Results here are not particularly dramatic, but behavior bounding does have fewer instances of zero sensitivity (inability to identify any errors) than either of the sequential approaches.

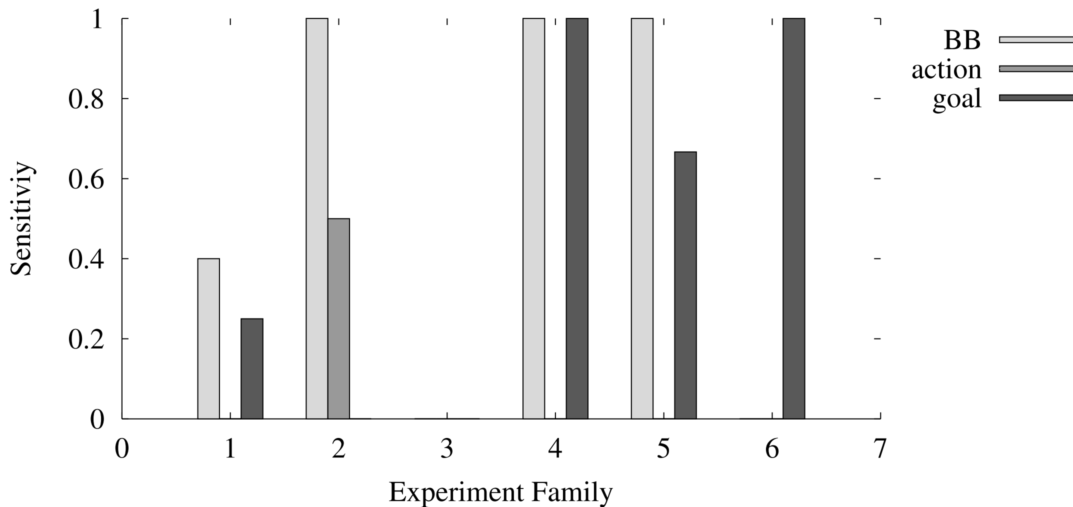

Figure 6.9: Behavior bounding: sensitivity in the MOUT domain

In experiment families three and six, errors are missed once again because one aspect of the hierarchical behavior representation becomes over-generalized.

Some of behavior bounding's strengths are better illustrated when we examine report density, as in Figure 6.10. Compared against either of the sequential approaches, behavior bounding's report density is exceedingly high. In cases where true errors are detected, the report density averages near 0.20, detecting about one true error for every five differences reported in the summary. Even though report density is lower than in the relatively simple object-retrieval domain, it is still high enough to be useful for the purposes of validating an agent's knowledge base.

Although behavior bounding clearly outperformed the sequential methods in the MOUT domain, there is obvious room for improvement. To identify why its efficacy was low compared to the object-retrieval domain, we looked back upon the domain itself and the novice-level agents that we examined.

One noticeable source of false positives, was due to so called *floating operators.* Floating operators are not performed in service of their parent goal. Essentially, they are goals or actions that occur opportunistically, potentially at any location in the goal hierarchy in order to respond to the dynamics of the environment without explicitly suspending or canceling the agent's goals. Because floating operators do not work in service of their parent goal, they effectively break the paradigm of the hierarchical behavior represen-

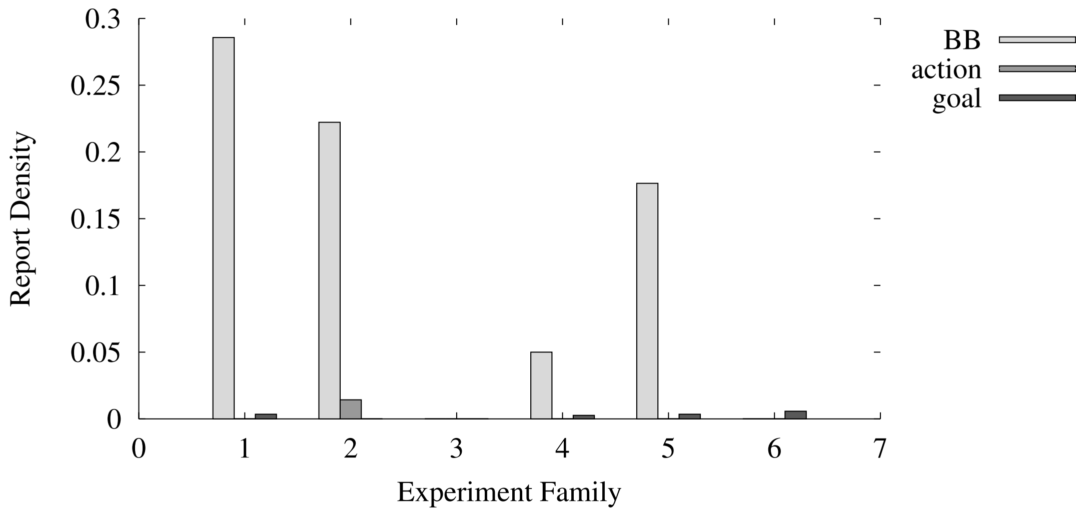

Figure 6.10: Behavior bounding: report density in the MOUT domain

tation; their effects can be twofold: first, they are likely to cause over-generalization by inappropriately changing the parent's node type from And to Or. Secondly, if limited observations are available, floating operators can result in representations of the novice agent's behavior that are inconsistent with the structure of the expert's behavior representation, thus failing to satisfy the lower bound on potentially correct behavior.

There are a number of potential methods that could be used to circumvent these problems. One method would be to create a level of indirection between the expert's native behavior representation and what is presented in the behavior traces. Through some preprocessing of the behavior traces, it would be possible to modify the topology of the expert's goal hierarchy so that floating operators no longer appeared (i.e., so they were mapped to static locations in the hierarchy). Although this could help circumvent the issues with floating operators, it may require significant engineering resources to process the behavior traces. More importantly, however, this introduces another source for errors and confusion, and as a result is probably best avoided. Another approach would be to tag floating operators so they could be treated differently by the Create-Hierarchy algorithm[3]. This would increase the initial cost of using behavior bounding to validate an agent, but it is likely that this cost would remain minor. A third method is simply

[3]While it may be possible to tag floating operators automatically based on where they occur in the goal hierarchy, and what generalizations they cause, it would be safest to require the knowledge engineer to provide the tags before the behavior comparison was performed.

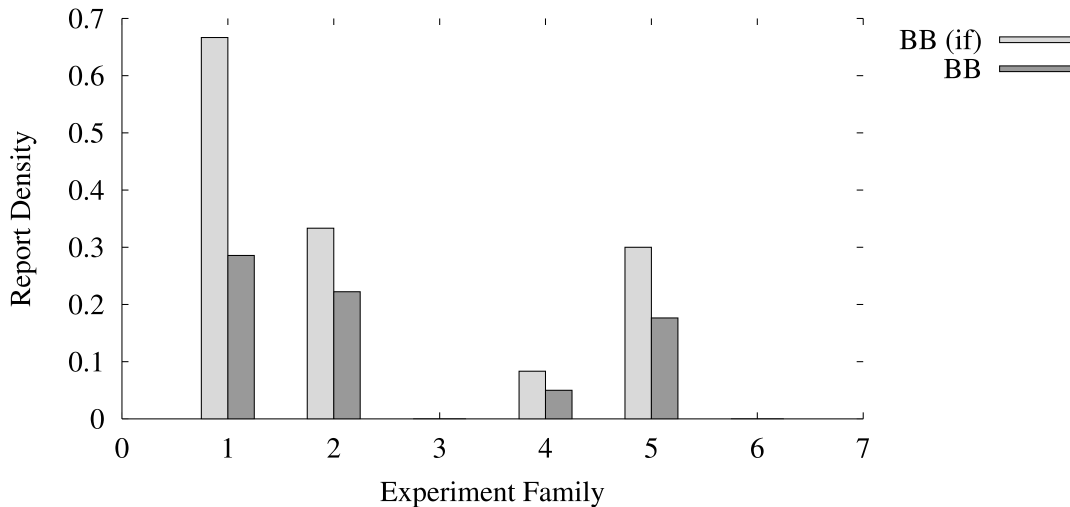

Figure 6.11: Behavior bounding: report density in MOUT when ignoring floating operators

to ignore floating operators altogether. Although this, of course, has the potential of reducing the number of errors that can be detected, it is also likely to have a significant payoff in terms of reducing false positives. Moreover, because floating operators do not fit naturally into behavior bounding's structure, it is likely that errors that do occur in the floating operators might be missed even if they were included in the HBR.

Figure 6.11 illustrates the effect on report density when floating operators are ignored. As expected, the number of false positives is reduced, thus increasing the report density on all experiment families other than 3 and 6 (where no errors are correctly identified with either method). Although the effect is somewhat subtle, it does raise the average report density (excluding experiment families 3 and 6) by nearly a factor of 2, from 0.18 to 0.35, an effect that makes the already acceptable error summary much more useful.

6.5 Efficacy as a Validation Tool

We have shown that behavior bounding has acceptable performance in two domains of distinct complexity, and argued that it would be well suited for detecting errors in many other goal oriented environments. However, up to this point, we have only hypothesized that the error reports provided by behavior bounding will decrease validation cost; we have not provided any direct evidence.

	Expert	Novice-A	Novice-B
Modification	N/A	New Proposal	Missing Preference
Manifestation	N/A	Intrusion	Commission
Distinct Behaviors	4	12	8
Consistent BTs	N/A	4	4
Avg. BT Length	67	69	68

Table 6.1: Properties of expert & novice agents in the validation efficacy test

To substantiate this claim, we performed an informal experiment in which a number of users attempted to find and correct flaws in an agent's behavior. As in previous experiments, agents were implemented in the Soar architecture. Each participant was a member of the Soar research group with at least six months of Soar programming experience. Participants identified two behavior flaws, one using standard debugging tools, and the other using information from behavior bounding's error summary. Once these flaws were identified, the participants corrected the agents' knowledge using VisualSoar, the standard Soar development environment.

Our test-bed agent was taken from the object retrieval domain discussed in Sections 5.3.1 and 6.4.1. The initial setup followed similar lines as our earlier experiments. We began by constructing an expert-level agent. This agent could perform its task in four distinct, but similar ways. We then constructed two novice-level agents based on this expert. Because each participant would validate each agent using a different method, one of our primary desires was to construct novice-level agents in such a way that they would be similarly difficult to validate. To help ensure that this was the case, we limited the differences in the novice's and expert's knowledge to a single rule. In the case of Novice-A, one rule was added that resulted in the agent performing a different sequence of actions than the expert. In the case of Novice-B, a preference rule was removed, resulting in two discrepancies: one in the parameters of the agent's internal goal and another in the parameters of the agent's primitive action. Aside from the differences mentioned above, the behavior of both novice-level agents was similar to that of the expert in all other respects.

Table 6.1 illustrates some of the important properties of the expert-level and novice-level agents. The first and second rows indicate the change that was made to construct each of the novice agents, and the form of error that results from these changes. The

Participant	Soar	BB Summary	Method Used First
1	A	B	BB Summary
2	B	A	BB Summary
3	A	B	Soar
4	B	A	Soar
5	A	B	Soar

Table 6.2: Assignment of validation methods and novice agents

third row indicates how many distinct behavior traces each agent is capable of generating. This value is important because it gives an indication of how many behavior traces the user might need to examine in order to get a good understanding of the range of behavior each agent is capable of producing. The fourth row indicates how many of the novice's behavior traces were consistent with expert behavior traces (i.e., error free). Finally, the fifth row indicates the average length of each agent's behavior trace. This gives some indication as to how much information must be examined in each instance of behavior.

Before each participant began the experiment, they were asked to read a short informational summary. This provided an overview of the experiment including a description of the debugging task, a summary of the agent's behavior, and a plain English description of some salient operators used by the agents. This overview was intended to familiarize the users with the agents and the domain without requiring each participant to build their own agent from the ground up. In addition, the participants were alerted that only one rule in each of the agent's knowledge bases had been modified, removed or added in order to generate the behavioral deviations. This information was provided to ensure that the time requirements of the participants was kept within a reasonable bound.

After the participant had read the informational summary, they were randomly assigned an agent to validate. We varied the pairing between the agent and the validation method as well as the order in which the validation methods were used, resulting in four permutations. Table 6.2 indicates each participant's (agent,validation method) pairings. For each experiment, we asked the participants to indicate when they were ready to modify the agent's knowledge and to articulate what changes they believed were required. This allowed us to measure the amount of time needed to identify the behavioral flaw as well as the total time required to correct the agent's behavior.

During the first phase of the debugging session, participants identified the flaw in

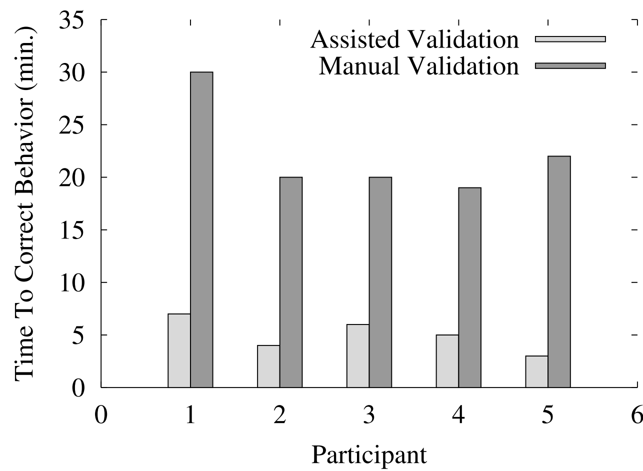

Figure 6.12: Total time required by each participant to identify and correct the agent error

the novice agent's behavior. In the Soar environment, no specific instructions on how to identify the errors were given. Participants were free look for errors using whatever debugging techniques they had developed previously. Because none of the participants had used, or even seen, the graphical behavior comparisons generated by behavior bounding, however, an initial tutorial was required. For this tutorial, each participant was given an overview of how behavior bounding worked and how it displayed summaries of the differences between two actor's behavior. After this discussion, the users were assisted in identifying four different errors (misplacement, intrusion, omission, commission) in an exceedingly simple mock environment. Once the participants had completed this tutorial, they were ready to examine the novice agent's behavior for flaws.

The second phase of the debugging session began once the participant determined that they were ready to try modifying the agent's knowledge in order to correct the error. Regardless of whether the error was identified using standard techniques or behavior bounding in the first phase, participants used the VisualSoar editing environment (a standard part of Soar's development environment) to locate and correct the faulty knowledge. Once the participant had made changes, they reexamined the novice agent's behavior to ensure that the problem had in fact been corrected. When the participant was confident that the problem was resolved, the clock was stopped and this was recorded as the total time needed to correct the agent's behavior.

Figure 6.12 illustrates the total time spent by each participant identifying and cor-

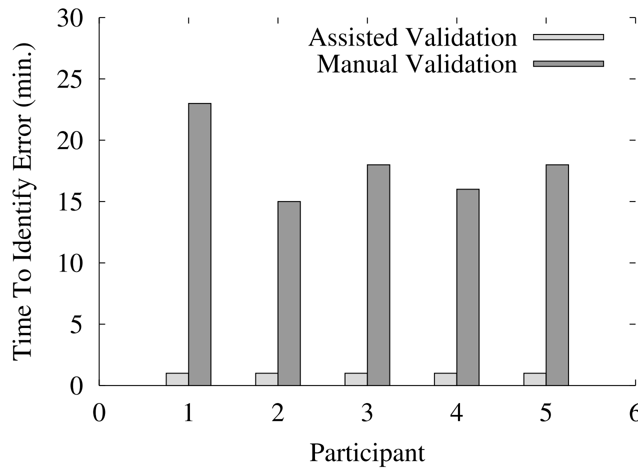

Figure 6.13: Time required by each participant to identify the agent error

recting errors in the novice agent's knowledge. For the first three points, Novice-A is being validated using standard debugging tools, while Novice-B is being validated using behavior bounding. For the remaining two points, the reverse situation occurs. Note that the times reported for these participants all fall within a relatively small range even when we consider times across all tasks (less than a factor of 1.6 between the slowest and fastest participant using the standard Soar debugging tools).

Figure 6.13 illustrates the time required to identify the error in the agent's behavior. Recall that each participant was asked to articulate what error they identified in the agent's behavior before attempting to correct it. Time to identify the error was measured from the beginning of the examination until the participant was able to identify the actual aspect of the agent's behavior that needed to be changed (e.g., by saying "I need to ensure action `X` occurs before action `Y`"). The figure illustrates how much time savings can be realized using the graphical error summary provided by behavior bounding.

As in Figure 6.12, the time required to identify errors also has low variance. Measurements using the standard Soar debugging tools varied by less that a factor of 1.44 even though in at least one of the participant's cases correctly identifying the error required a relatively time consuming multiphase process of looking at the agent's behavior using the normal Soar debugging tools, looking at the agent's knowledge base and then re-examining the agent's behavior before finally identifying the correct error. Somewhat surprising to us was the fact that measuring time with minute precision yielded no vari-

ance between users of the behavior bounding method. Although it is unfortunate that we cannot inspect these measurements in greater detail, we were extremely encouraged by the fact that there was such a significant difference with and without the aid of behavior bounding's error report.

From the graphs, it seems safe to conclude that the error report provided by behavior bounding does, in fact, provide information that is both relevant to identifying differences between two agents' behavior and useful in isolating faulty knowledge. Although on one level these results may be considered best cases because we constructed errors that we believed would demonstrate the effectiveness of behavior bounding, there are at least two reasons why these results may be on the conservative side of optimistic.

First, recall that one of behavior bounding's main advantage is that it represents aggregate behavior concisely, in a single hierarchical data structure. This means that as the complexity of the agent's behavior grows, we expect to see relatively small increases in the size of the hierarchical behavior representation, as opposed to potentially larger growth in the length of the overall behavior. Recall that it is this longer sequence of behavior that is examined using standard Soar debugging tools. Thus, we would expect that as the complexity of the agent's behavior increases, debugging time increases more slowly using behavior bounding as opposed to standard tools.

A second advantage of behavior bounding's concise hierarchical representation is that its growth is limited to the size of the agent's complete goal hierarchy regardless of how many behavior traces are examined. In contrast, using Soar's standard debugging tools, a knowledge engineer must examine each instance of behavior separately. This means that as the variability of the agent's behavior increases, we should again expect debugging time to increase more slowly using behavior bounding as opposed to standard tools.

Finally, it is also interesting to note that this test is clearly heavily influenced by the design of behavior bounding's user interface. Although we attempted to make an intuitively simple interface, no real experiments were conducted to determine the best way to graphically present the behavior summary. As a result, it is possible that future implementations would be capable of producing an even larger increase in efficiency.

6.6 Extensions to Behavior Bounding

Our experiments with behavior bounding have all yielded encouraging results. Yet, in the complex MOUT domain, our results do leave room for improvement. In Section 6.3.3, we examined some expressivity limitations of behavior bounding's HBR and possible modifications to circumvent these limitations. In this section, we examine two other extensions to behavior bounding that may also improve its performance.

6.6.1 Manual Definition of Lower Boundary Node

By itself, the lower boundary is a minimal specification of the parameters necessary for correct behavior. That is, it does not contain all the constraints required to discriminate between correct and incorrect behavior. Although we have suggested that the lower boundary node is easily formed by completely generalizing the upper boundary node, a better approach may be to construct it manually.

Because the hierarchy represented by the lower boundary node simply identifies the space of potentially acceptable goal decompositions, it could be created as part of the knowledge acquisition process. In fact, Lee and O'Keefe [28] and Yen [59] have argued independently that constructing an overview of the ways in which goals decompose into sub-goals and primitive actions is an important step in the knowledge-acquisition process. Moreover, they argue that identifying the relationship between goals, sub-goals and primitive actions helps to organize the agent's knowledge base and serves as a foundation for further knowledge-acquisition. Thus, it may be the case that constructing the lower boundary node manually is a process that introduces little or no additional effort on the part of the domain expert and the knowledge engineer. In fact, it may actually benefit knowledge acquisition by providing more structure to the process.

If constructing the hierarchical representation corresponding to the lower boundary node by hand is a relatively low cost process, it is reasonable to ask how this manual effort could be leveraged to improve behavior bounding's performance. At least two improvements are possible.

The most obvious use of the manually constructed HBR is to help validate the agent's design early during the implementation process. A number of authors have suggested that the earlier validation can take place, the less costly it will be. By constructing the lower boundary by hand, it may be possible to identify whether the agent adheres to

these constraints by statically analyzing its knowledge–without needing to see the agent interact with the environment.

A second way to leverage the fact that the HBR is constructed manually would be to embed new information within it. This information would provide more details about the minimal requirements for acceptable behavior than could be specified by the goal decomposition relationships alone. Arbitrary information could be used to augment this HBR, it need not even be limited by the representational constraints of the upper boundary node. Because these augmentations will all be made manually, the only limitations would be to ensure that the overall effort required to augment the lower boundary node remains relatively low, and to ensure that the novice's behavior can be compared against the new lower boundary efficiently[4].

6.6.2 Sometimes/Always Constraints

Another potentially useful modification to the HBR, would be to change the association of the node type constraints. In the current version of behavior bounding, And and Or constraints are associated with parent goal nodes. Alternatively, we might associate similar labels with the child nodes such as Sometimes and Always. Although the change is subtle, it would offer somewhat greater representational power. The semantics of And and Or nodes are easily covered and it is also possible to encapsulate new decomposition relations. Moreover, this change clearly has no effect on the learnability or construction cost of the representation. We have not tested this modification in detail, but preliminary results in the MOUT domain do indicate a minor improvement in performance.

6.7 Satisfying Properties of an Ideal Approach

In Section 6.3.4 we examined three properties of an ideal error detection method. Now, we return to the full set of properties enumerated in Chapter 3. The first four properties of an ideal error detection method describe the overall effort required to use or implement the approach. We have already argued that behavior bounding meets two of these four properties: minimizing supportive engineering (**P2**) and providing a grounded

[4]Note however, if the lower bound contains constraints that cannot be represented in the HBR, some modifications to the error detection process will be required.

	Cost				Human-Level Agent Support				
Method	P1	P2	P3	P4	P5	P6	P7	P8	P9
Behavior Bounding	√	√	√	√	√	√	√	√	√
Sequential Approach	√	√	—	—	√	√	√	**Some**	√
Manual Validation	—	√	√	—	√	√	√	√	√

Table 6.3: Properties of behavior bounding

stopping criterion (**P4**). The remaining two properties are discussed below.

Like the sequential approach to behavior comparison, behavior bounding minimizes human oversight by providing a summary of the similarities and differences between two agents behavior. As our experiments in the MOUT domain indicated, even in complex environments, behavior bounding provides much more concise summaries than the sequential approach, typically keeping false positives within reasonable bounds. As a result, we consider behavior bounding to have successfully minimized human oversight (**P1**).

The fourth requirement is to minimize reliance on example behavior. Unlike the sequential approach to behavior comparison, behavior bounding has polynomial sample complexity. This means that the number of behavior traces required to learn the aggregate representation of behavior will maintain a reasonable growth rate even in complex environments, thus satisfying (**P3**).

The remaining five requirements address a comparison method's ability to support human-level behavior. Earlier in this chapter, we identified how behavior bounding meets one of these criteria (**P8**: Supporting imprecise specifications of correctness). Here we examine the remaining properties.

One of behavior bounding's most appealing attributes is that it can be used without any domain knowledge beyond what is available directly from the actors' behavior traces. This property makes it easy to adapt this approach to different environments (**P5**) simply by gathering examples from the new environments and using them to generating new HBRs. Similarly, because behavior bounding relies on examples of expert behavior to generate its model of correctness, and because the domain expert and knowledge engineer are involved in the final phase of diagnosing errors (because they interpret the report) behavior bounding leverages tacit expert knowledge similar to manual approaches to validation (**P6**).

Finally, behavior bounding's HBR represents information about the actors' internal behavior (goal selections) and external behavior (actions). This means that it can be used to evaluate different aspects of reasoning (**P7**). Furthermore, because the HBR is built from information in the behavior traces, and because these traces contain information only about the actor's state, goals and actions, behavior bounding can be used with to compare actors regardless of whether they are humans or machines (**P8**).

Overall, behavior bounding meets all the criteria set forth in Chapter 3. Table 6.3 summarizes its ability in relation to the manual approach to validation and the sequential approach to behavior comparison.

Chapter 7

Conclusion

In the previous chapters, we examined the problem of how constructing complex human-level agents could best be facilitated. The general class of methods we investigated were all aimed at providing summaries of similarities and differences between two agent's behavior. Because identifying behavior differences is a problem that must be addressed by many areas of artificial intelligence research, the methods we examined in this thesis could be applied to a number of problems beside knowledge-base validation including:

- Scoring a strict Turing Test: a general approach to detecting behavior errors could be used to objectively measure differences between human behavior and software agent behavior. A perfect score would be indicated if no detectable differences were identified.
- Automatic correction of knowledge bases: a system which modifies an agent's knowledge to remove bugs must determine when the agent's knowledge needs to be changed and when changes have improved the agents behavior. A general approach to error detection could serve both of these purposes.
- Intelligent tutoring systems: fundamentally, the problem of identifying errors in human behavior is very similar to identifying errors in software agent behavior. So long as the error detection system relies on information that can be obtained from either a human or a software agent, it should be possible to reverse their roles such that a novice human's behavior is compared against an expert agent's behavior to determine when an error has been made.
- Intelligent interfaces: some interfaces, or controls, are used for relatively narrow, focused tasks. In these situations it would be possible to provide a usage model

that covers each of the interface's prescribed functions. By comparing the user's behavior to this gold standard, the interface may be able to help the user perform their task, or even recover from incorrect user behavior.

- Machine learning; systems that learn how to perform a task must be able to evaluate the knowledge they have created. In particular, systems that learn by observation stand to gain from generalized error detection approaches by increasing their ability to reflect on how their learned behavior deviates from observations of expert behavior.

Originally, we proposed examining five approaches to comparing behavior. All of these methods could be considered among the general class of sequence-based approaches described in Chapter 5. As our investigation began, it became increasingly clear that in general, sequence-based approaches held little, if any, promise in domains with moderate to high complexity. In particular, we noticed that these methods' poor ability to create a compact representation of aggregate behavior was one of their greatest weaknesses. Instead of continuing down this path and exploring all five of our original sequence-based methods, we began considering other approaches that might overcome some of the weaknesses we had noticed.

After some consideration, we began to focus on hierarchical representations of aggregate agent behavior, which then led to the idea of behavior bounding. This approach addressed a number of the problems with sequence-based methods, in particular behavior bounding maintains a concise representation of behavior, and it is relatively straightforward to map between information in the summary and errors or flaws in the agent's knowledge.

Although it is by no means perfect, behavior bounding performs well in both our testbed domains. Moreover, we were also able to show that information provided by behavior bounding's summary report provides information that can be used to dramatically decrease validation costs even when users have relatively little experience using it. With future enhancements we should expect that behavior bounding's ability to detect salient differences between two actors' behavior can be improved beyond its current capabilities. However, by maintaining the requirement that the representation of the actor's behavior is concise, efficient, and easy to understand it is likely that behavior bounding will never be able to achieve perfect sensitivity across all domains and across all forms of errors.

7.1 Contributions

In Chapter 1 we indicated that the primary goal of our research was to explore different methods for comparing two actor's behavior, and that our work makes five main contributions to the field of artificial intelligence. In this section, we examine those contributions in more detail examining them in the order they are presented in this thesis.

First, in Chapter 4, we identified a number of different ways in which behavior deviations can be classified. We proposed a set of useful classifications for different behavior deviations and a general method of determining the relative salience among these classes. This salience scheme allows us to examine the performance of different behavior comparison methods even if they identify deviations in distinct and dissimilar ways. Equally importantly, we introduced the report density metric that allows us to measure the quality of information in an error detection method's summary.

In Chapter 5 we described the sequence-based approach to behavior comparison. In addition, we performed both a theoretical and empirical analysis of its strengths and weakness. Through this analysis we were able to conclude that this general class of comparison methods has significant shortcomings that are become increasingly pronounced in complex domains.

Finally, in Chapter 6, we made a number of significant contributions. We began by describing behavior bounding, a novel method to identify differences in two agent's behavior that improves upon the sequential method by representing behavior with a concise model as opposed to an set of observations whose growth may be unlimited. Moreover, behavior bounding improves upon previous model based approaches because it requires minimal engineering to construct (it is learned directly from observations), it leverages tacit expert knowledge (because the expert need not articulate parameters for correct behavior) and it provides a grounded stopping criteria (because PAC-Learning can tell us when the HBR is within a specified tolerance of correctness).

Although behavior bounding's hierarchical representation is based in part on similar structures used throughout the artificial intelligence community, its method for classifying behavior as correct or incorrect is very reminiscent of the ideas behind Mitchell's version spaces [33]. In Chapter 6 we identified the relationship between knowledge base validation and machine learning by showing how behavior bounding's constrained hierarchical representation can be learned efficiently from a set of observations of expert

behavior and then used to identify discrepancies in a novice agent's behavior.

Lastly, we performed both a theoretical and an empirical analysis of behavior bounding. Although this illustrated that the approach was not without flaws, it did present convincing evidence that behavior bounding could be helpful in reducing an agent's development cost.

7.2 Future Work

As with most research projects, this one presents as many new questions as it answers. Although it seems reasonable at this point to close the book on sequence-based methods for comparing behavior, there are many questions about behavior bounding that are yet to be answered. A number of these would be obvious starting points for future work. Below, we examine five of the most interesting questions.

Clearly, the results our empirical evaluation of behavior bounding in the complex testbed environment were not ideal. Perhaps the first question to address is how we can improve behavior bounding's efficacy without overly complicating its model. In particular, it would interesting to see how much improvement could be made while still maintaining PAC-Learnability. Initially, we should begin by enriching behavior bounding's HBR using some of the enhancements discussed in Chapter 6. However, we would also like to enable behavior bounding to deal with errors of repetition. Identifying a fundamentally sound method for detecting these errors while maintaining a concise, PAC-Learnable representation is still an open problem.

Second, in Chapter 6 we stated that two assumptions leveraged by behavior bounding are that the agent's knowledge about performing its particular task is based around a hierarchy of goals and that constraints between goals are only necessary between sibling nodes. It would be interesting to conduct an experiment that examined how violating these assumptions affected the performance of the approach.

One can imagine building a testbed in which a template describes a general agent structure that can be instantiated to form a number of similar, but distinct agents. We could use such a testbed to design expert-level agents that varied along two dimensions: sub-goal independence and expressiveness of the hierarchy (measured, for example, by inverse branching factor). A detailed analysis could be performed in a manner similar to our earlier experiments: creating a series of expert/novice pairs and examining the

performance of behavior bounding on each of these test cases. This would require manual analysis of a potentially huge number of novice-level agents. As a result, it is likely that performing this experiment would require some new ability to partially automate the performance analysis, or a less time intensive performance metric.

Third, in the initial chapters of these thesis, we suggested that it would be possible to compare both humans and agents using the methods we presented. Although the data that comprises behavior traces can theoretically be acquired from humans, we have not demonstrated that it is, in fact, practical to do so. This leaves a clear avenue for future work.

Fourth, we have also made the claim that behavior comparison is relevant to a number of problems in AI. We mentioned five of these earlier in this chapter and in Section 1.3. In the future, it would be interesting to apply our behavior comparison techniques to these other domains. This work would help us reach a better understanding about the role of behavior comparison plays in each of these domains and would perhaps identify important attributes of behavior comparison methods that we overlooked because of our concentration on the validation problem.

Finally, the founding premise of building an agent validation system is to reduce the time required to develop and field intelligent systems. Although the current system works fine in a laboratory setting, a significant step is still required to integrate the system within a development environment so that it has the opportunity to play a real role in the design, development and testing of future human-level agents.

Appendices

Appendix A

The Create-Hierarchy Algorithm

In Chapter 6, we presented the Create-Hierarchy algorithm, which we also illustrate here in Figure A.1. At a high-level, this algorithm examines one or more behavior traces and builds a hierarchical representation of the goals, sub-goals and primitive actions that are used to perform a particular task. By calling this function with a single behavior trace, B, and $H \leftarrow$NIL, a hierarchical representation of a single behavior trace is generated. By calling the method iteratively with each $B \in \mathcal{B}$, H is augmented and generalized until it covers all of the examples it has been presented. In this chapter we decompose the algorithm and examine it in detail. As part of this examination, we will present complexity results with respect to three variables: l, the length of the behavior trace; d, the depth of the goal hierarchy; and b, the branching factor of the goal hierarchy. At the end of the analysis, we will simply our results, illustrating the derivation of the complexity analysis presented in Chapter 6.

The algorithm begins by defining two local variables on lines 1 & 2. W is a hierarchical behavior representation similar to what will be stored in H. However, W stores information that is relevant only to the current instance of a particular goal, thus we refer to it as the *working* hierarchical representation, and we refer to H as the *final* hierarchical representation. The variable *lastStk* is initially set to NIL. As the behavior trace is processed, and the tuple $(s, G, a)_i$ is read, *lastStk* stores the goal/action stack, $[G_{i-1}, a_{i-1}]$, and is used to determine if the agent has completed any the goals it was previously pursuing.

On line 3, the main loop of the algorithm begins in which each tuple $(s, G, a) \in B$ is processed. Two inner loops are performed before *lastStk* is updated on line 26 and the next tuple is processed. The first inner loop, beginning on line 5 can be described as merging the working hierarchy, W, with the final hierarchy H. This loop is only activated when a goal or action that was previously being pursued has come to completion (signaled by the fact that it no longer appears in the agents goal/action stack). The second loop, which begins on line 14, builds the working hierarchical representation (W) based on what goals action actions the actor is currently pursuing. Because this second loop is always

```
CREATE-HIERARCHY(B, H)
 1  W ← empty tree
 2  lastStk ← NIL  // previous goal/action stack
 3  for each (s, G, a) in B
 4  do
 5      for i = 0 to length[lastStk]
 6      do
 7          if GOAL-COMPLETED(lastStk[i])
 8             then h_g ← FIND-NODE(H, lastStk[i])
 9                  if h_g = NIL
10                     then
11                          ADD-SUBTREE(H, PARENT(lastStk[i]), lastStk[i])
12                     else
13                          GENERALIZE(H, h_g, W, lastStk[i])
14      for each g_i in [G, a]
15      do
16          p_g ← PARENT(g_i)
17          w_g ← FIND-NODE(W, p_g, g_i)
18          if w_g = NIL
19             then
20                  w_g ← ADD-NODE(W, p_g, g_i)
21                  CONSTRAIN-CHILDREN(W, p_g)
22             else
23                  if OUT-OF-ORDER(W, p_g, w_g)
24                     then UPDATE-CONSTRAINTS(W, p_g, w_g)
25                  GENERALIZE(w_g, g_i)
26      lastStk ← [G, a]
27  return H
```

Figure A.1: The CREATE-HIERARCHY algorithm redeux

invoked, we will examine its behavior first, following the algorithm's natural execution path.

The inner loop beginning on line 14, processes the actor's current goal/action stack. To do this, we descend the stack and maintain two new inner variables. The first variable, p_g, is associated with the current symbol's parent (which is also g_i from the previous iteration). The second variable, w_g, is the node in the working hierarchy, W, that corresponds to g_i. The call FIND-NODE takes p_g as an argument simply to illustrate that finding w_g can be done in constant time by a hash table lookup.

If g_i cannot be found in the working hierarchy, w_g is set to NIL and it must be added into W as a child of p_g. This operation occurs on line 20 and can clearly be done in constant time. When this new node is added to W, we must also ensure that ordering constraints are added to all of the other children of p_g. This is done on line 21, in $O(b)$ time.

Returning to the call on line 17, we examine the case where g_i was found in the working hierarchy. In this situation, $w_g \neq$NIL, and the algorithm enters the block beginning on line 23. First, we check to see if the ordering constraints between all of the children of p_i are consistent. In other words, we check to see if these constraints allow w_g to occur in its current position (after all the children that have previously been added to g_i). If not, we update these constraints on line 24 in a worst case time bounded by $O(b)$. Regardless of whether ordering constraints needed to be generalized, on line 25 we generalize the data in the w_g so that all the goal parameters represented in g_i are covered. Typically the goals have a relatively small number of associated parameters. Thus it is reasonable to assume that this value can be capped at a small constant, giving this step in the algorithm constant cost as well. On line 26, the *lastStk* variable is replaced with the current goal/action stack, and the next tuple $(s, G, a)_{i+1}$ is processed.

The second time through this outer loop, it is possible that some of the elements in $[G, a]$ do not correspond to those in *lastStk*. If this is the case, we consider those goals in *lastStk* that do not have mates to have been completed. In lines 5–7, we examine *lastStk* for exactly this condition, and we consider the GOAL-COMPLETED operation to be a constant time hash-key comparison. If a goal has been completed, then the branch in W corresponding to this goal can be mapped into the final hierarchical representation H. This process is very similar to how W is built in lines 17–25. As before, we first determine if there is a node in H corresponding to the completed goal ($lastStk[i]$).

If there is no node in H corresponding to the completed goal, then $h_g =$NIL, and we must add the branch from W rooted at $lastStk[i]$ (line 11). Depending the size of the branch, this can be an expensive operation because all descendants will need to be visited and copied. In the worst case, the entire goal hierarchy may need to be copied into H giving this operation a cost bounded by b^d.

In the alternative case, a node in H corresponding to the completed goal is found. In this situation, the branch in H needs to be generalized with the branch in W (line 13). This is a recursive process in which each node in the branch must be generalized both with respect to ordering constraints, node-type constraints, and node parameters just as occurs in lines 23–25. Thus the cost of generalizing the branch is bounded by $b \cdot b^d$ because each of the b^d nodes has at most b ordering constraints associated with it.

Once all elements in the behavior trace have been processed, the final hierarchical representation, H, is returned. This return value can be passed back as an argument to an iterative call to the CREATE-HIERARCHY function to produce a representation that covers multiple behavior traces.

If we add up the cost of one iteration of each of the two inner loops, they are bounded by $O(b \cdot b^d)$ and $O(b)$ respectively. Since these inner loops iterate over the goal/action stack, the total time to process these loops is $O(b \cdot d \cdot b^d)$ and $O(b \cdot d)$ respectively. The outer loop requires l iterations, giving the entire algorithm a cost that is bounded by $O(l \cdot b \cdot d \cdot (b^d + 1))$ or simply $O(lbd \cdot b^d)$. If the size of the goal hierarchy is given by $N = b^d$, then we can say that the algorithm runs faster than $O(lN^2)$, which is the result presented in Chapter 6.

Appendix B

Validation Efficacy Pre-Experiment Handout

The following text is a copy of the document that was handed to the participants before they began the experiment described in Section 6.5. Minor cosmetic changes have been made to ensure that the text is consistent with the formatting of the rest of this document.

Objective

The goal behind this experiment is to determine how long it takes to find bugs in relatively simple soar programs. The agents you will debug will be referred to as the "novice" programs, while the gold standard of appropriate behavior will be referred to as the "expert". We will explore two debugging methods, the first using standard soar debugging tools and the second using a behavioral comparator which highlights differences between two agent's behavior.

Steps in the Task

1. Examine the novice and expert behavior using either Soar's basic tools or the behavior comparator.

2. Identify differences between the expert and the agent, and articulate these to me. (i.e., "it looks like the novice never selects operator x")

3. Modify the novice's soar code so that the novice will perform the task in the same manner as the expert. **Remember, the goal is for the novice to emulate the expert's behavior.**

4. Re-examine the behavior to ensure your changes worked.

Note that the expert and the novice may be able to perform their tasks in more than one way, so you should examine their behavior on multiple passes through the environment. The idea is that you want to ensure that the set of behaviors pursued by

the novice are identical to the set of behaviors pursued by the expert. *Both the novice and the expert are soar programs, but you will only modify the novice agents.* In addition, you should not look at any rules in the expert program. It is designed to simulate a human performing the task, so the rules are considered to be inaccessible.

Overview of Agent Behavior

The agent you will be examining performs a simple errand at the local grocery store: buying cheese.

The agent begins at home, where it must select an appropriate method of transportation. It considers using a bike, car or its feet by checking their availability. Before leaving it must also plan a route to the store, multiple routes may exist and they may not all be correct. Once it has a route and a mode of transport, the agent travels to the store. At the store it roams around looking for cheese. When found, the agent proceeds to the checkout examines its wallet and checkbook for necessary funds, selects a method of payment and finally pays for the cheese. At this point the task is complete.

The constraints governing the expert's behavior are what dictate how the novice should act. It is not enough that the novice get to the store and buy the cheese. In fact, the novice already does that. Instead, your goal is to ensure that the novice behavior emulates the expert. So, for example, if the expert always travels a certain route to the store, or always pays by credit card, the novice should as well.

Overview of Some Relevant Operators

```
travel:  a high level goal used to get from home to the store

   pick-transport: a goal in which potential modes of transport are
                  examined and one is selected.

      query-<feet,bike,car>: check the availability of different
                           transport methods

      select-transport: select one of (bike, car, feet) for getting
                      to the store

   plan-route: find a path from home to the store

   move: using the selected mode of transportation, go to the store

find-dairy: Find the cheese in the store

purchase: a high level goal to buy the cheese (once it's in hand)

   pick-payment: a goal in which potential methods of payment are
                                examined and one is selected.

   query-<wallet,checkbook>:  check these sources for necessary funds
```

How Novice Behavior Differs from Expert Behavior

The novice agents are slight variations of the experts. The differences have been introduced by modifying/adding/or removing a single rule. Thus the changes that need to be made are relatively simple. Moreover, all differences in behavior can be identified simply by examining what operators are selected, and examining the augmentations of these operators. (i.e., you will not have to change state elaboration rules, only operator preferences and proposals).

Note: The expert-level agent has four (4) possible behaviors

Examining Behavior in Soar

- Begin by starting soar. In the soar directory, type `./start-soar.tcl`
- Create two agents, one which will be the expert and one which will be the novice.
- Load the expert Soar code into one agent window using the command `source load-expert`.
- Load the novice Soar code into one agent window using the command `source load-novice-a` or `source load-novice-b` depending on which agent you have been assigned.
- Examine their behavior and identify differences out loud before changing any Soar code.
- Make any necessary changes and repeat the tests to ensure things have worked.

Examining Behavior Using Comparison Tool

- Run the comparison tool. Based on the agent you have been assigned to debug, type:
  `./analyze-novice-a` or
  `./analyze-novice-b`
- Identify out loud how the novice's behavior differs from the expert's before changing the Soar code.
- Modify the agent.
- Generate new traces for the modified agent using the commands:
  `./generate-traces-for-novice-a` or
  `./generate-traces-for-novice-b`.
- Rerun the comparator to validate your changes (as in the first step).

Appendix C

Notes and Details from Analytic Results

C.1 Results from Chapter 5

C.1.1 Size of Behavior Space

Assumptions: Behavior is composed of goal and actions symbols selected from a set S. The number of symbols in a sequence of correct expert behavior falls within a finite range (E_l, E_u).

Let B_{PC} be the space of potentially correct behaviors; i.e., all the sequences with symbols drawn from S whose lengths are contained by the acceptable range.

$$|B_{PC}| = \sum_{i=E_l}^{E_u} |S|^i$$

For each sequence length i, we fill the slots of the sequence with members from S, each choice is made independently from the others.

$$|S|^{E_u} \leq |B_{PC}| \leq |S+1|^{E_u}$$

Clearly the lower bound is correct. We can think of the expression $|S+1|^{E_u}$ as referring to the number of sequences formed by symbols in S plus a special *blank symbol*. Thus this upper bound includes sequences that are effectively different lengths, from $0 \ldots E_u$ elements. Since $E_l \geq 0$, the upper bound is also clearly correct.

C.1.2 Towers of Hanoi Example

In the four disk puzzle, we being with the largest disk, and proceed to the smallest, choosing which peg to place them on. This yields 3^4 states. For each state, there are at least two possible moves, we can move the smallest top disk on to any of the other pegs, thus there are at least 162 states.

C.2 Results from Chapter 6

C.2.1 Efficacy of Lower Boundary Node

The number of possible sequences consistent with the goal decomposition of the unconstrained hierarchy is calculated in two ways:

First, for simplicity, assume that behavior sequences are always length b^d, and no symbol is repeated. For a hierarchy of depth d, and branching fact b, the possible sequences are generated by first selecting an ordering of the b nodes in the first ply of the tree. Next, we count the number of subsequences generated by each of the subtrees rooted at these b nodes, and multiply all these figures together. This gives the recurrence relation:

$$
\begin{aligned}
S(b,d) &= b!S(b,d-1)^b \\
S(b,1) &= b!
\end{aligned}
$$

We find the closed form for this relation by induction:

$$S(b,d) \stackrel{?}{=} b!^{\sum_{j=0}^{d-1} b^j}$$

Initial Case:

$$
\begin{aligned}
S(b,1) &= b!^{\sum_{j=0}^{0} b^j} \\
S(b,1) &= b!
\end{aligned}
$$

Assume the relation holds for the first $d-1$ values:

$$
\begin{aligned}
S(b,d) &= b!S(b,d-1)^b \\
S(b,d) &= b!\left(b!^{\sum_{j=0}^{d-2} b^j}\right)^b \\
b!^{\sum_{j=0}^{d-1} b^j} &\stackrel{?}{=} b!\left(b!^{\sum_{j=0}^{d-2} b^j}\right)^b \\
b!^{\sum_{j=0}^{d-1} b^j} &\stackrel{?}{=} b!\left(b!^{b\sum_{j=0}^{d-2} b^j}\right) \\
b!^{\sum_{j=0}^{d-1} b^j} &\stackrel{?}{=} b!\left(b!^{\sum_{j=0}^{d-2} b^{j+1}}\right)
\end{aligned}
$$

$$b!^{\sum_{j=0}^{d-1} b^j} \stackrel{?}{=} b!\left(b!^{\sum_{j=1}^{d-1} b^j}\right)$$

$$b!\left(b!^{\sum_{j=1}^{d-1} b^j}\right) = b!\left(b!^{\sum_{j=1}^{d-1} b^j}\right)$$

$\square$

Bibliography

Bibliography

[1] B. Anrig and J. Kohlas. Model-based reliability and diagnostic: A common framework for reliability and diagnostics. In M. Stumptner and F. Wotawa, editors, *DX'02 Thirteenth International Workshop on Principles of Diagnosis*, pages 129–136, Semmering, Austria, 2002.

[2] Lee Brownston, Robert Farrell, Elaine Kant, and Nancy Martin. *Programming Expert Systems in OPS5: An introduction to Rule-based Programing.* Addison-Wesley, Reading, Massachusetts, 1985.

[3] *CLIPS Reference Manual: Version 6.05.*

[4] Susan Craw. Refinement complements verification and validation. *International Journal of Human-Computer Studies*, 44:245–256, 1996.

[5] Susan Craw and D. Sleeman. Automating the refinement of knowledge-based systems. In L. C. Aiello, editor, *Proceedings of the ECAI90 Conference*, pages 167–172, Stockholm, Sweden, 1990.

[6] Randall Davis. Consultation, knowledge acquisition, and instruction: A case study. In P. Szolovits, editor, *Artificial Intelligence in Medicine*, chapter 3. Westview Press, Boulder Colorado, 1982.

[7] Kutluhan Erol, James Hendler, and Dana S. Nau. HTN planning: Complexity and expressivity. In *Proceedings of the Twelveth National Conference on Artificial Intelligence*, pages 1123–1128. AAAI Press/MIT Press, 1994.

[8] Yolanda Gil. Efficient domain-independent experimentation. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 128–134, 1993.

[9] Yolanda Gil and Eric Melz. Explicit representations of problem-solving strategies to support knowledge acquisition. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 469–476, 1996.

[10] Allen Ginsberg, Sholom M. Weiss, and Peter Politakis. Automatic knowledge base refinement for classification systems. *Artificial Intelligence*, 35(2):197–226, 1988.

[11] Avelino J. Gonzalez and Valerie Barr. Validation and verification of intelligent systems—what are they and how are they different? *Journal of Experimental And Theoretical Artificial Intelligence*, 12(4):407–420, 2000.

[12] D. Haussler. Quantifying inductive bias: AI learning algorithms and Valiant's learning framework. *Artificial Intelligence*, 36:177–221, 1988.

[13] John H. Holland. *Adaption in Natural and Artificial Systems*. University of Michigan Press, 1975.

[14] Adele E. Howe and Gabriel Somlo. Modeling discrete event sequences as state transition diagrams. In *Proceedings of the Second Conference on Intelligent Data Analysis* [15].

[15] Adele E. Howe and Gabriel Somlo. Modeling intelligent system execution as state transition diagrams to support debugging. In *Proceedings of the Second International Workshop on Automated Debugging*, 1997.

[16] Marcus J. Huber. JAM: A BDI-theoretic mobile agent architecture. In *Proceedings of the Third International Conference on Autonomous Agents*, pages 236–243, Seattle, Washington, May 1999.

[17] IEEE. *Glossary of Software Engineering Terminology*. Standard 610.12-1990. IEEE, 1990.

[18] Research Triangle Institute. The economic impacts of inadequate infrastructure for software testing. Planning Report 02-3. National Institute of Standards and Technology, 2002.

[19] Bonnie E. John and David E. Kieras. The GOMS family of user interface analysis techniques: Comparison and contrast. *ACM Transactions on Computer–Human Interaction*, 3(4):320–351, 1996.

[20] Randolph M. Jones, John E. Laird, Paul E. Nielsen, Karen J. Coulter, Patrick Kenny, and Frank V. Koss. Automated intelligent pilots for combat flight simulation. *AI Magazine*, 20(1):27–42, 1999.

[21] Shekhar H. Kirani, Imran A. Zualkernan, and Wei-Tek Tsai. Evaluatuion of expert system testing methods. *Communications of the ACM*, 37(11):71–81, 1994.

[22] Ron Kohavi and Foster Provost. Glossary of terms. *Machine Learning*, 30(2/3):271–274, 1998.

[23] J. E. Laird, A. Newell, and P. S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 1987.

[24] John E. Laird. Using a computer game to develop advanced AI. *Computer*, 34(7):70–75, 2001.

[25] John E. Laird and Michael van Lent. Human-level AI's killer application: Interactive computer games. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, pages 1171–1178, Austin, Texas, 2000.

[26] Nada Lavrač and Sašo Džeroski. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, New York, 1994.

[27] J. Lee, M. J. Huber, E. H. Durfee, and P. G. Kenny. UM-PRS: An implementation of the proceduaral reasoning system of multirobot applications. In *Conference on Intelligent Robotics in Field, Factory, Service, and Space (CIRFFSS'94)*, pages 842–849, Houston, Texas, 1994.

[28] Sunro Lee and Robert M. O'Keefe. Developing a strategy for expert system verification and validation. *IEEE Transactions on Systems, Man and Cybernetics*, 24(4):643–655, April 1994.

[29] V. I. Levenshtein. Binary code capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10:707–710, 1966.

[30] Peter Lucas. Analysis of notions of diagnosis. *Artificial Intelligence*, 105:295–343, 1998.

[31] Peter Lucas. Bayesian model-based diagnosis. *International Journal of Approximate Reasoning*, 27(2):99–199, 2001.

[32] Ole Jakob Mengshoel and Sintef Delab. Knowledge validation: Principles and practice. *IEEE Expert*, 8(3):62–68, 1993.

[33] Tom M. Mitchell. Generalization as search. *Artificial Intelligence*, 18(2):203–226, 1982.

[34] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[35] Patrick M. Murphy and Michael J. Pazzani. Revision of production system rule-bases. In *Proceedings of the Eleventh International Conference on Machine Learning*, pages 199–207. Morgan Kaufmann, 1994.

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

[37] Tin A. Nguyen, Walton A. Perkins, Thomas J. Laffey, and Deanne Pecora. Knowledge base verification. *AI Magazine*, 8(2):69–75, 1987.

[38] Robert M. O'Keefe, Osman Balci, and Eric P. Smith. Validating expert system performance. *IEEE Expert*, 2(4):81–90, 1987.

[39] D. E. O'Leary. Validation of expert systems with applications to auditing and accounting expert systems. *Decision Science*, 18(3):468–486, 1987.

[40] Dirk Ourston and Raymond J. Mooney. Theory refinement combining analytical and empirical methods. *Artificial Intelligence*, 66(2):273–309, 1994.

[41] Douglas Pearson. *Learning Procedural Planning Knowledge in Complex Environments*. PhD thesis, University of Michigan, 1996.

[42] David Poole. A methodology for using a default and abductive reasoning system. *International Journal of Intelligent Systems*, 5(5):521–548, December 1990.

[43] Alun D. Preece, Rajjan Shinghal, and Aïda Batarekh. Verifying expert systems: A logical framework and a practical tool. *Expert Systems With Applications*, 5:421–436, 1992.

[44] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[45] Lawrence R. Rabiner. A tutorial on hidden markov models and selected application in speech recognition. *Proceedings of the IEEE*, 77(2):257–285, February 1989.

[46] Anand S. Rao and Michael P. Georgeff. Bdi agents: From theory to practice. In *Proceedings of the First International Conference on Multiagent Systems*, San Francisco, California, 1995.

[47] Marie-Christine Rousset. On the consistency of knowledge bases: The covadis system. In *Proceedings of the Eighth European Conference on Artificial Intelligence*, pages 79–84, 1988.

[48] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propogation. In D. E. Rumelhard and J. L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1, chapter 8, pages 318–362. MIT Press, Cambridge, Massachusetts, 1986.

[49] Edward H. Shortliffe. Computer programs to support clinical decision making. *Journal of the American Medical Association*, 258(1), 1987.

[50] Patrick Simen, Thad Polk, Rick Lewis, and Eric Freedman. Goal management in a recurrent neural network. In *Proceedings of the International Conference on Computational Intelligence and Neuroscience*, 2001.

[51] Marcelo Tallis. A script-based approach to modifying knowledge-based systems. *International Journal of Human-Computer Studies*, To Appear.

[52] Wei-Tek Tsai, Rama Vishnuvajjala, and Du Zhang. Verification and validation of knowledge-based systems. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):202–212, 1999.

[53] Michael C. van Lent. *Learning Task-Performance Knowledge Through Observation*. PhD thesis, University of Michigan, 2000.

[54] Michael C. van Lent and John E. Laird. Learning hierarchical performance knowledge by observation. In *Proceedings of the 1999 International Conference on Machine Learning*, pages 229–238, 1999.

[55] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974.

[56] Scott A. Wallace and John E. Laird. Toward a methodology for AI architecture evaluation: Comparing Soar and CLIPS. In N.R. Jennings and Y. Lespérance, editors, *Intelligent Agents VI — Proceedings of the Sixth International Workshop on Agent Theories, Architectures, and Languages (ATAL-99)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, 2000.

[57] Scott A. Wallace and John E. Laird. Toward automatic knowledge validation. In *Proceedings of the Eleventh Conference on Computer Generated Forces and Behavior Representation*, pages 447–456, 2002.

[58] Scott A. Wallace and John E. Laird. Behavior bounding: Toward effective comparisons of agents & humans. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, page To appear, 2003.

[59] John Yen and Jonathan Lee. A task-based methodology for specifying expert systems. *IEEE Expert*, pages 8–15, February 1993.

[60] Lotfi A. Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.

[61] Neli Zlatareva and Alun Preece. State of the art in automated validation of knowledge-based systems. *Expert System With Applications*, 7(2):151–167, 1994.