Journal of Artificial Intelligence Research 34 (2009) 165–208 Submitted 07/08; published 03/09

Behavior Bounding: An Efficient Method for High-Level Behavior Comparison

Scott Wallace WALLACES@VANCOUVER.WSU.EDU
Washington State University Vancouver
14204 NE Salmon Creek Avenue
Vancouver, WA 98686

Abstract

In this paper, we explore methods for comparing agent behavior with human behavior to assist with validation. Our exploration begins by considering a simple method of behavior comparison. Motivated by shortcomings in this initial approach, we introduce behavior bounding, an automated model-based approach for comparing behavior that is inspired, in part, by Mitchell's Version Spaces. We show that behavior bounding can be used to compactly represent both human and agent behavior. We argue that relatively low amounts of human effort are required to build, maintain, and use the data structures that underlie behavior bounding, and we provide a theoretical basis for these arguments using notions of PAC Learnability. Next, we show empirical results indicating that this approach is effective at identifying differences in certain types of behaviors and that it performs well when compared against our initial benchmark methods. Finally, we demonstrate that behavior bounding can produce information that allows developers to identify and fix problems in an agent's behavior much more efficiently than standard debugging techniques.

1. Introduction

Over the past few decades, intelligent systems have been asked to perform increasingly complex and mission critical tasks in domains such as medical diagnosis (Shortliffe, 1987) and simulated aerial combat (Jones et al., 1999). Despite a number of successes, these complex agents have yet to become fully integrated into mainstream software. Much of this impasse may be attributable to the fact that developing these agents is often extremely time consuming and expensive.

Development requires three high-level steps: specification, implementation, and validation. The difficulties associated with each step are determined by the properties of the agent and the task it is intended to perform. In this paper, we focus on a class of agents we term *interactive human-level agents*. Such agents are typified by training simulations in which agents participate in mixed human-computer teams to accomplish a particular training objective (e.g., Swartout et al., 2001; Traum et al., 2003; Jones et al., 1999; Rickel et al., 2002). In these domains, the agent plays a role normally fulfilled by an expert human who may not be available for all training episodes. These agents are distinguished by three properties. First, the agent's performance is judged based on its ability to behave as a human expert would behave in a similar situation. Such a design criterion is often particularly important in training simulations where agents operate as part of a mixed human-computer team playing a role that is normally occupied by another person. Second, like humans themselves, interactive human-level agents must interact with an external, and typically very complex, environment in order to perform many of their tasks. Finally, unlike the situation faced in other design problems, complete specifications for correct behavior are often impracticable if not impossible to obtain. This, unfortunately, is a well documented property of many systems built to model human domain experts (e.g., Tsai, Vishnuvajjala, & Zhang, 1999; Weitzel & Kerschberg, 1989; Lee & O'Keefe, 1994; Menzies, 1999). For interactive human-level agents, the specification of how a task should be performed typically comes directly from the human domain expert, and as a result, comparing the agent's behavior with this gold standard is the only way to determine if the design criteria have been met.

A good example of an interactive human-level agent is TacAir-Soar (Jones et al., 1999). 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 be used when there are not enough human participants for a complex exercise, these agents must model expert-level behavior very closely so as to achieve the same training results as if a fully human team was used. Thus, it is not acceptable for the agents simply to achieve correct final states (e.g., by shooting down the enemy planes). Instead, the agent must pursue a trajectory through the state/action space that emulates the human's trajectory (behavior). As in most complex domains, meeting this requirement is challenging because the expert may perform the task differently on different occasions.

For many human-level agents, the development steps of specification and implementation are often woven together during knowledge acquisition—the process through which the developer interviews a human expert to identify and encode the parameters for correct behavior. Often, this process involves exposing the rules or procedures that govern how the expert decomposes a task into a series of goals, subgoals and primitive actions (task decomposition). Once these rules or procedures have been elicited, the developer can encode that knowledge in a form that is usable by the underlying agent architecture.

This traditional approach of knowledge acquisition is rarely free of errors. The process of task decomposition works well enough to identify the relationships between task goals and subgoals that it is considered a useful means of both acquiring and encoding task knowledge (e.g., Lee & O'Keefe, 1994; Yen & Lee, 1993; Yost, 1996). However, at a finer level of granularity, knowledge acquisition is highly prone to errors. In part, this is due to the fact that the human participants are stretched beyond their areas of expertise. For the domain expert, this means communicating how tasks should be performed instead of simply performing them. For the 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., van Lent & Laird, 1999), for the most part they have not been incorporated into widespread use. As a result, developing complex intelligent agents remains a time consuming and difficult process.

A distinguishing characteristic of the work presented here is the previous stated assumption that correct specifications are difficult or impossible to obtain. This is in contrast to the majority of recent agent validation approaches using model checking or temporal logic (e.g., Bordini, Fisher, Visser, & Wooldridge, 2004, 2006; Fisher, 2005). These systems seek to identify implementation errors by proving whether a particular implementation upholds strict logical constraints (specifications). The underlying assumption in model checking is that errors originate in the implementation—not in the specification. If this assumption is violated, the system must be tested against a gold standard of behavior to ensure correctness as the specification cannot be fully trusted. In this sense, the testing methods proposed in the paper can be viewed as a complementary approach for achieving the same objective: a correctly functioning agent.

Our work is further distinguished from typical machine learning approaches because we are interested in creating artifacts that can help a person validate an existing agent's behavior—we do not necessarily need to learn how to produce the behavior. Our approach is intended for applications in which current learning systems are unable to perform well or are untrusted by the end users. We will revisit our distinction from traditional machine learning approaches again in Sections 4 and 10.

1.1 From Manual to Semi-Automated Behavior Comparison

The standard approach to test-based validation requires that both the knowledge developer and the domain expert monitor the agent's behavior in a large number of scenarios (Kirani, Zualkernan, & Tsai, 1994; Tsai et al., 1999). Although standard, it is clear that this approach has a number of significant drawbacks. Principal among these is that the participation of two humans is required to assess the agent's performance in each test. By the time validation takes place, however, gross inadequacies in the agent's behavior will have been corrected. Thus, although it is very likely that some errors will still exist, their manifestations will probably be relatively few and far between. This means that much of the time spent on validation will not be useful for identifying problems in the agent's behavior.

To improve upon the standard validation approach, a semi-automated method that makes more efficient use of the domain expert and the developer's time would be highly desirable as it could substantially decrease the cost of testing. In this paper, we explore the issue of how to meaningfully compare two actors' trajectories through state/action/goal space (i.e., their behavior) given a set of examples.

Comparison, in this paper, simply means identifying how the actors' trajectories are similar or different to one another. Thus, we are interested in a comparison that goes well beyond simply indicating if two actors achieved the same final states. Rather, it should take into account the actions performed and the motivations behind these actions. This could be done simply by comparing observed trajectories directly, or by inferring a general model for the actors' trajectories and comparing these models. In either case, a key challenge is that we are interested in producing artifacts that are easy for a human to interpret and could be used to assist her in tasks such as validation.

The potential uses of behavior comparison extend well beyond agent validation and into many other tasks where humans may want to know how two actors perform tasks differently. Scoring a modified (non-speech based) Turing test, for example, requires humans to perform a comparison between two actors' behavior. Similarly, consider a human supervisor examining a student's performance on a lesson with an intelligent tutoring or training system. The examination and review could be facilitated if the tutoring system were capable of comparing how the student's behavior differed from an internal gold standard and could then relay this information to the instructor in a manner that was easy to interpret. In each of these applications, the basic process for comparing behavior and the artifacts produced remains constant. The differences stem only from the source of behavior (e.g., human or machine, expert or novice) and how the results are used (to identify programming errors, to score a test, or to evaluate a student's performance). For simplicity and cohesiveness, this paper will focus on using behavior comparisons to aid the agent validation problem, but the discussion and results can also be applied to other tasks as well.

1.2 Outline

In the remainder of this paper, we examine two methods for comparing interactive goal-oriented behavior such as that exhibited by human-level agents and their human counterparts. We begin by describing a primitive representation of behavior upon which we can build our comparison methods. Next, we describe a simple sequence-based comparison, but deficiencies with this method lead us to examine more sophisticated model-based approaches.

The main contributions of this paper are fourfold. First, in Section 4, we identify the requirements of a useful comparison system. Then, beginning in Section 5, we describe a novel model-based approach for comparing two actors' behavior. This approach, called *behavior bounding*, uses a hierarchical behavior representation that can be built from observations of human or computer-agent behavior. Third, we demonstrate that behavior bounding meets the requirements of a useful behavior comparison system and support these claims with both theoretical and empirical evidence. Finally, we show that information from behavior bounding's comparison can significantly aid the process of identifying problems in an agent's behavior, thus speeding agent validation by a significant factor.

2. Behavior Traces

At its most primitive, behavior can be represented as a trajectory though state/action/goal space that we will refer to as a behavior trace. A behavior trace is a sequence of tuples $B = ((s, G, a)_0, (s, G, a)_1, \ldots, (s, G, A)_n)$ in which each tuple $(s, G, a)_i$ indicates the environmental state (s), the goals being pursued by the actor (G), and the action being performed (a) at the i^{th} sampling point. The actor's goals are not directly observable and must be explicitly provided by the actor performing the task. Goals are important for our purposes because we are not only interested in *what* the actors do, but we are also interested in the motivation behind their actions.

In this project, we make three main assumptions about the nature of the actor's goals. First, we assume that the actor's goals are part of the actor's internal state. These goals are not simply given by the task description. Although the task certainly informs goal selection, these goals arise from the interactions between the agent's internal desires and the environmental situations encountered during the task. Second, we assume that the actor's goals can change as the environment changes and as the task moves toward completion. This means that goals can be used to structure the agent's task into subtasks and that appropriate goals and subgoals will generally differ during distinct phases of the task. Third, we assume that the actor's choice of goals (and actions) is based upon a static set of knowledge. That is, the agent does not learn.

Note that as we have defined it, the behavior trace does not give complete information about the agent's internal state. Indeed, the actor is likely to perform a potentially large amount of reasoning in order to select G or a. For example, the actor may perform an expected utility calculation or a look-ahead search. However, this process and any information that is not explicitly represented in G or a is completely absent from the behavior trace. Although this provides us with only a limited amount of information with which to perform a behavior comparison, it also ensures that it will be possible to collect behavior from either human or computer agent actors.

Behavior capture is the process of collecting information from an actor to build a behavior trace. As noted above, limiting the information in a behavior trace is critical to ensure that behavior capture is possible. The state and action portion of the behavior trace can be captured simply by observing the actor perform the specified task. Depending on whether the actor is human or a computer agent, the way in which the actor records how goals change during a task will vary. For the computer agent, (G, s) pairs can simply be written to a file during task performance. For a human expert, goal annotations can be made verbally during task performance or immediately following task completion as suggested by van Lent and Laird (1999).

3. Sequence-Based Comparison

A simple approach to comparing the actors' behavior can be performed with the following steps:

Acquire a set of behavior traces from the human expert and the agent for the specified task. These sets, H and A, represent the human expert's and agent's behavior respectively over a number of different trials.

Extract relevant symbols from the behavior traces. Some information gathered through observation may be irrelevant for detecting errors. For example, if the human expert's behavior never changes given different values of the state symbol z, then z is likely to be irrelevant for detecting errors. In this step, the salient symbols from the sets H and A are used to create two new sets of sequences H^* and A^*.

Compare each sequence $a \in A^*$, to the contents of H^*. Compute the minimal number of edit operations (insert, delete, modify) that would be required to transform a into h, where h is the sequence in H^* that is initially most similar to a. Each edit operation indicates a potential error.

Report all deviations (after removing any redundancies) between the human's and agent's behavior. This report summarizes all potential errors.

This simple approach performs a more detailed analysis of behavior than simply checking that the agent and the expert reach the same final (goal) state. In this way, the agent's externally observable behavior as well as some aspects of its internal reasoning process can be inspected to ensure consistency with the human expert's. In addition, this methodology has the ability to identify a large number of possible errors because it has access to all the salient properties of the behavior trace. However, this simple approach also suffers from a number of potentially serious flaws.

1. The actors' behavior is represented as a set of sequences. As the complexity of the domain increases it is likely that two effects will be noticed: the average length of sequences in H^* and A^* will grow (i.e., the complex tasks will take longer to solve), and these sequences will be composed of a larger number of symbols (e.g., the state space will become richer). The number of distinct sequences with lengths between l_{min} and l_{max} and composed of s symbols grows as $\sum_{l=l_{min}}^{l_{max}} s^l$. Thus, enumerating this space is likely to be infeasible. Moreover, because interactive human-level agents can typically solve problems in a number of different ways, and typically operate within complex domains, it is likely that the sequential approach described in this section will be particularly susceptible to this effect.

2. The sequence based comparison fails to make any assumptions about how the actors' behavior may be constrained. That is, the sequential behavior representation provides no method for expressing *a priori* knowledge about how symbols can be placed relative to one another within a particular sequence. Instead, the representation is completely unconstrained; sequences of length l can be constructed by making l independent symbol selections. Although this makes it possible to use this simple approach with any variety of behavior (even behavior that is completely unstructured), it also makes it impossible to leverage regularities that might exist in a large classes of goal directed tasks (such as the fact that unlocking a door must always be accomplished before the door is opened).

4. Model Based Approaches

To improve upon the simple sequence-based method of error detection, we propose a comparison method that leverages an abstract representation of the actors' behavior. We call such methods *model-based* because they do not compare instances of the actors' behavior directly (as the simple sequential approach would). Instead, these methods compare abstract representations of the actor's behavior (models), to identify similarities and differences in the underlying behavior. Central to any such approach are the considerations that influenced the model's design. Our choice of models is guided by the following design requirements:

Low Complexity The behavior model must be significantly less complex than the representations that define the agent itself. If this requirement is violated, two problems may result. First, constructing the model (either by hand, or automatically through some observational framework) is likely to be as difficult as constructing the agent's knowledge base. Second, understanding the model and the behavior it represents is likely to be no easier than examining the agent's internal representation. If the comparison is being used to validate the agent's underlying knowledge base, this is clearly undesirable as it results in a recursive validation problem. However, we can achieve this low complexity requirement by using a model that represents behavior at a relatively high level of abstraction compared to the agent's internal implementation.

Low Human Effort The human effort required to build the behavior model must remain low. We have argued that one of the main uses of the behavior comparison would be to reduce the cost of validating a human-level agent. If the low human effort requirement is violated, the original validation costs (due, for example, to the time requirements of examining numerous test scenarios) have simply been replaced with new costs, resulting in no net benefit. We can achieve this low cost requirement by using an automated system to build behavior representations from a series of observations with little or no human supervision.

Compatibility It must be possible to build and use the behavior model with both human actors and software agents. As we discussed in previous sections, behavior comparison has a number of potential applications, but many rely on being able to examine both human and software agent behavior. Thus, the contents of our model must be limited to data that can be collected from either of these types of participants. In Section 2, we described how behavior traces could be collected from both human actors and computer agents. As a result, we can achieve this requirement by using a model that is built from behavior traces.

Efficiency The computational costs associated with building and using the model must not become infeasible as the complexity of the domain increases. Although a primary motivation of automated behavior comparison is to replace human effort with computational effort, we must be careful to construct the model in such a way that it does not become impossible to use. We can achieve this requirement by using an abstract model of the actors' behavior that does not grow directly as a function of the number of behaviors it encapsulates.

Efficacy A good model must be effective at identifying similarities and differences between two actors' behavior. This is perhaps the most basic requirement we have presented. However, the desire for an effective model that captures all the subtleties of an actor's behavior is likely to be in direct conflict with the previously presented requirements. As a result, a good model must balance its need to represent actors' behavior precisely and thus to be able to distinguish all similarities and differences in their behavior with the other overall needs. Unfortunately, there can be little *a priori* assurance that a particular model will be effective. This requirement must be addressed through theoretical and empirical testing once the model has been implemented.

Note that unlike any traditional machine learning tasks, we do not necessarily need to produce a model that can be used to *perform* the task. That is, we do not need to learn a policy or a set of plan operators. As described above, there is a trade-off between the model's efficacy and its complexity. At one end of this spectrum are executable models of the task. Here, efficacy is maximized, but the model would be necessarily complex and would likely be more difficult for a human to use to validate behavior than if they were looking directly at hand coded rules or procedures. Such models are certainly valuable if the goal is to learn behavior directly from a set of examples, and a variety of approaches have been pursued in the machine learning literature; the most closely related are discussed later in Section 10. Our approach, however, attempts to target a different point in the efficacy/complexity spectrum where the model cannot perfectly describe many complex tasks, but as a result the model can be examined much more quickly than the agent's internal implementation. Thus, while the standard approach in machine learning literature is to empirically evaluate a learned model by comparing it to an optimal model or to a hand-coded model, here we are interested in something else: namely, whether our model can maintain efficacy in complex environments and whether it can improve a person's ability to quickly uncover and fix problems in existing agents. In Sections 8.2 and 9 we examine these issues.

4.1 Model-Based Diagnosis

Prior work in model-based diagnosis (e.g., Anrig & Kohlas, 2002; Lucas, 1998) has examined how to detect errors given a model of correct behavior. In general, however, the models in these systems are relatively complicated and intended to identify problems with mechanical or solid state devices as opposed to software agents. The CLIPS-R (Murphy & Pazzani, 1994) system was designed expressly to ensure correct software agent behavior, and bears some similarity to our approach.

In CLIPS-R, the behavior model consists of a set of tuples (S_i, C_{S_f}, C_E), each of which specifies the initial world state (S_i), a set of constraints describing acceptable final world states (C_{S_f}), and execution constraints (C_E) which must be met as the task is being performed. Final state constraints indicate facts about the environment or the agent that must be either true or false once the task is complete (e.g., `(not (gas-empty car))`). Note that the final state constraints define a behavior model in the classical planning sense; there is no description of what sequence of events should lead to the final state. This information is provided by the execution constraints (C_E), which are represented as a finite state machine describing acceptable orderings of the agent's observable actions. Execution constraints can be used to describe relationships between these actions. For example, a constraint might specify that the action `unlock-door` should always proceed `open-door`. Superficially, the requirements for the CLIPS-R approach seem relatively simple to meet. However, two serious problems exist.

First, specifying the exact set of execution constraints required for correct operation is very similar to writing the conditions of rules. If the execution constraints govern behavior at a very fine level of granularity, it is likely that they will be similarly difficult to design and validate as the agent's rule base itself (a recursive validation problem). In this case, the requirements of low complexity and low human effort would be violated. On the other hand, if they constrain behavior at a higher level of granularity, such as the task level, the efficacy requirement is called into question: will they be powerful enough to work in the complex environments of human-level agents?

A second serious problem arises because the CLIPS-R approach provides little guidance as to how to determine appropriate constraints, especially appropriate execution constraints. The benefits of the approach hinge completely on the developer's ability to enumerate adequate and appropriate execution constraints for any particular task. Yet if the developer can enumerate the constraints required to judge whether the agent's behavior is correct, why were they not included in the agent's knowledge base directly?

It should be noted that although the problems mentioned above may be encountered when CLIPS-R is used with any particular agent, they are likely to become most obvious (and problematic) as the complexity of the agent and domain increases. As already noted, these are exactly the types of agents and environments that interest us, and so the concerns raised above are particularly salient for our work with interactive human-level agents. In contrast, the original CLIPS-R work (Murphy & Pazzani, 1994) examines the system's ability to correctly identify flaws in two very simple agents whose knowledge bases contain nine and fifteen rules respectively. Both agents examined in the CLIPS-R work performed tasks that were more akin to classification than they were to the highly interactive tasks that interest us.

5. Behavior Bounding

As an improvement to CLIPS-R and to the simple method presented in Section 3, our approach to behavior comparison, called behavior bounding, automatically and efficiently builds concise high-level models of both the human expert's and agent's behavior by examining behavior traces to meet the first three requirements described in Section 4. The human expert's behavior model is used to identify boundaries on acceptable behavior in a manner reminiscent of Mitchell's Version Spaces (Mitchell, 1982). Potential errors are reported by comparing the model of agent behavior to these boundaries. Behavior bounding can be used to identify programming errors in the agent's knowledge base and can also identify discrepancies between the expert's explanation of how the task should be performed and how the expert actually performs the task. This is in contrast to a high-level model built similarly to the agent's knowledge base (as, presumably, in CLIPS-R) using indirect information such as interviews to determine what constraints should be met during task performance.

5.1 The Hierarchical Model

Behavior bounding leverages the assumption that although knowledge acquisition is highly prone to errors with respect to the details of how a task should be performed, high-level information (specifically general relationships between goals, sub-goals and primitive actions) is much more reliable. Behavior bounding's hierarchical behavior representation is inspired by the hierarchical models used in And/Or trees, HTN planning (Erol, Hendler, & Nau, 1994) and GOMS modeling (John & Kieras, 1996) to encode the variety of ways in which particular tasks can be accomplished. Conceptually, behavior bounding encodes three relationships. First, it identifies decomposition relationships between goals, sub-goals and primitive actions. Second, it identifies ordering relationships between nodes in the hierarchy. Finally, behavior bounding identifies how goals and actions are instantiated by saving generalized parameters (i.e., features from the internal or world state that are directly associated with the goals and actions begin pursued).

The hierarchical behavior representation (HBR) used in our approach is an And/Or tree with binary temporal constraints representing 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 indicate the set of sub-goals or primitive actions that are relevant to accomplishing the specified goal.

Figure 1 illustrates a small subsection of a hierarchical behavior representation. Goal nodes are drawn with ovals and primitive actions with rectangles. And constraints are represented in the standard fashion with an arc across all child nodes; temporal constraints

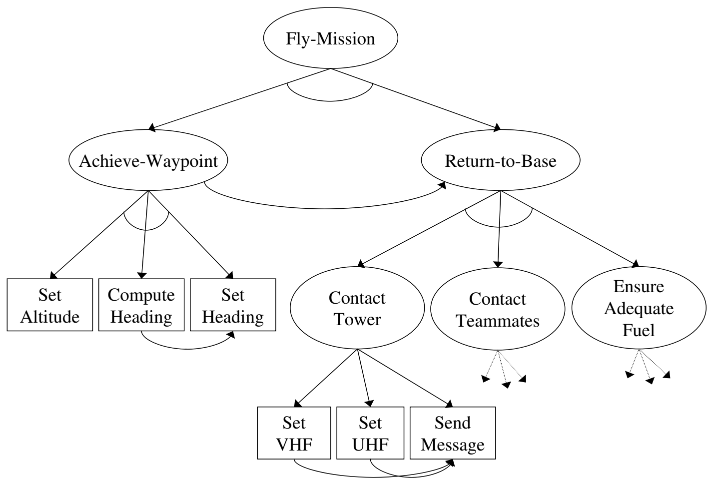

Figure 1: A Hierarchical Behavior Representation

are represented with directed arcs between sibling nodes. Note that total order between siblings is possible but not required by the representation. The semantics of OR nodes in our representation does not necessarily indicate that only one subgoal (or action) is required to accomplish a given goal. Rather, the OR node indicates simply that the complete set of subgoals (or actions) is not always required to accomplish the task. Thus, the semantics of OR nodes does not preclude the use of temporal relations; they merely state the order that multiple goals/actions occur if indeed more than one is pursued.

The HBR can be viewed as a simple constraint model based on observations of the actor's behavior. It encodes some of the same relationships that Fisher uses in his temporal logic models of agents (Fisher, 2005): namely step rules (what goals/actions to expect next); and sometimes rules (what goals/actions to expect in the future). As a result, the HBR could be used as a source for the types of temporal logic constraints required for model checking when (as in the case of human-level agents) the expert is not capable of providing such logical constraints directly.

5.2 Building the HBR from Behavior Traces: An Overview

In Section 6 we present a detailed explanation of how a HBR is acquired from behavior traces along with the underlying algorithm. Here, we present a conceptual overview of this process by describing how the partial behavior trace on the left-hand side of Figure 2 is used to build the HBR on the right side of the same figure.

Initially, we begin with an empty HBR. The behavior trace (Figure 2, left hand side) 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 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 processed, 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. Note that if an action or subgoal is encountered in multiple contexts (as a descendant of two or more distinct parents) the HBR will create a node for each such context. This is appropriate as the parameters associated with the goal/action and its interaction with sibling goals/actions will likely depend on its higher-level context.

This generation process results in the HBR on the right-hand side of Figure 2 (note that the parameters associated with each goal and action, and listed in the behavior trace segment, are not displayed to improve the clarity of the figure). Here goal nodes (ovals) with children are all of type And. In addition, all siblings are totally ordered as indicated by temporal constraints (directed arcs between siblings). The highly constrained nature of this HBR (And goals and total ordering) is typical of representations built from a single behavior trace. As more behavior traces are used to generate the structure, the HBR is generalized to cover all input observations.

At a structural or topological level, generalization occurs in two ways. The first is when an And constraint is turned into an Or constraint. In our example, `Achieve-Waypoint` is an And goal because every time it was observed, it was completed by pursuing all three of the subgoals: `Set-Altitude`; `Compute-Heading`; and `Set-Heading`. If a second behavior trace indicated that `Achieve-Waypoint` was successfully completed by performing only the subgoal `Set-Heading`, then `Achieve-Waypoint` would become an Or node to correspondingly indicate that it does not require all subgoals to be accomplished.

Similarly, generalization of binary temporal constraints occurs as needed to represent the observed orderings of goals and actions. Returning to our example in Figure 2, `Achieve-Waypoint` was observed to occur only once. Thus, its representation in the HBR indicates a total order between its three subgoals. If `Achieve-Waypoint` were performed a second time with a new sequence of these same three subgoals, the ordering constraints within the HBR would change. For example, if `Achieve-Waypoint` were performed by pursing: `Compute-Heading`; `Set-Altitude`; and `Set-Heading`, in that order, the temporal constraint between `Set-Altitude` and `Compute-Heading` would be removed. This process of building the HBR and the underlying algorithm will be discussed in more detail in Section 6.

Generalization also occurs for the parameters associated with each goal or action, effectively expanding the set of parameters associated with each node as more and more obser-

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
    ...

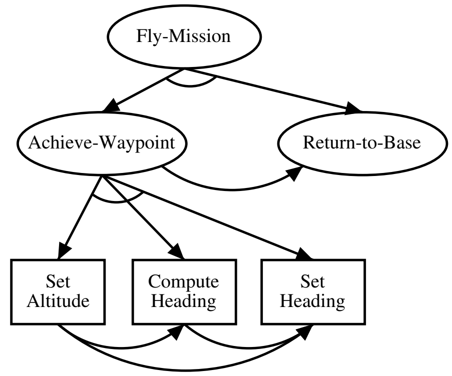

Figure 2: Constructing the hierarchical behavior representation from a behavior trace

vations are made[1]. Consider Figure 2 where the parameter associated with `Set-Altitude` is `30000`. If we later see `Set-Altitude` performed with the parameter `20000`, the HBR will contain the generalization of these two observations, namely that `Set-Altitude` can have parameters in the range `20000`–`30000`. Each parameter associated with a goal or action is generalized to cover observations in the behavior traces. For numerical parameters, generalization is performed by expanding the acceptable range to include the new value. For symbolic parameters, generalization is performed by adding the new symbol to a set of acceptable values.

5.3 Representational Simplicity

The HBR discussed above is clearly a much less complex representation of behavior than most agents' underlying knowledge base. Indeed, the hierarchical structure ensures that constraints cannot be formed between arbitrary goals or actions. This property also means that the HBR may be less complex even than the model used by CLIPS-R, which allows an arbitrary finite state machine to describe the acceptable sequences of external actions.

Behavior bounding ensures a high-level model of behavior by abstracting away internal data-structures the agent may use to perform the task if they cannot be represented by the hierarchy. While it is possible to store arbitrarily complex information in the HBR, it is unlikely to happen in practice. Consider, for example, depth first search which uses an an open list to discriminate between alternative behaviors. While the final result of the search (a goal or action) is naturally captured by the HBR, forcing the HBR to capture the details of the search is impractical as it requires pushing all information captured by the open list into the goal hierarchy.

More specifically, consider an agent using search to select between two potential actions: `Set-Altitude`; and `Set-Heading`. First, note that the search process itself would only be represented in behavior bounding's HBR if the agent explicitly made searching a goal. However, even if `Search` was an explicit goal, information about the open-list (states that still need to be tested) would only be available to the HBR if it was made an ex-

1. For the purposes of this paper, parameter generalization is less interesting than structural generalization. We include this brief discussion mainly for completeness.

plicit parameter of the goal. Even this formulation, however, would leave a large amount of information about the search process unrepresented in the HBR. Specifically, because search is encapsulated as a single goal without any substructure, it would be impossible to determine the manner in which various search nodes were visited. In order to represent this information, we would need to push all the relevant data structures (in this case the open-list) into the goal hierarchy itself. Thus, we would need to create explicit goals for each (*state*, *open-list*) pair. This approach of pushing arbitrary information into the goal hierarchy is clearly both undesirable and unlikely to occur frequently in any well designed agent. Thus, we can be reasonably certain that behavior bounding's HBR will always be a high-level, abstract, representation of the agent's (or actor's) behavior.

The representational limitations of the HBR leads 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, nor is it sufficient to generate behavior. Most human-level agents rely on intermediate data-structures that are not available through the environment or through the structure of the goal hierarchy (for example agents that use look-ahead to select the next goal or action, or perform an expected utility calculation). Rather, our hypothesis is that the representation provided by behavior bounding is sufficient to identify a large class of errors in agent behavior without sacrificing efficiency. Moreover, we hypothesize that behavior bounding can help identify potential problem spots in the agent's knowledge (e.g., the ordering of actions in a specific goal) even if an exact error cannot be identified.

5.4 Representational Assumptions

In contrast to the behavior representation used for the simple comparison described in Section 3, the HBR makes three strong assumptions about the organization of the actors' knowledge and the effects of this organization on the actors' behavior. These assumptions increase the efficiency and efficacy of error detection for certain types of human-level agents.

The first assumption used by behavior bounding is that the actor's goals are organized hierarchically, with more abstract goals located toward the top of the tree. Hierarchical task structure is exploited by a number of agents and agent architectures, thus this assumption is not particularly limiting. 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. The goal stack assumption implies that concurrent goals (two or more goals simultaneously pursued at the same depth of the hierarchy) cannot be modeled explicitly by the HBR. One way to circumvent this limitation is to implement concurrent goals as nested goals. Because our test architecture (Soar) does not directly support concurrent goals, this is the approach typically taken to achieve such behavior. As we will see in Section 8.2.5, this approach does allow us to create and use a HBR but may also result in some representational problems. The hierarchical goal assumptions described above provide the important benefit of constraining acceptable orderings of goal and actions that an agent may pursue. This property will be analyzed in more detail in Section 8.1.

The second assumption leveraged by behavior bounding relates to the independence of goals. In the HBR, 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 is easy to constrain the way 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 to build the model. Consider a task that requires completing two goals, each of which could be fulfilled in four distinct ways. The behavior is represented as an ordered pair (a_1, a_2) indicating the action taken to fulfill goals one and two respectively. A sequential representation that makes no assumptions about goal independence (such as the one described in Section 3) would require sixteen distinct observations to cover the acceptable behavior space (one for each distinct (a_1, a_2) pair). In contrast, behavior bounding would only require four observations so long as the set of observations included every possible value of a_1 and every possible value of a_2[2]. This impact on efficiency is significant and is the direct result of leveraging the assumption about how goals are likely to add regular structure to an actor's behavior.

Finally, recall from Section 5.1 the third assumption upon which behavior bounding is built. This is that knowledge acquisition is relatively reliable for correctly identifying the general goal/subgoal relationships an expert uses to perform the target task even though this same process of knowledge acquisition is very prone to errors when attempting to identify all the rules necessary to encode the task. This assumption provides a justification for using a behavior representation that focuses on the relationships between goals, subgoals and primitive actions while purposefully neglecting much of the internal information an actor may use to select her behavior.

The net effect of building the HBR based on these assumptions is a model that meets the criteria set forth in Section 4. The model is likely to be much more concise than the agent's implementation (low complexity)—we are not learning complete plan operators, but instead a generalization of the actor's trajectories through goal/action space. In addition, the HBR can be generated automatically by examining an actor's behavior traces thus meeting our second requirement (low human effort). Because the behavior traces can be captured from either human or computer agent actors, the HBR meets the third requirement (compatibility). In the following sections, we will present the method behavior bounding uses the HBR to perform comparisons. In addition, we will examine the remaining two requirements of an ideal model-based approach (efficiency and efficacy) in detail.

6. 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

2. Thus, if $a_1, a_2 \in \{1, 2, 3, 4\}$ then the pairs $(1, 1)$, $(2, 2)$, $(3, 3)$, $(4, 4)$, would be sufficient to cover the acceptable behavior space in behavior bounding but not in the sequential representation.

represents behavior efficiently and thus meets the fourth requirement (efficiency) outlined in Section 4.

```
CREATE-HIERARCHY(B, H)
W ← empty tree
lastStk ← NIL  // previous goal/action stack
for  each (s, G, a) in B
do
    for i = 0 to length[lastStk]
    do
        if GOAL-COMPLETED(lastStk[i])
          then h_g ← FIND-NODE(H, lastStk[i])
                if h_g = NIL
                  then
                        ADD-SUBTREE(H, PARENT(lastStk[i]), lastStk[i])
                  else
                        GENERALIZE(H, h_g, W, lastStk[i])
    for  each g_i in [G, a]
    do
        p_g ← PARENT(g_i)
        w_g ← FIND-NODE(W, p_g, g_i)
        if w_g = NIL
          then
                w_g ← ADD-NODE(W, p_g, g_i)
                CONSTRAIN-CHILDREN(W, p_g)
          else
                if OUT-OF-ORDER(W, p_g, w_g)
                  then UPDATE-CONSTRAINTS(W, p_g, w_g)
                GENERALIZE(w_g, g_i)
    lastStk ← [G, a]
return H
```

Figure 3: The Create-Hierarchy algorithm

In Section 5.2 we presented an overview of the process behind building the HBR from a behavior trace. The Create-Hierarchy algorithm (Figure 3) specifies this process explicitly. The algorithm takes two arguments as input: B, a behavior trace; and H, a HBR representing previously observed behavior (or nil if no behavior has yet been observed). Create-Hierarchy returns a new HBR covering the behavior in H and the new observation B. Thus, calling this procedure with a single behavior trace B and $H \leftarrow$nil generates a hierarchical representation of a single behavior trace by examining the way in which goals decompose into subgoals and primitive actions during task performance. Iteratively calling Create-Hierarchy with different behavior traces will augment and generalize H until it covers all of the example traces. This algorithm can be executed in $O(lN^2)$ time where l

is the (maximum) length of the behavior trace and N is the number of nodes in the goal hierarchy.

Classifying the sample complexity of our hierarchical representation is straightforward. From Haussler's equation (Haussler, 1988; Mitchell, 1997), we know that the number of training examples required for a consistent learner to learn any target concept (with probability $(1 - \delta)$ and error bound ϵ) in its hypothesis space (H) is m where:

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

The HBR can be viewed as an ordered tuple $P = (p_1, p_2, \ldots, p_{|N|})$ where each p_i is itself a tuple containing the type of the node i (either And or Or) as well as a list $L = (l_1, l_2, \ldots, l_{|N|})$ such that $l_a = 1$ iff g_i is ordered before l_a. Note that since ordering constraints only occur between siblings, the length of the list L would only need to be length $|N|$ in the degenerate case. The size of this hypothesis space is bounded by $2^{|N|+|N|^2}$ in the worst case, but based on the shape of the hierarchy may be much smaller. Substituting the size of the hypothesis space back into Equation 1 we find that m does indeed grow polynomially:

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

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.

7. Identifying Errors

In general, we can view a behavior comparison method as an algorithm which divides the space of possible behaviors into two regions: behaviors that are likely to be consistent with the expert, and behaviors that are likely to be inconsistent with the expert. The simple comparison method described in Section 3 does this by enumerating consistent behaviors. The model used in behavior bounding, however, allows us to divide the space of possible behaviors more efficiently and into more refined regions without enumerating their contents. Intuitively, the idea is to organize HBRs into a lattice; individual points in this lattice are then used to define boundaries between different quality behaviors in a manner reminiscent of Mitchell's Version Spaces (Mitchell, 1982).

Recall that the hierarchical behavior representation is a hierarchy with nodes corresponding to goals, subgoals and primitive actions. Nodes are linked hierarchically based on the goal/subgoal decomposition relationships observed in behavior traces. The HBR can be viewed as consisting of two parts:

1. The *basic structure* which is a hierarchy of nodes that are labeled with the names of goals, subgoals and actions and are connected by parent/child relationships in a manner that corresponds to the observed behavior.

2. A *set of constraints* that are imposed upon the nodes in the basic structure. Constraints include the And/Or typing of nodes, binary temporal constraints, and constraints on the allowable parameter space of any goal, subgoal, or action.

Because the constraints are formed through a specific to general learning algorithm, the generalization process creates a lattice of HBRs that are related in the following manner: 1) they share the same basic structure; and 2) they differ in the specificity of their constraints. Thus, the hierarchical behavior representation allows us to define an ordering from specific to general over the space of behavior hierarchies by starting with a maximally constrained hierarchy (at the top) and iteratively removing constraints until none remain.

Behavior bounding leverages this ordering over hierarchies to efficiently partition the behavior space into different regions. The process begins by using traces of expert behavior (the specification) to create a corresponding HBR. Once created, we can identify the node it occupies in this ordered space (call this node `A` in Figure 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 4 (labeled `B`). It contains the basic structure (goal/subgoal relationships) 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 4 labeled as neither more nor less specific than `A`).

Together, the upper and lower boundaries create three regions in the behavior space. Nodes that are a specialization of the expert's behavior (above 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 between the upper and lower boundary nodes corresponds to behavior that is likely to be incorrect but perhaps with a lower probability than the region below the lower boundary node[3].

Mitchell (1997) defines the version space as a subset of hypotheses (from a hypothesis space) that are consistent with a given set of training examples. By ordering the hypothesis space from specific to general, Mitchell's learning algorithm (Mitchell, 1982, 1997) identifies the version space without enumerating its contents. Instead, the version space is represented by the concepts (hypotheses in the ordered hypothesis space) that form its upper and lower

3. 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 on how the HBR corresponding to the lower boundary node was formed (see Section 11 for an alternative method).

boundaries. These are the S-Set and G-Set that specify the most specific hypotheses and most general hypotheses in the version space respectively. As training examples are obtained, the S-Set becomes progressively more general while the G-Set becomes increasingly specific until both converge on the correct hypothesis.

Just as Mitchell's S-Set and G-Set are used to delimit a set of consistent hypotheses without enumerating them, the upper and lower boundary nodes in our approach serve a similar purpose. The upper boundary node (UBN) plays a similar role to the S-Set. However, while the S-Set is used to incrementally converge on the correct hypothesis (and in doing so becomes increasingly general), the upper boundary node is viewed *as* the correct hypothesis. Thus the UBN's value is in delimiting the portion of the lattice that is consistent with its specification. The lower boundary node, on the other hand, plays a similar role to the G-Set. But, while the G-Set identifies hypotheses inconsistent with training data, the lower boundary node simply identifies HBRs that are not in the same lattice because they have a distinct topological structure.

Once these boundaries have been established, we can quickly determine whether any arbitrary HBR is a specialization of either boundary node. This analysis, which can clearly be done in polynomial time with respect to the number of distinct goals, subgoals, and actions, allows us to quickly determine the degree to which the behaviors of two actors are, or are not, consistent with one another. The inconsistencies uncovered in this process form the basis of behavior bounding's error report and can be displayed in either a standard text format or visually using a graphical user interface. For the remainder of this paper, we will use terminology appropriate for comparing two actors playing the roles of either *expert* or *novice*. The actor referred to as the *expert* represents the correct behavior specification. The actor referred to as the *novice* we expect to exhibit partially incorrect behavior. As described in Section 1, these roles could be played by either software agents or humans depending on the situation at hand.

8. Error Identification Efficacy

At this point, we have provided a good deal of support for behavior bounding and its HBR by presenting analytical arguments on its behalf. The final criteria that must be addressed is its efficacy with respect to identifying errors. To do this, we will examine two components of the HBR. First, we will provide analytic results indicating the effectiveness of the unconstrained hierarchical representation (the lower boundary) at identifying behavior that is known to be incorrect. Second, we will provide empirical evidence that behavior bounding as a whole is effective at distinguishing between correct and incorrect behavior.

8.1 The Lower Boundary Node

At first glance, it is not obvious how much behavior can be classified by the lower boundary node. Without And/Or constraints or binary temporal constraints, the lower boundary node only specifies which subgoals belong to which goals. Through this specification, the lower boundary node constrains the set of allowable goal/sub-goal/action sequences. The effectiveness of this simple constraint mechanism 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,

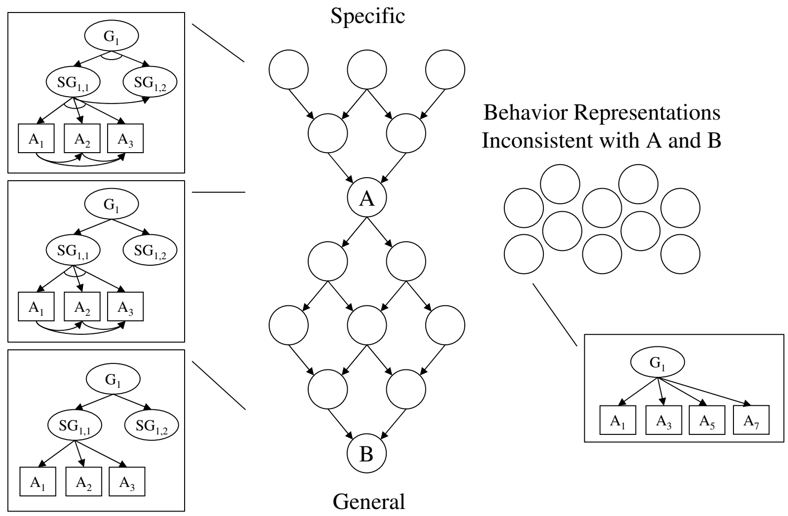

Figure 4: Imposing Order on the Behavior Space

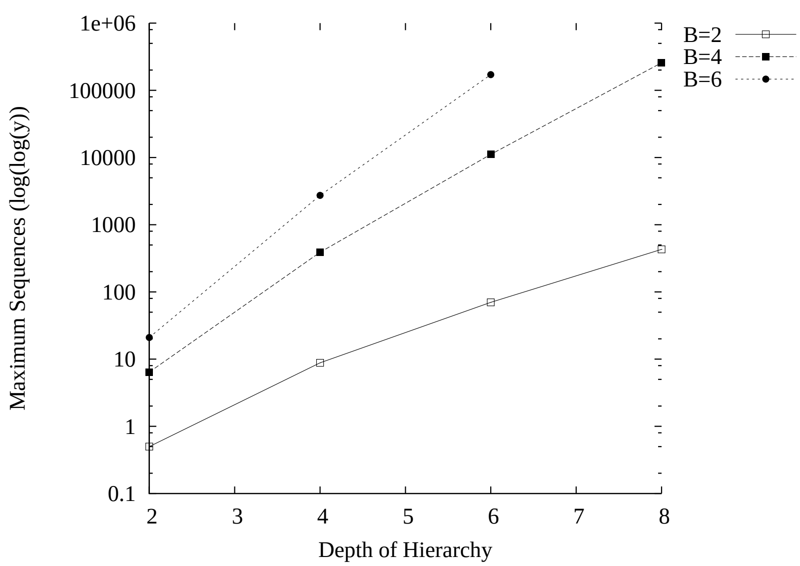

Figure 5: Filtering Capability of the Lower Boundary Node

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, there are $b!^s | s = \sum_{j=0}^{d-1} b^j$ possible sequences that are consistent with the goal decomposition of the unconstrained hierarchy. In contrast, there are $b^d!$ sequences in which the symbols may be placed without necessarily conforming to the unconstrained hierarchy. For hierarchical structures of reasonable size, this makes the lower boundary node effective at filtering an exponential number of potential behavior sequences. For example, in a small hierarchical structure of depth 4 and branching factor 2, only 1 in approximately $6.4 \cdot 10^8$ of the possible sequences of length 16 are consistent with the goal decomposition specified by the unconstrained hierarchy. Figure 5 illustrates the filtering capability of the lower boundary node. The x-axis of the figure indicates the depth of the hierarchy and lines are plotted for branching factors 2,4, and 6. The y-axis indicates the ratio of possible sequences accepted by a goal hierarchy to the number of total possible sequences for an unconstrained symbol set of the same size; note that the y-axis is doubly-logarithmic ($\log\log(y)$ is plotted).

Although the lower boundary node is an extremely simple data structure, the information it stores is of significant value. Used alone, it can identify a very large (exponentially increasing) number of behavior sequences as inconsistent with the expert's goal decomposition structure and therefore incorrect.

8.2 Empirical Evaluation

Our empirical study has two aims. First, we want to determine whether behavior bounding identifies errors in agent behavior well enough to be considered useful for the purposes of validation. Second, we want to compare behavior bounding's effectiveness to that of the simple sequential approaches described in Section 3. To this end, we implemented behavior bounding along with two versions of the sequential approach to serve as benchmarks. The first benchmark, M_1, extracts the sequence of actions $A = (a_0, a_1, \ldots, a_n)$ from the behavior trace $B = ((s, G, a)_0, (s, G, a)_1, \ldots, (s, G, a)_n)$ while the second benchmark, M_2, extracts the sequence of goals $G = (G_0, G_1, \ldots, g_n)$ from B. In both cases comparison is performed by computing the minimal edit distance between two behavior traces. Remember that the sequential methods are not particularly efficient representations; they can grow exponentially in the length of the behavior trace and have an exponential sample complexity. However, for this same reason, they do make interesting benchmarks of efficacy.

Performance is judged based on ability to: 1) correctly identify errors in agent behavior; 2) identify *all* errors that have occurred; and 3) produce minimal amounts of spurious information when reporting errors. To make such an assessment, we must compare the errors identified by the automated comparison to a record of errors that were manually identified and known to have actually occurred. This requires a manual inspection of the behavior traces and a taxonomic classification of possible differences. In the following subsections we begin by describing how errors are classified and then move on to discuss the experimental method and assessment process in detail.

8.2.1 Behavioral Differences

At the simplest level, all differences (potential errors) can be identified by a single discrepancy between two particular symbols in the behavior traces such as a particular pair of goals or actions. This type of mismatch can occur in one of three ways. As before, we will refer to desired behavior as being captured in the expert's behavior traces, while untrusted or imperfect behavior is captured in the novice's behavior traces.

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. For example, consider an agent flying a tactical military aircraft patrolling the air space between two waypoints. Assume the specification for correct behavior dictates that agent travel between the way-points until an enemy aircraft is spotted at which point the agent should contact the command center to receive clearance to engage the enemy. In this situation, an error of commission would occur if the agent contacts his wingman instead of the command center and then proceeds to enter the engagement.

Omission If the expert's behavior trace contains a goal or action symbol where there is no corresponding symbol in the novice's behavior trace, this error is an omission. Following the example above, an omission would occur if the agent immediately begins to engage the enemy without interjecting any other substitute goal or action to replace the missing call to the command center.

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. An intrusion would occur if the agent contacts the command center and receives clearance to engage the enemy but then proceeds to continue to the waypoint before returning back to engage the enemy.

In our experiments, it was often relatively straightforward to classify errors into these three categories. However, in some situations there were enough differences between the two actors' behavior that it was difficult to determine whether a deviation was a commission or one of the other forms. In such situations, we marked the error as belonging to either category and considered it acceptable for a comparison method to identify it as either form.

When more than one of the simple errors listed above occurs, it may be possible to identify a relationship between them. We call such related errors compound errors and note that uncovering a single compound error is preferable to identifying many simple errors because the compound error is a more concise description of the underlying problem. Note that clearly we cannot consider all possible relationships between multiple errors as this would have problematic computational implications. Rather, we are interested in relationships that occur frequently in practice. We identify two such compound errors. The first is a *misplacement error* in which two goal or action symbols are transposed in the novice's behavior trace; often this is due to incomplete specification of the constraints for one or both of the goals or actions that take part in the error. The second is a *duplication error* in which one or more goal or action symbols reoccurs inappropriately. In computer agents,

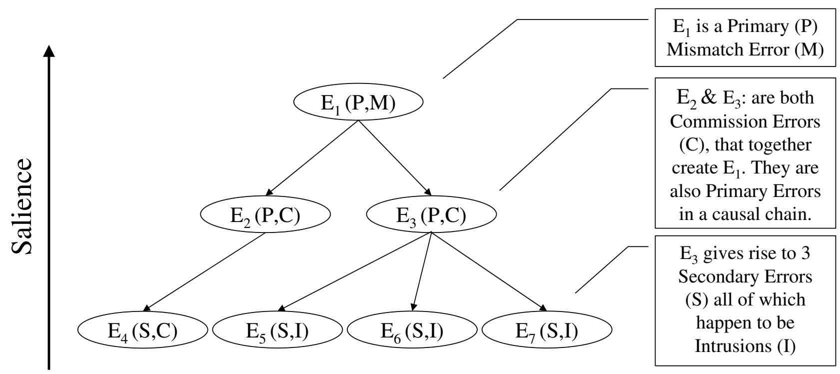

Figure 6: Multiple related errors result in a salience hierarchy

this type of error often occurs because the termination condition for a particular goal or action is incorrectly specified.

Errors can also occur among subsequences in the behavior trace. 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: a *primary error* is the first in a causally linked sequence of errors, *secondary errors* are subsequent errors in such a sequence. Although problems in their own right, secondary errors can be corrected simply by correcting the primary error. Often these occur because a higher level goal was incorrectly selected and naturally led to an entire sequence of incorrect behavior.

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 than 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. Figure 6 illustrates this relationship. Towards the top are primary compound errors and toward the bottom are secondary individual errors. Correcting an error at any level in the hierarchy will also resolve all descendant errors.

8.2.2 Method

Ideally, an empirical evaluation would directly examine how much human effort is saved by using the behavior comparison methods during the development of a number of complex human-level agents. However, developing the complex agents we're interested in is a time consuming task and developing multiple independent versions is beyond the scope of this experiment. Instead, we have selected an approach that identifies the effectiveness of error detection methods without directly examining development time. Using our method, 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,

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. Construct individual experiments by partitioning behavior traces into multiple groups.
7. Evaluate how well each error detection method identifies the cataloged errors.

Figure 7: An overview of the steps in our evaluation process

we can obtain a 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 seven high level steps outlined in Figure 7 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*, E, whose behavior we will attempt to reproduce. The idea of replacing the human expert with a 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 a means of determining how the expert's and novice's behavior differ that might not 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 attribute is important for conducting performance assessments.

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 of our 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.

Our expert-level agents, as well as the novice agents described below were implemented with Soar (Laird, Newell, & Rosenbloom, 1987), a forward-chaining rule based system. Soar provides natural constructs for defining the goal-subgoal relationships required by behavior bounding. In addition, Soar provides a programming interface that allows behavior traces to be captured easily. Although Soar is naturally compatible with behavior bounding, it is by no means the only agent architecture that fits this criteria. Most rule based systems can use task decomposition as a basis for problem solving even if the goal hierarchy must be implemented in the agent's working memory. Such an agent design is easily done in CLIPS (Giarratano & Riley, 1998) as demonstrated by Wallace and Laird (2000). Apart from rule-based systems, many other agent architectures allow developers to define an agent's knowledge base and behavior using task decomposition relations. Two such examples are PRS (Ingrand, Georgeff, & Rao, 1992) and PRODIGY (Veloso et al., 1995).

Given the expert-level agent (E), we begin the second step by constructing novice agents ($N_0, \ldots, N_n$) which are partially correct implementations 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 differences arise because the novice-level agents do not have the same knowledge as the expert-level agent. Instead, some portion of the novice's knowledge base has been purposely corrupted. Each expert/novice pair (E, N_i) will later be examined by the comparison methods to identify similarities and differences between the actors' behavior.

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 as well as 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 and mitigates some of the complications that arise when counting elements in the confusion matrix.

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 is representative of flaws that would occur naturally during the development process. If our comparison methods examined the novice-level agent's knowledge base directly, this would indeed be a serious concern. However, all of our comparison methods identify errors 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 8.2.1. Thus, we should have a high degree of confidence that the changes we introduced in the following experiments do represent many of the observable errors we would expect to see in an actual development environment.

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. Some of this information will require examinations of the behavior traces collected in the next step.

The fourth step is to acquire concrete examples of both the expert's and novice's behavior by gathering the behavior traces, BT_E and BT_{N_i}, that will be used to compare the agents' behavior. In most situations including those examined in this study, 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: 1) no two behavior traces are identical; and 2) all of the predicted errors actually occur in at least one of the novice's behavior trace.

While we are examining the novice's 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 manifests. In this way, we annotate all of the attributes of the error (e.g., whether it is primary or secondary, omission or commission). This includes details that may not have been clear during the initial assessment of how both actor's knowledge differed (step 3). 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.

Cataloging which errors occur in each behavior trace is an extremely tedious process representing the bulk of the experimental effort. As a result, we try to maximize our use of each behavior trace by constructing families of individual experiments to evaluate the impact of different sets of observational data without capturing and inspecting new behavior traces.

Instead of simply running one experiment for each (E, N_i) pair, we run multiple experiments using different subsets of our observational data. This process begins after the actor pair (E, N_i) has been selected and after the behavior traces, BT_{N_i} and BT_E, have been captured and inspected. At this point, we split the observations into a number of subsets: $n_{ij} \subset BT_{N_i}$ and $e_k \subset BT_E$ to form individual experiments. A single experiment consists of examining each comparison method's performance on a pair of these subsets (n_{ij} and e_k). A *family* of experiments contains the experiments that compare all n_{ij} to all e_k 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. By constructing experiment families in the way, we are able to examine the impact of different observational data without being overwhelmed by the manual inspection task.

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. Because our validation approach relies on testing, we cannot hope to identify errors that do not occur in the captured behavior traces.

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 traces 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 the 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. 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 additional investigation either by manually examining some examples of behavior or by examining the novice agent's knowledge base.

8.2.3 Counting Errors

Because the error forms identified in Section 8.2.1 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 has 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 we calculate what errors were and were not recognized depends both on what errors manifest in the behavior traces, and what errors are detected by the automated system.

Our approach to counting can be generalized by the following rules:

- If only simple errors (omission, commission, intrusion) 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 (duplication, misplacement) are detected correctly, 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 count it as a false positive.
- If a primary error (first error in a causal string) is detected correctly, count true positives for the primary error and all of the secondary errors (subsequent errors in a causal string) 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 number of additional errors required to cover all true errors.

One of the side 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 the metric *Report Density* which we will use to assess each comparison method's performance.

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

Because report density makes no reference to the number of errors that go unidentified by a particular behavior comparison metric, a complete assessment requires the use of a second metric. For our experiments, we use sensitivity which is calculated as follows:

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

Sensitivity measurements fall in the range $[0, 1]$. As sensitivity goes to one, all errors are identified by information in the summary. Conversely, as sensitivity goes to zero, no errors are identified by the data in the summary. Thus, we favor comparison methods which can obtain higher report density without sacrificing sensitivity. In the following two subsections we put the experimental framework and assessment metrics described thus far to use evaluating the performance of behavior bounding and the benchmark sequential methods in two distinct domains.

8.2.4 Object Retrieval Domain

Our first test environment is a simulated object retrieval domain in which an agent must navigate a grid-based world to find and collect a pre-specified object (initial results appear in Wallace & Laird, 2003). This environment is relatively simple because it is both discrete (no real valued sensors) and deterministic (no exogenous events). In addition, agents operating in this environment generate behavior sequences of relatively short length:

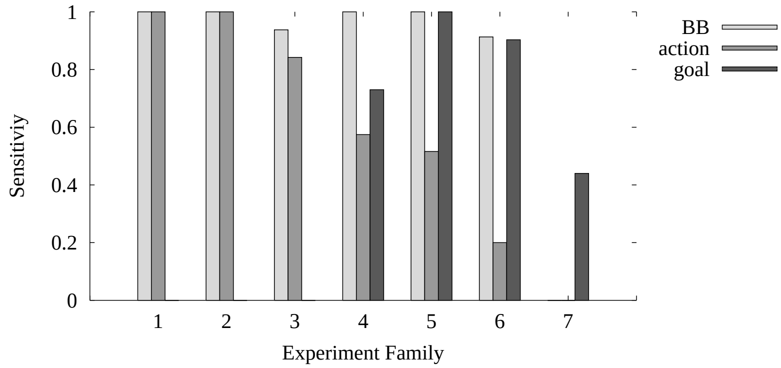

Figure 8: Sensitivity in the object retrieval domain

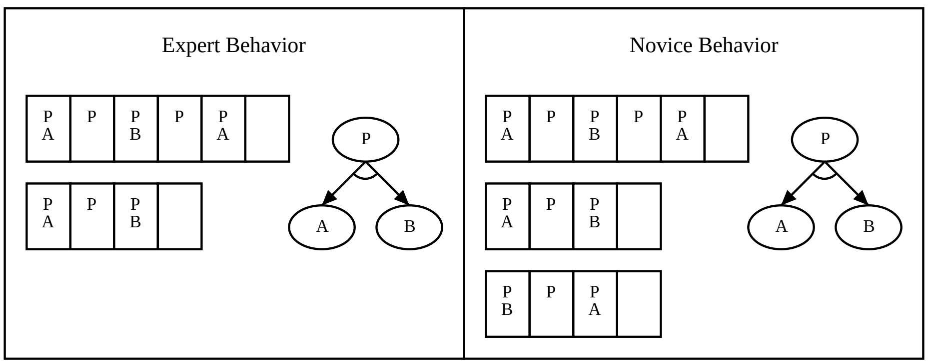

Figure 9: Limitations of behavior bounding's HBR in experiment family seven

approximately 20 to 30 goal or action elements are generated and the agent visits approximately 65 states. The agent's complete goal hierarchy has a maximum depth of 5 and contains 32 goal and action nodes together. Although this environment is simple in many ways, it does serve as a reasonable test for behavior bounding. Critically, correct behavior in the object retrieval domain requires reasoning (e.g., route planning) that relies on data structures that are not fully represented within the goal/sub-goal hierarchy.

Figure 8 illustrates the sensitivity across the seven experiment families in the object retrieval domain (ordering in the figure is arbitrary). 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 Section 3 on all but the final experiment family. The poor performance on this final experiment family is the second phenomena. This is due to limitations of the hierarchical representation itself which we discuss below.

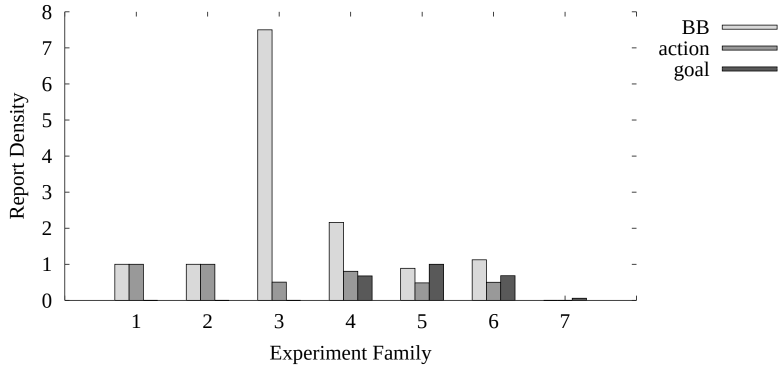

Figure 10: Report Density in the object retrieval domain

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, A and B, in the following sequence: A, B, A. The second decomposition is performed by pursuing 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 9. 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 9), it is of little surprise that the same HBR is produced and no differences are detected between the expert and the novice. In contrast, this error is readily identified by the goal-based benchmark approach (M_2). We could address this particular problem using a modified version of the HBR as we will describe in Section 11.2. However, even this approach requires some additional changes to the agent's internal representation for this particular behavior to be encoded correctly.

Behavior bounding's ability to detect errors while maintaining very concise reports is illustrated by its relatively high report density (see Figure 10). Recall that report density measures the amount of useful information in an error detection method's summary. Scores of one indicate that on average one error could be detected for each discrepancy indicated in the summary; scores less than one indicate the summary contains false positives. Report density scores higher than one are also possible but only when reports remain exceedingly concise by identifying 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 and goal-level

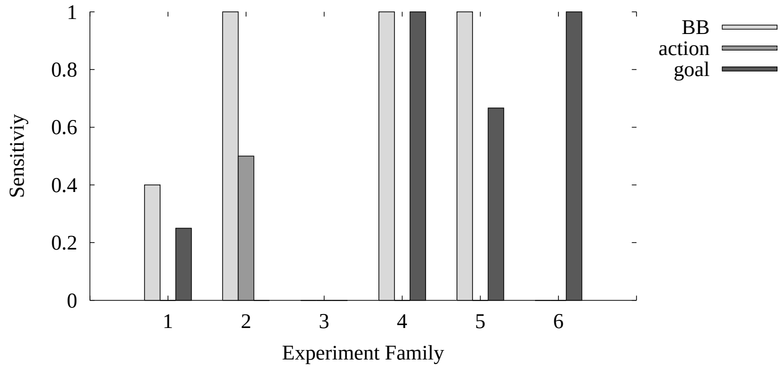

Figure 11: Sensitivity in the MOUT domain

primary 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 maintain a relatively low false positive count.

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

8.2.5 MOUT Domain

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. It is continuous, non-deterministic (exogenous events occur frequently) and has much longer sequence lengths than the object retrieval domain: between 30 and 200 goal/action elements are generated and the agent visits approximately 4000 distinct states per behavior trace (the state typically changes many times between the selection of a new goal or action). The goal hierarchy for the MOUT domain is larger than for the object retrieval domain containing 44 nodes and a maximum depth of 6. 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.

Figure 11 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. In addition, this figure points out the inherent scaling problems associated with the sequential method and illustrates their dramatic effects in more complicated environments. Experiment families three and six where behavior bounding's sensitivity drops to zero are worthy of note. Here, errors are again due to one aspect of the hierarchical behavior representation becoming over-generalized.

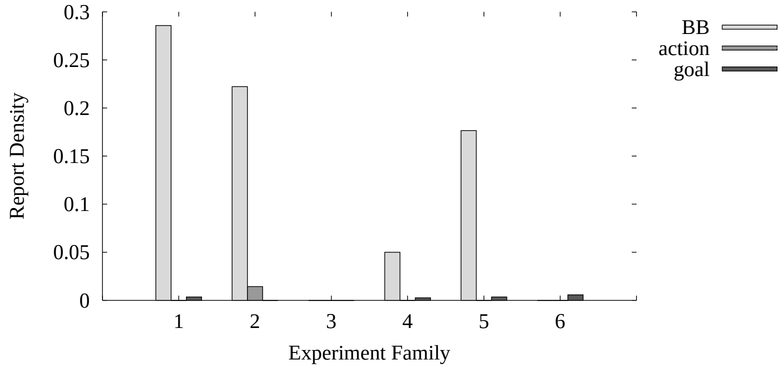

Figure 12: Report Density in the MOUT domain

Some of behavior bounding's strengths are better illustrated when we examine report density, as in Figure 12. 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 testing an agent's knowledge base. Equally worthy of note is the fact that even when the two benchmarks methods were more sensitive than behavior bounding, the usefulness of their error reports are questionable at best due to the exceedingly low report density.

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 at the domain itself and at 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 other goals. In other agent architectures, floating operators may be better described as concurrent top-level goals. Soar does not support concurrent goals, however, and floating operators are the prevailing method for this encoding this type of opportunistic behavior.

Because floating operators do not work in service of their parent goal, they effectively break the paradigm of the hierarchical behavior representation and 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. Second, 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 (i.e., the floating operator may be observed in different parts of the expert's and novice's hierarchy). This situation will

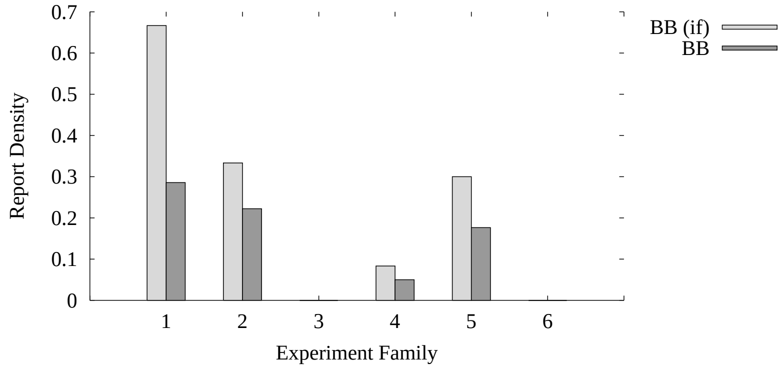

Figure 13: Report Density in the MOUT domain ignoring floating operators

result in a behavior representation that fails to satisfy the basic structure requirements of the lower boundary node.

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 is probably best avoided as a result. Another approach would be to tag floating operators so they could be treated differently by the Create-Hierarchy algorithm[4]. 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 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 13 illustrates the effect on report density when floating operators are ignored (note change of scale on y-axis). 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 more useful.

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

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

9. 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.

To substantiate this claim, we performed an experiment in which five human participants attempted to find and correct flaws in an agent's behavior both with and without information from behavior bounding's error report[5]. 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 with, and one without the aid of behavior bounding's error report. In the unaided situation, participants relied on standard debugging tools and techniques that they were already in the practice of using. Once the flaw was identified, the participants corrected the agents' knowledge using VisualSoar, the standard Soar development environment. In the aided situation, participants were given behavior bounding's error report to help make sense of the agent's behavior. Thus, in the experiments presented below, there are two conditions: aided, and unaided. Condition is a within-subject variable, which is to say that each participant experiences both.

Our test-bed agent was taken from the object retrieval domain discussed in Section 8.2.4. The initial setup followed similar lines as our earlier experiments. We began by constructing an expert-level agent that exhibited "correct" behavior. This agent could perform its task in four distinct but similar ways and required 78 Soar rules to encode. Note that in normal use, observations of correct behavior are likely to come from human experts. However, by creating a correct agent first, it is possible to describe precisely how flawed agents differ from the ideal (both in behavior and in their implementation). This property is critical for the experiment.

After creating the expert-level agent, we constructed two novice-level agents (Novice-A and Novice-B). The participants' task was to identify and correct any behavioral differences between the novice agents and the expert-level agent. Because each participant would validate both novice agents (using a different method for each one), one of our primary

5. Initial results reported by Wallace (2007).

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

It is worth noting that the flaws introduced into these agents are minor by most standards. In this experiment, flawed behavior does not result in deadlocks or infinite loops. Indeed, when viewed in the classical sense, these agents are not necessarily "flawed". They are successful in achieving the desired final state (finding the lost object). However, the agents do not pursue the same trajectories through state/action/goal space, and the participants' task is to determine how these trajectories differ and then find and correct the fault that causes the difference.

Because none of the participants had used, or even seen, the graphical behavior comparisons generated by behavior bounding, they were given a short, 15 minute, tutorial to become familiar with the graphical behavior summary provided by our interface. In addition, participants were asked to read a short summary that provided a description of the debugging task, a summary of the agent's behavior, and a plain English description of some salient goals and actions that would be pursued during task performance. 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.

At this point, participants were randomly assigned an agent to validate. We attempted to mitigate bias by varying the order in which the aided and unaided tests were presented as well as the pairing between the agent and the validation method. 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 how the novice agent's behavior differed from the standard set by the correct expert-level agent. In the unaided situation, no specific instructions were given on how to identify errors. Participants were free to look for errors using whatever debugging techniques they had developed in the course of working with Soar. Similarly, in the aided situation no specific instructions on

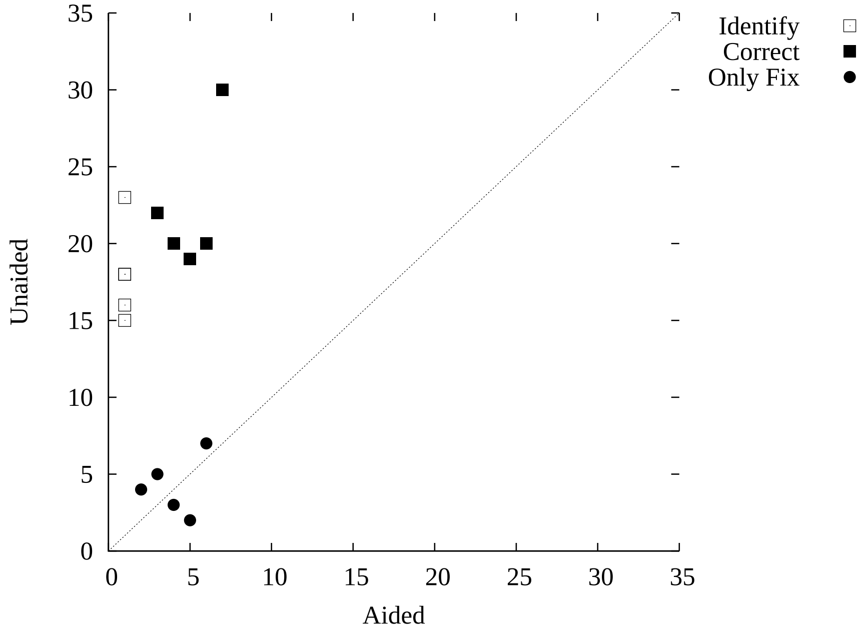

Figure 14: Time required to identify and correct errors using two techniques

how to identify errors were given. Participants generalized their tutorial experience to interpret the information in behavior bounding's error report and to identify what changes would be required to make the flawed agents behave correctly. In both situations, when the participant correctly identified the error in the flawed agent's behavior (e.g., by saying "The novice does not always perform action `X` before action `Y`"), the elapsed time was recorded. We call this the time required to *identify* the error.

The second phase of the debugging session began once the participant determined that they were ready to try modifying the flawed agent's knowledge in order to correct the observed 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) for this portion of the task. Once the participant had made changes, they re-examined 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 the time spent from the beginning of phase one until the end of phase two was recorded as the time needed to *correct* the agent's behavior[6].

Figure 14 shows the time spent by each participant on both the aided and unaided tasks and highlights the benefits of behavior bounding. The x-coordinate indicates time spent debugging in the aided situation when information from behavior bounding's error summary was used while the y-coordinate indicates time spent in the unaided situation when only

6. There were no cases in which the participant believed the agent's behavior had been corrected when in fact errors remained.

the participant's normal debugging techniques were used. Three sets of points are plotted: time to identify the error; time to correct the error; and time required only for the fix (i.e., the difference between time to correct and time to identify). The line $y = x$ is also plotted for reference; points that lie to the left of this line indicate that the participant performed better (i.e., faster) in the aided situation.

The cluster of points nearest to the origin (labeled "only fix" in the legend) indicate that behavior bounding had little if any effect on the time required to fix the agent's knowledge error once it was identified. Instead, behavior bounding's impact, as expected, comes from the reduction in time required to identify the error. This leads to a reduction in the overall time required for the validation task. A paired t-test was used to determine statistical significance of each of the three timed operations illustrated in the figure. Not surprisingly, the test confirms a statistically significant performance advantage is gained by using information from behavior bounding on both the *time to identify* and *time to correct* the error ($p = .0006$, $p = .0002$ respectively). The paired t-test does not indicate a statistically significant difference in the times required simply to fix the error for the aided and unaided situations ($p = .85$), again matching expectations.

From this data, 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 a number of reasons why these results may be on the conservative side of optimistic.

First, we would expect the HBR to be more useful as the complexity of the domain and of the agent's behavior increases—developers wishing to examine raw behavior traces will need to look at longer traces and more traces for complex environments whereas with the HBR, they only need to view one data structure. Second, the test conducted above is clearly influenced by the design of behavior bounding's user interface. We conducted no formal experiments to increase the quality of the interface, so it is quite possible that future implementations would be capable of delivering information more effectively to the user, thus producing an increase in efficiency.

10. Related Work

As noted previously in Section 4, a number of other areas of artificial intelligence, particularly machine learning have addressed problems closely related to those we examined here. In the following subsections, we briefly comment on some of the most salient areas.

10.1 Plan Recognition

The behavior comparison we have described is related to keyhole plan recognition (Albrecht, Zukerman, & Nicholson, 1998), or more closely, to the team monitoring by overhearing work of Kaminka, Pynadath, and Tambe (2002). In team monitoring, the objective is to determine what task an agent or set of agents is performing given limited observations of their actions and the communications that pass between them. Plan recognition is possible, in part, because a complete team-level plan allows the monitoring system to identify the agent's goals as observational information is acquired. When enough information is obtained, a single plan can be identified and ascribed to the agent(s). In behavior comparison, the objective is similar. The salient difference between our work and plan recognition is that we are not given the plan library; instead we are attempting to recreate a model of its execution through a series of observations in order to determine whether both actors will pursue their goals in the same manner (i.e., have the same plan library).

10.2 Learning By Observation

A number of systems (e.g, van Lent & Laird, 1999; Wang, 1995; Konik & Laird, 2006) have also been developed to learn procedural rules or plan operators from observations of expert behavior. Wang's OBSERVER (Wang, 1995) learns STRIPS style operators; van Lent's KnoMic (van Lent & Laird, 1999) learns production rules for the Soar agent architecture and Konik's system (Konik & Laird, 2006) creates first order logic rules that are later converted into Soar productions. All three systems use similar behavior traces as our approach, although Wang's OBSERVER works only with primitive actions so there is no notion of non-atomic goals and thus no need to annotate them in the behavior traces. Of these systems, Konik's has been demonstrated within the most complex domain (a 3-D virtual environment in which an agent must learn to successfully navigate a series of rooms).

The key difference between our approach and theirs lies in the fundamental premise. While we are interested in learning a simple and concise model of behavior that an outside third-party can use to validate an existing (but untrusted) agent, these systems aim to learn the agent's knowledge altogether. While learning complete task knowledge is clearly an important goal for the community, there remain a set of important task domains (e.g., military and mission critical applications) where learned systems are often treated with skepticism and human coded systems are still preferred. The approach we have described, however, could be useful to help bridge this gap by allowing skeptical parties to validate the behavior of learned systems. Thus, while it may seem on the surface that by solving the "learning executable task knowledge" problem one also solves the behavior comparison problem we have outlined, that is not the case—in mission critical applications, the agent's behavior still requires validation and a human in the loop to "sign off" on its correctness. Moreover, when knowledge is learned instead of engineered, the validation task is likely to become much more difficult as there is no one to document the system or to field questions about the function of any particular component.

10.3 Hierarchical Reinforcement Learning

Reinforcement Learning seeks to provide methods by which an agent can learn to approximate an optimal behavioral strategy while interacting with its environment. In reinforcement learning, optimality is defined by a reward function that is outside of the agent's control (it is part of the environment) and the agent learns through interaction with the environment how to maximize this function. Traditional (flat) approaches to reinforcement learning such as Q-Learning (Watkins & Dayan, 1992) may require a long training time to converge on an optimal policy. Price and Boutilier (2003) show how reinforcement learning can be facilitated by observing a mentor perform a task while Hierarchical Reinforcement Learning (Dietterich, 2000; Andre & Russell, 2002; Marthi, Russell, Latham, & Guestrin,

2005) seeks, in part, to reduce the complexity of the learning problem with the use of external domain knowledge in the form of a programmer-defined action hierarchy.

Both traditional Reinforcement Learning (RL) and Hierarchical Reinforcement Learning (HRL) differ significantly from our approach in three fundamental ways. First, as with the method described in the previous subsection, the goal in (H)RL is to learn an executable model for behavior, not a model that can be used to help validate a system. Second, in (H)RL, models are learned via interaction with the environment and with an environmentally defined reward function. Instead, we are interested in learning directly from observation of expert behavior without experimental interaction in the environment. Finally, unlike both RL and HRL, we do not assume the existence of a reward function and moreover we are not interested in optimal behavior in any sense other than close approximation to human behavior.

Aside from these important differences, there is a commonality between Hierarchal Reinforcement Learning and our approach that stems from the behavior model. An open issue in Dietterich's presentation of MAXQ (Dietterich, 2000) and restated by Barto and Mahadevan (2003) is whether the programmer-supplied information (the MAXQ task-graph) in Hierarchical Reinforcement Learning could be acquired automatically. Each subtask M_i in a MAXQ task-graph is a three tuple $\langle T_i, A_i, \tilde{R}_i \rangle$. $T_i(s_i)$ partitions the state space into active states S_i and terminal states T_i (a subtask can only be executed if the current state is in S_i). A_i is a set of actions that can be performed to achieve the subtask and $\tilde{R}_i(s'|s,a)$ is a pseudo reward function indicating how desirable each terminal state is for this subtask.

Our approach could be used to help construct part of the MAXQ task graph directly from observations. First, the goal/subgoal hierarchy we build can be used directly to identify A_i, the set of actions that can be performed in each subtask. Second, some task parameters that we learn are tied to information in the state (this relation can be observed directly in the behavior trace). This information combined with the temporal constraints we learn for all goal/action nodes could be used to identify some of the conditions when a task could be entered (some properties of the active states identified by the predicate T_i). Together this could help construct the MAX-Q task graph based on observations of an expert's performance.

10.4 Inverse Reinforcement Learning

Inverse Reinforcement Learning (IRL) (e.g., Abbeel & Ng, 2004; Ramachandran & Amir, 2007) attempts to reconstruction an implicit reward function given a set of example behaviors. IRL in combination with RL has been used in simple domains to reproduce behavior for which there is no explicit reward function. This would permit a system to, for example, learn to model a human expert's behavior by 1) reconstructing the expert's implicit reward function by observing example behaviors and then 2) interacting with the environment to generate a policy that maximizes this implicit reward. Together, these technologies provide a potentially powerful alternative to the learning by observation methods described previously. However, to the best of our knowledge IRL has not yet been demonstrated within a hierarchical setting, and so the learning by observation methods still present the current state of the art for learning hierarchical task knowledge.

11. Extensions to Behavior Bounding and Future Directions

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 5, we noted some of the representational limitations of behavior bounding's HBR. Here, we examine extensions to behavior bounding that could positively affect its performance and briefly describe a promising direction for future work. We leave the implementation of these extensions and detailed discussion as future work.

11.1 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.

The hierarchy represented by the lower boundary node simply identifies the space of potentially acceptable goal decompositions. As a result, it would be logical to create this structure early in the design phase as expert knowledge is being acquired for the agent. Lee and O'Keefe (1994) as well as Yen and Lee (1993) 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 knowledge acquisition. Moreover, they argue that identifying the relationship between goals, sub-goals and primitive actions helps to organize the agent's knowledge 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 making the process more structured and directed.

If constructing 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. One such use of the manually constructed HBR is to help validate the agent's design early during the implementation process. It is generally believed 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.

11.2 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 modestly more representational power. The semantics of And and Or nodes are easily covered: an And node is simply one in which all children are Always while an Or node is one in which all children are Sometimes. The semantics of Sometimes and

Always also make it possible to encapsulate new decomposition relations that do not occur with the And/Or relation.

Recall the problematic behavior in Section 8.2.4 where the HBR fails to correctly encode the proper decomposition relations (specifically that goal P can decompose into subgoals A, B, A or into subgoals A, B but not into B, A). Sometimes/Always constraints can encode this decomposition, albeit only if an additional layer of subgoal is added to the task specification. By introducing two new subgoals so that P decomposes into C^*, D[7] and C decomposes into A^*, B^* while D decomposes into A^*, we would be able to encode the correct behavioral patterns with respect to P, A, and B with the only caveat of having to interject two new goals C and D. Of course, the point of this discussion is not to justify such ad-hoc modifications to the task structure, but rather to show a concrete instance where Sometimes/Always constraints may add beneficial representational power.

Sometimes/Always constraints have no effect on the learnability or construction cost of the HBR. And while we have not tested this modification in detail, preliminary results in the MOUT data sets do indicate a minor improvement in performance for this domain.

11.3 Additional Enhancements

Two additional enhancements to the HBR are also left as future work. The first is the ability to deal with concurrent goals or actions. As Soar does not support concurrent operators, this cannot be tested within our exiting system. However, if such support were added to the HBR, it may be possible to avoid some of the issues associated with floating events encountered in the MOUT domain. The second enhancement would be to allow more than one node to be constructed to represent a given action/goal within a particular context. In the current representation, there are no two sibling nodes with the same name (there is exactly one node to represent all identically named goal/actions within any context). While this keeps the representation simple, it also can be held responsible for some representational problems like the one discussed in Section 8.2.4. The disadvantage of this approach is that it is unclear when new nodes should be added to the hierarchy. If a new node is added each time a goal/action is pursued, then the hierarchy grows much more rapidly (directly as a function of the length of the behavior tracing) increasing the computational complexity and decreasing the rate of generalization.

11.4 Behavior Bounding in the Runtime Environment

A promising direction for additional future work is to use the ideas presented in this paper, specifically the constraints contained in the upper-boundary node's behavior representation, to monitor an agent's behavior at runtime. This approach, which we have recently begun to explore, provides a mechanism for determining when an agent may be making inappropriate decisions (Wallace, 2005b, 2005a). Inconsistencies between an agent's desired course of action and the constraints specified by the upper boundary node could be used to enforce social policies such as interaction protocols between groups of agents or to dynamically adjust an agent's degree of autonomy if it begins to make questionably choices. Moreover, the high-level constraints specified by the hierarchical behavior model require

7. * indicates an ALWAYS node

no direct knowledge of the agent's underlying implementation language (only of its goal decomposition). This means that our approach could also be used as a safeguard against implementation errors in agents built by third parties that may not have been adequately validated.

12. Contributions

We have introduced behavior bounding, a model-based approach for comparing two actors' behavior. This novel approach uses a hierarchical behavior representation motivated by the desire to build a high-level model of behavior from observations of either human or computer agent performance that is efficient to create and maintain and effective in use. We have demonstrated how behavior bounding meets these requirements by providing both theoretical and empirical support for these claims. Finally, we have shown that information from behavior bounding's comparison can significantly aid the process of identifying problems in an agent's behavior, thus speeding knowledge-base validation by a significant factor.

Acknowledgments

I would like to thank John Laird for his help in reviewing early versions of this paper, along with members of the UM Soar research group who participated in the user study. Portions of this work were supported by the Office of Naval Research under contract N61339-99-C-0104.

References

Abbeel, P., & Ng, A. Y. (2004). Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the Twenty First International Conference on Machine Learning*, pp. 1–8.

Albrecht, D. W., Zukerman, I., & Nicholson, A. E. (1998). Bayesian models for keyhole plan recognition in an adventure game. *User Modeling and User-Adapted Interaction*, *8*(1-2), 5–47.

Andre, D., & Russell, S. J. (2002). State abstraction for programmable reinforcement learning agents. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pp. 119–125.

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

Barto, A. G., & Mahadevan, S. (2003). Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems: Theory and Applications*, *13*, 343–379.

Bordini, R. H., Fisher, M., Visser, W., & Wooldridge, M. (2004). State-space reduction techniques in agent verification. In *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pp. 896–903.

Bordini, R. H., Fisher, M., Visser, W., & Wooldridge, M. (2006). Verifying multi-agent programs by model checking. *Autonomous Agents and Multi-Agent Systems*, *12*, 239–256.

Dietterich, T. G. (2000). Hierarchical reinforcement learning with the MAXQ function decomposition. *Journal of Artificial Intelligence Research*, *13*, 227–303.

Erol, K., Hendler, J., & Nau, D. S. (1994). HTN planning: Complexity and expressivity. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pp. 1123–1128. AAAI Press/MIT Press.

Fisher, M. (2005). Temporal development methods for agent-based systems. *Autonomous Agents and Multi-Agent Systems*, *10*, 41–66.

Giarratano, J., & Riley, G. (1998). *Expert Systems: Principles and Programming*. PWS Publishing Co., Boston, MA.

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

Ingrand, F. F., Georgeff, M. P., & Rao, A. S. (1992). An architecture for real-time reasoning and system control. *IEEE Expert*, *7*(6), 33–44.

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

Jones, R. M., Laird, J. E., Nielsen, P. E., Coulter, K. J., Kenny, P., & Koss, F. V. (1999). Automated intelligent pilots for combat flight simulation. *AI Magazine*, *20*(1), 27–42.

Kaminka, G. A., Pynadath, D. V., & Tambe, M. (2002). Monitoring teams by overhearing: A multi-agent plan-recognition approach. *Journal of Artificial Intelligence Research*, *17*, 83–135.

Kirani, S. H., Zualkernan, I. A., & Tsai, W.-T. (1994). Evaluation of expert system testing methods. *Communications of the ACM*, *37*(11), 71–81.

Konik, T., & Laird, J. E. (2006). Learning goal hierarchies from structured observations and expert annotations. *Machine Learning*, *64*(1–3), 263–287.

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

Lee, S., & O'Keefe, R. M. (1994). Developing a strategy for expert system verification and validation. *IEEE Transactions on Systems, Man and Cybernetics*, *24*(4), 643–655.

Lucas, P. (1998). Analysis of notions of diagnosis. *Artificial Intelligence*, *105*, 295–343.

Marthi, B., Russell, S., Latham, D., & Guestrin, C. (2005). Concurrent hierarchical reinforcement learning. In *Proceedings of the International Joint Conference on Artificial Intelligence 2005*, pp. 779–785.

Menzies, T. (1999). Knowledge maintenance: the state of the art. *The Knowledge Engineering Review*, *14*(1), 1–46.

Mitchell, T. M. (1982). Generalization as search. *Artificial Intelligence*, *18*(2), 203–226.

Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill.

Murphy, P. M., & Pazzani, M. J. (1994). Revision of production system rule-bases. In *Proceedings of the Eleventh International Conference on Machine Learning*, pp. 199–207. Morgan Kaufmann.

Price, B., & Boutilier, C. (2003). Accelerating reinforcement learning through implicit imitation. *Journal of Artificial Intelligence Research*, *19*, 569–629.

Ramachandran, D., & Amir, E. (2007). Bayesian inverse reinforcement learning. In *Proceedings of the International Joint Conference on Artificial Intelligence 2007*, pp. 2586–2591.

Rickel, J., Marcella, S., Gratch, J., Hill, R., Traum, D., & Swartout, W. (2002). Toward a new generation of virtual humans for interactive experiences. *IEEE Intelligent Systems*, *17*(4), 32–38.

Shortliffe, E. H. (1987). Computer programs to support clinical decision making. *Journal of the American Medical Association*, *258*(1), 61–66.

Swartout, W., Hill, R., Gratch, J., Johnson, W. L., Kyriakakis, C., LaBore, C., Lindheim, R., Marsella, S., Miraglia, D., Moore, B., Morie, J., Rickel, J., Thiebaux, M., Tuh, L., Whitney, R., & Douglas, J. (2001). Toward the holodeck: Integrating graphics, sound, character and story. In *Proceedings of the Fifth International Conference on Autonomous Agents*, pp. 409–416.

Traum, D., Rickel, J., Gratch, J., & Marsella, S. (2003). Negotiation over tasks in hybrid human-agent teams for simulation-based training. In *AAMAS '03: Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems*, pp. 441–448.

Tsai, W.-T., Vishnuvajjala, R., & Zhang, D. (1999). Verification and validation of knowledge-based systems. *IEEE Transactions on Knowledge and Data Engineering*, *11*(1), 202–212.

van Lent, M. C., & Laird, J. E. (1999). Learning hierarchical performance knowledge by observation. In *Proceedings of the 1999 International Conference on Machine Learning*, pp. 229–238.

Veloso, M., Carbonell, J., Perez, A., Borrajo, D., Fink, E., & Blythe, J. (1995). Integrating planning and learning: The PRODIGY architecture. *Journal of Theoretical and Experimental Artificial Intelligence*, *7*(1), 81–120.

Wallace, S. A. (2005a). Abstract behavior representations for self-assessment. In *AAAI Spring Symposium on Meta-Cognition in Computation (ASSMC 2005). AAAI Technical Report SS-05-04.*, pp. 120–125.

Wallace, S. A. (2005b). S-Assess: A library for self-assessment. In *Proceedings of the Fourth International Conference on Autonomous Agents and Multiagent Systems (AAMAS-05)*, pp. 256–263.

Wallace, S. A. (2007). Enabling trust with behavior metamodels. In *AAAI Spring Symposium on Interaction Challenges for Intelligent Agents (ASSICIA 2007). AAAI Technical Report SS-07-04.*, pp. 124–131.

Wallace, S. A., & Laird, J. E. (2000). Toward a methodology for AI architecture evaluation: Comparing Soar and CLIPS. In Jennings, N., & Lespérance, Y. (Eds.), *Intelligent Agents VI — Proceedings of the Sixth International Workshop on Agent Theories, Architectures, and Languages (ATAL-99)*, Lecture Notes in Artificial Intelligence, pp. 117–131. Springer-Verlag, Berlin.

Wallace, S. A., & Laird, J. E. (2003). Behavior Bounding: Toward effective comparisons of agents & humans. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pp. 727–732.

Wang, X. (1995). Learning by observation and practice: An incremental approach for planning operator acquisition. In *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 549–557.

Watkins, C. J. C. H., & Dayan, P. (1992). Q-learning. *Machine Learning*, *8*, 279–292.

Weitzel, J. R., & Kerschberg, L. (1989). Developing knowledge-based systems: Reorganizing the system development life cycle. *Communications of the ACM*, *32*(4), 482–488.

Yen, J., & Lee, J. (1993). A task-based methodology for specifying expert systems. *IEEE Expert*, *8*(1), 8–15.

Yost, G. R. (1996). Implementing the Sisyphus-93 task using Soar/TAQL. *International Journal of Human-Computer Studies*, *44*, 281–301.