S-Assess: A Library for Behavioral Self-Assessment

Scott A. Wallace
Washington State University Vancouver
14204 NE Salmon Creek Ave.
Vancouver, WA 98686
swallace@vancouver.wsu.edu

ABSTRACT

Developing and testing intelligent agents is a complex task that is both time-consuming and costly. This creates the potential that problems in the agent's behavior will be realized only after the agent has been put to use. As a result, society is left with a vexing problem: although we can create agents that seem capable of performing useful tasks autonomously, we are simultaneously unwilling to trust these agents because of the inherent incompleteness of testing. In this paper we present a framework that brings validation techniques out of the laboratory and uses them to monitor and constrain an agent's behavior concurrent with task execution. Applications of this framework extend well beyond helping to ensure safe agent behavior through run-time validation. They also include the ability to enforce social or environmental policies or to regulate the agent's autonomy.

Categories and Subject Descriptors

I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search; C.4 [**Performance of Systems**]: Reliability, availability, and serviceability

General Terms

Reliability

Keywords

Agents, self-assessment, safe behavior

1. INTRODUCTION

As our theoretical knowledge and access to computational power increases, it is likely that there will be an increased demand for, and an increased ability to produce autonomous agents. However, as the task of designing agents increases in complexity, so will the difficulty of debugging these systems and validating their behavior. Because current validation techniques rely on examining the agent's behavior in a number of test scenarios, they are incomplete by their very nature. As a result, it is difficult, if not impossible, to obtain complete confidence in the quality of the agent's performance when the domain is of any real complexity. The gap between an agent's promised ability to act autonomously on our behalf and our ability to validate its correctness begs us to ask: how much trust should be granted to unvalidated agent technologies, and how should new technologies be introduced into their operating environments without subjecting the environment and the agent to unnecessary risk?

To reduce the risks associated with introducing new, unproven, agents into their domain, we believe that it will be critical to extend validation methods beyond the laboratory and into the run-time environment. At another level, this problem of run-time validation can be viewed as the more general problem of ensuring the agent upholds a set of operating constraints appropriate for its domain. Note that by definition, this problem is trivially solved for a correctly engineered agent. That is, a correct agent will use its own internal preference scheme (such as would be defined by a utility function) to determine the most appropriate behavior given its current goals and operating environment. It is only in situations where the agent's behavior is incorrectly or incompletely specified that the agent's own policy will fail to meet its operating constraints.

The research presented here is fundamentally different from much of the work on agent systems because our default assumption is that the agent's knowledge is inadequate for correctly accomplishing its task. Our goal is to establish how to ensure correct behavior by identifying and possibly preventing errors at run-time using high-level descriptions of operating constraints which are external to, and possibly in conflict with, preferences described in the agent's knowledge base.

The remainder of this paper begins by examining how errors are introduced into an agent's knowledge base, and how previous work has attempted to identify and correct such errors. Our analysis provides a taxonomy of approaches for addressing behavior correctness that we use to classify related work. We develop the notion of behavioral self-assessment in which the agent or agent architecture evaluates potential goals and actions with respect to externally defined constraints to help ensure its own correctness. We then describe a framework for behavioral self-assessment and present the S-Assess library, an initial implementation of this framework. We close the paper by describing potential uses for our library outside of validation and potential avenues for future work.

AAMAS'05, July 25-29, 2005, Utrecht, Netherlands.
Copyright 2005 ACM 1-59593-094-9/05/0007 ...$5.00.

2. ENSURING CORRECT BEHAVIOR

In this section, we will consider how errors may be introduced into agent systems and the standard approach for detecting and correcting these errors before the agent leaves the laboratory. We then examine potential approaches for ensuring correct behavior once the agent has entered its operating environment.

2.1 Sources of Errors

The agent design process consists of four high-level steps. First, specifications are created for what tasks the agent should perform and how these tasks should be performed. Second, the specification is implemented as an agent program. Third, the implementation undergoes an iterative process of testing and refinement. Finally, in the fourth stage, the agent leaves the laboratory and is put to use in its actual operating environment. Errors are most likely to be introduced at two of these design phases: during specification (phase one), and during implementation (phase two).

Errors may be introduced during the specification phase for a number of reasons. Superficially, this may be because the specification was created hastily without sufficient detail or refinement. However, as we seek to design agents that perform increasingly complicated tasks, there is another, more significant reason that errors may be introduced at the specification phase. This is due to the fact that it can be exceedingly difficult to precisely specify correct operating parameters.

Take, for example, the TacAir-Soar project [7] in which the design goal is to build agents to fly tactical military aircraft as part of a simulated training exercise. The ultimate purpose of the simulation is to train human pilots for situations that may occur in the real world (where teams will consist entirely of humans). As a result, it is critical that agents used in the training simulation produce behavior consistent with that of human experts. The problem, however, is that the best specification for expert pilot behavior exists in the minds of expert pilots. It is not codified in any complete manner, and because the domain experts are not necessarily skilled at describing the reasoning behind their actions, it can be extremely difficult for them to externalize a correct specification.

Clearly, without a correct and complete design specification, it is unlikely that the agent's behavior will be correct and complete. Note however that for all tasks we are likely to want to perform, there is a specification of correctness, even if it exists solely in the designer's mind.

In addition to the specification phase, errors are also likely to be introduced during the implementation phase because the task of creating an executable system from a design specification is non-trivial. Large knowledge based agents may take several man years to develop and represent many thousands of lines of executable code. As the complexity of these implementations increases, the probability that errors will be introduced via simple programming mistakes rises quickly. Regardless of the quality of the original specification, it is exceedingly likely that all but the most trivial agents will have some programming errors in the early phases of testing and refinement.

2.2 Detecting Errors in the Laboratory

Currently, most agent development projects operate under the implicit assumption that critical errors can be removed before the agent leaves the laboratory. Most agent systems are designed around the iterative processes of testing and refinement. As the agent system nears completion, testing becomes more sophisticated: the focus typically turns to case-based validation in which the agent's behavior is evaluated in a number of test scenarios.

During validation, domain experts and knowledge engineers examine the agent's behavior in each test scenario looking for errors and correcting the agent's knowledge as required. Two critical properties of this validation process deserve to be explicitly highlighted.

The first critical property in this approach is the assumption that the humans supervising testing will be able to identify errors when they occur. As a result, this means that some specification for correct agent behavior must exist, even if only within the designer's mind. This reinforces our earlier assertion to the same effect and paves the way towards a method for detecting errors automatically.

The second critical property of the validation process is that case based testing is incomplete by its very nature. In order to ensure that the agent's behavior is free of errors, testing would need to examine all possible situations. Clearly, this is infeasible. In most non-trivial task domains the number of environment states is extremely large if not infinite. Even with a careful examination of equivalence relations among states, time and cost restrictions often prevent developers from dedicating adequate resources to testing. The net result of this second property is that although standard in the community, case based testing is insufficient to guarantee agents will behave correctly once they leave the laboratory.

3. MONITORING RUN-TIME BEHAVIOR

The discussion above highlights the inadequacies of current testing approaches and the need to extend validation beyond the laboratory. The main challenge in such an effort will be creating a framework that can adequately identify errors without direct human supervision. However, our reinforced belief that specifications for correct agent behavior exist, even if they have not been properly codified or implemented, suggests that this approach is possible.

Working on this belief, one can identify a simple method for extending validation to the agent's operating environment. We simply need to represent the operating constraints within a high-level behavior model and compare the agent's behavior to this high-level model as the task is being performed. In the following subsections, we categorize potential approaches to performing this simple run-time monitoring. We examine the relationship of previous work to this taxonomy of approaches, and motivate the design decisions that lie beneath our behavior assessment framework. Later, in Section 5.1 as we describe our implementation, we examine appropriate choices for a high-level behavioral model paying particular attention to specifications that may be difficult for domain experts to articulate.

3.1 Potential Approaches

We distinguish potential approaches to run-time behavior monitoring along two dimensions. The first dimension is the time at which monitoring occurs. The second dimension is the relationship of the monitoring system (or subsystem) to the agent. Each approach represents a point in this two dimensional space, with distinct assets and liabilities. We begin with an overview of the taxonomy and each approach it covers. As we develop the categories we briefly present the assets and liabilities of each approach from a theoretical perspective, and from the perspective of previous work.

Along the time dimension, we distinguish between monitoring that occurs at plan-time and monitoring that occurs at execution-time (either just before or just after the agent performs its action or selects a new goal). Along the second dimension, we distinguish two relationships between the monitoring system and the agent which we describe below.

3.1.1 External Monitoring

The first relationship occurs when we consider a monitoring system that is external to the agent. Here, the monitor would be its own process and would interact with the agent only through communication channels provided by the environment. Such a system could be designed to monitor the agent's behavior at plan-time. The system could, for example, accept, reject, or suggest modifications to the agent's proposed plan in order to ensure consistency with the domain's operating constraints. Critically, however, this approach relies on the agent to communicate its intended plan before beginning execution; if the agent cannot be trusted to cooperate in this manner, the approach is useless.

In contrast, an external system that monitored agent behavior at execution-time would not necessarily need direct cooperation from the agent. Monitoring could be done simply by watching the agent's sequence of externally visible actions (keyhole observation). The advantage of this approach is that the system could easily analyze observations about a number of agents' behaviors simultaneously, thus enforcing a group policy as opposed an individual agent policy. The main disadvantage stems from the fact that keyhole observation is useful only for detecting errors that can be observed from the agents' external actions. This has three implications. First, errors that are isolated to the internal reasoning and goal selection process may not be uncovered. Second, the agent may be able to perform a "sleight of hand" whereby an inappropriate action goes unobserved by the monitoring system. Finally, and perhaps most importantly, because this approach relies on observing actions, it is reactive—limited to detecting errors once the behavior has been performed, as opposed to proactively preventing the errors from occurring.

External monitoring has been the focus of much attention in the security community (e.g., [5,9]), and has been introduced as an important component of self-healing systems in IBM's autonomic computing initiative [8]. In the security community, external monitoring is a natural method to ensuring correct run-time behavior. The typical assumptions here are that a group of processes works correctly unless they are subverted by a malevolent outsider. When subverted, however, the process group may behave inconsistently with its operating constraints: changing or deleting files or allocating resources to inappropriate tasks. In this setting, it makes sense to monitor the system's activity through a common interface, such as log files. Thus, the interactions of multiple components can be validated and the monitoring system ensures compatibility with existing technologies.

3.1.2 Self-Assessment

A second relationship occurs when we consider a monitor that is an internal component of the agent itself. Such a monitor could be an extension to the agent architecture, or an additional knowledge-base used by the architecture. Both cases are instances of self-assessment, but each has moderately different benefits. In particular, if assessment is performed by the architecture, it is likely to be transparent to the agent proper, and thus to the agent's designers. In contrast, assessment performed by an additional knowledge-base is likely to allow a tighter coupling to the agent's domain knowledge, since both will be implemented using the same underlying representation.

The benefits of self-assessment as compared to external monitoring are multi-fold. Because the assessment subsystem is encapsulated by the agent, the agent cannot withhold its intentions from the monitor. Moreover, because the assessment subsystem will be able to examine more than externally visible aspects of the agent's behavior, this approach will likely be able to detect a larger set of errors than an external approach. Finally, because assessment could be performed before the agent commits to pursing a particular action or goal, self-assessment may be able to prevent errors as opposed to simply identify them. Each of these three properties make it increasingly likely that this approach will be advantageous for ensuring correct agent behavior.

Within the AI community, self-assessment has had limited attention. Typically, the self-assessment subsystem has been built into more traditional planning architectures. This approach is used to ensure plans are built according to both the agent's priority scheme and a set of external constraints. Weld and Etzioni describe such a system that is capable of enforcing high-level behavior constraints at plan-time [13]. Work on the CIRCA real-time architecture [11] by Atkins and others (e.g., [1,2]) has focused on ensuring agents avoid failure states when engaged in complex tasks with real-time constraints such as piloting aircraft. Perhaps not surprisingly, both systems mentioned here focus on monitoring behavior at plan-time. Weld and Etzioni's system has no method to ensure consistency at execution-time. CIRCA, however, does perform limited run-time monitoring, but this monitoring only allows the system to initiate replanning if unanticipated states arise.

3.2 Execution-Time Self-Assessment

Interestingly, relatively little research has focused directly on the issue of *execution-time* self-assessment. We believe execution-time self-assessment is critical for two reasons. First, in complex, dynamic environments, agents cannot assume that plan-time enforcement of domain constraints will be sufficient to prevent execution-time errors. Second, many agent architectures do not enforce strict plan then act cycles and so cannot directly benefit from research on plan-time monitoring methods. In the remainder of this paper, we describe a general framework for execution-time self-assessment and present our initial implementation of this framework.

4. A SELF-ASSESSMENT FRAMEWORK

We can view the agent's mind as a decision procedure that takes as input the agent's goals, perceptions and knowledge and outputs an action to perform or a new goal to pursue (we will refer to these outputs collectively as *operators*). At a finer level of detail, we can view this decision procedure as a three phase process. In the first phase, potential operators are identified. In the second phase, operators are ranked

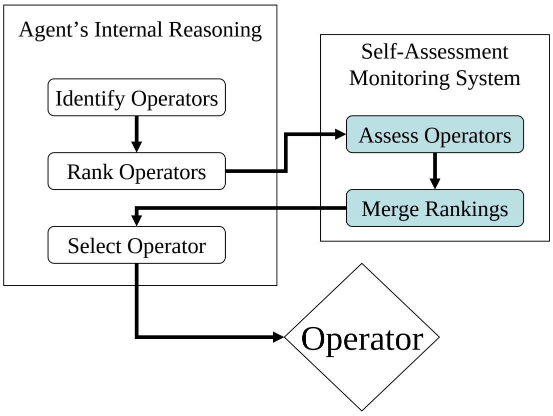

Figure 1: The Self-Assessment Framework

relative to one another. Finally, in the third phase, the most appropriate operator is selected and scheduled to be performed. Depending on the agent's implementation, these three phases may be controlled deliberately by the agent or may be controlled internally by the underlying architecture.

Our self-assessment framework fits on top of this three phase decision process. The framework interrupts decision making after the agent ranks potential operators. At this point, operators are re-evaluated with respect to the monitoring system's high-level domain constraint model. This evaluation creates a new ranking structure that is potentially inconsistent with the agent's preferences. As a result, rankings must be merged to ensure consistency with domain constraints while upholding the agent's preferences to the greatest degree possible. The merging process creates a finalized ranking of the potential operators[1]. This ranking can then be passed back to the agent's decision making process which will now select the best operator based on the finalized preferences (see Figure 1). This approach allows the monitoring framework to take into account the agent's own preferences (based on the agent's knowledge, goals, and perceptions), yet it also prevents errors by coercing decision making to ensure domain constraints are upheld.

The framework presented here can be used regardless of whether the agent performs deliberate planning and acting phases, however its most appropriate use will vary somewhat based on this property. Within a reactive agent system where the decision making process occurs in (soft) real-time, the framework will be used exactly as presented here. The agent will necessarily need to evaluate potential operators before making a commitment since it cannot rely on a pre-existing plan.

In contrast, a planning agent selecting operators at runtime does not necessarily need to evaluate potential options since it already has a formula (the plan) for how to behave. For such agents, the assessment framework may only be able to accept or reject the agent's desired operator (since the agent may not evaluate a set of potential options). If the assessment framework rejects the agent's intended operator, it must initiate replanning, otherwise the agent may fail to continue problem solving in a reasonable manner. With such agents, it may be beneficial to use the framework at plan-time as well as at execution-time. At plan-time, agents must evaluate potential operators to determine which are most appropriate for the plan. As a result, the self-assessment framework is likely to have better control over what operators are incorporated into the plan to begin with. Used in this fashion, the self-assessment subsystem resembles Atkins extensions to CIRCA [1, 2]. Note however, that the ability to use the framework without a distinct planning phase and with arbitrary constraint models makes the framework a generalization of CIRCA's specific approach.

5. DESIGN DECISIONS

Implementing the self-assessment framework described in Section 4 requires two major design decisions. First, we must determine an appropriate model for the high-level domain constraints used to evaluate and guide the agent's behavior. Second, we must select an appropriate agent architecture for our implementation.

5.1 A High-Level Constraint Model

An appropriate constraint model (CM) is critical because it determines the self-assessment system's ability to distinguish between correct and incorrect behavior. Previous work has examined simple logical sentences to define domain constraints [13], and explicit information about failure states [1]. Both approaches attempt to delineate correct and incorrect behavior based on the states that the environment, or the agent, may enter.

While a state-based approach may be sufficient in some situations, especially when a clear notion of failure states exists, it is insufficient to classify a wider range of interesting behaviors. In particular, agents such as TacAir-Soar that are designed to emulate human behavior are probably best characterized by the sequence of goals and actions pursued and not by the specific states they can or cannot enter.

To distinguish acceptable and unacceptable behavior sequences, we require a model that describes temporal relationships. Moreover, our selection must be informed by the following three competing requirements.

Simplicity The model must be easier to understand than the agent's internal knowledge representation. Otherwise, the task of constructing the model will be likely to introduce errors at the implementation phase (as is likely to happen when implementing the agent itself). This presents a recursive validation problem that must be avoided.

Constructibility The human expert may hold the only specification for correct behavior, but may be unable to communicate this specification in a meaningful manner. By limiting the data contained in our constraint model, we can use inductive techniques to construct a specification from observations of expert behavior. Thus we avoid requiring the expert to articulate the model directly.

Efficacy The model must allow the assessment system to identify and prevent salient errors. The previous requirements aim to make the model simple to understand and construct. However, both of these require-

[1] It is possible that none of the agent's proposed operators will be consistent with the domain constraints. In Section 6, we consider this scenario and present some strategies for handling such exceptions.

ments could have the effect of reducing the model's ability to distinguish correct behavior from incorrect behavior, thereby severely reducing its efficiency.

To balance these three requirements, we look to previous work on automated validation and knowledge base refinement. Recent work by Wallace and Laird [12] has examined a high-level model inspired by the hierarchical representations used in AND/OR trees, HTN planning [4] and GOMS modeling [6] to encode the variety of ways in which particular tasks can be accomplished.

Their model was designed with constructibility and simplicity in mind. In particular, it only relies on data that can be collected from annotated observations of task performance. This structure can be created automatically from performance traces and it has been shown to be PAC-Learnable (establishing the efficiency of creating and maintaining it). Finally, because this structure has been reasonably effective at identifying differences between two agents' behavior, it seems to make appropriate tradeoffs between each of the three requirements we consider here. We leverage this model, the hierarchical behavior representation (HBR), making minor changes that have the double advantage of increasing the HBRs representational power while also reducing the complexity of our self-assessment system.

The HBR is a tree with binary temporal constraints representing the relationships between the agent's goals and actions. It is defined as a node-topology (a hierarchical relationship between nodes) and a set of constraints (unary node type constraints and binary temporal constraints between siblings). In the HBR, internal nodes correspond to goals and leaves correspond to primitive actions. A node's children indicates the set of subgoals or primitive actions that are relevant to accomplishing the specified goal. In Wallace and Laird's work, unary constraints on the nodes specify a type of either AND or OR. These constraints in turn specify whether all or some of a node's children (subgoals and actions) are required to successfully accomplish the parent goal. In our model, we change these unary constraints to specify the frequency with which a particular operator occurs. Specifically, nodes are tagged as either SOMETIMES or ALWAYS. Semantically, a goal whose children are all ALWAYS nodes is equivalent to an AND node in the original HBR specification. Similarly, a goal whose children is some mix of SOMETIMES and ALWAYS nodes is equivalent to an OR node in the original HBR specification. It should be clear from this analysis that the model presented here is representationally equal to, if not more powerful than, the original HBR specification.

An example HBR is illustrated in Figure 2. Here, the subgoals `Destroy-Lead` and `Destroy-Wingman` are relevant for completing their parent goal, `Engage-Enemy`. The manner in which subgoals should be used to achieve their parent goal is encoded by the subgoal's node-type constraint (SOMETIMES vs ALWAYS) and the ordering constraints between subgoals. In Figure 2, SOMETIMES and ALWAYS nodes are represented with ovals and rectangles respectively. Binary temporal constraints are represented with arrows between siblings. From examining the figure, we can determine that the hierarchy specifies `Engage-Enemy` may be correctly accomplished by first accomplishing `Destroy-Lead` and then accomplishing `Destroy-Wingman`.

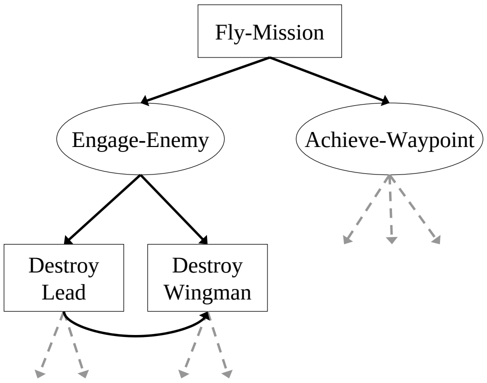

Figure 2: Behavior Representation

5.2 Agent Architectures

Having selected an appropriate high-level model of correct behavior for our self-assessment system, we now turn to the selection of an appropriate agent architecture. In some sense, this decision is less critical. We would expect it will simply make our implementation more or less difficult, as opposed to directly affecting the function of the system as a whole.

We have selected the Soar rule based system [10] as the initial agent architecture for creating our self-assessment system. Soar is a compelling choice for four reasons. First, and most critically, we are relatively familiar with the architecture and methods for creating Soar agents. Second, Soar has an explicit notion of the three phase decision process outlined in Section 4. Although it is easy to adapt other agent architectures such as CLIPS [3] to emulate the three phase decision process by adding small amounts of flow control knowledge, in Soar this is unnecessary. Third, Soar has been used for tasks that make use of explicit plan and then act phases as well as for creating much more reactive agents that choose operators in soft real-time. Finally, because the source code for the Soar architecture is publicly available, we could implement the self-assessment system either as a set of Soar rules or as a modification to the architecture itself.

6. THE S-ASSESS LIBRARY

We have implemented a self-assessment system that uses a high-level model of correct behavior to enforce domain constraints. As described in Section 5.1, the constraint model is a hierarchal representation that indicates the appropriate relationship between goals, subgoals, and primitive actions as well as constraints on the frequency with which operators are selected and constraints on their relative ordering. Our assessment system is implemented within the Soar agent architecture as a modular knowledge base, or library, called S-Assess. The library itself is a domain independent set of 62 Soar rules and follows the framework specified in Section 4.

The S-Assess library works under the assumption that a constraint model (CM) suitable for assessing behavior is stored in a specific location of the agent's working memory. A likely setup would require the agent's designers to create and load such a model before the agent left the laboratory.

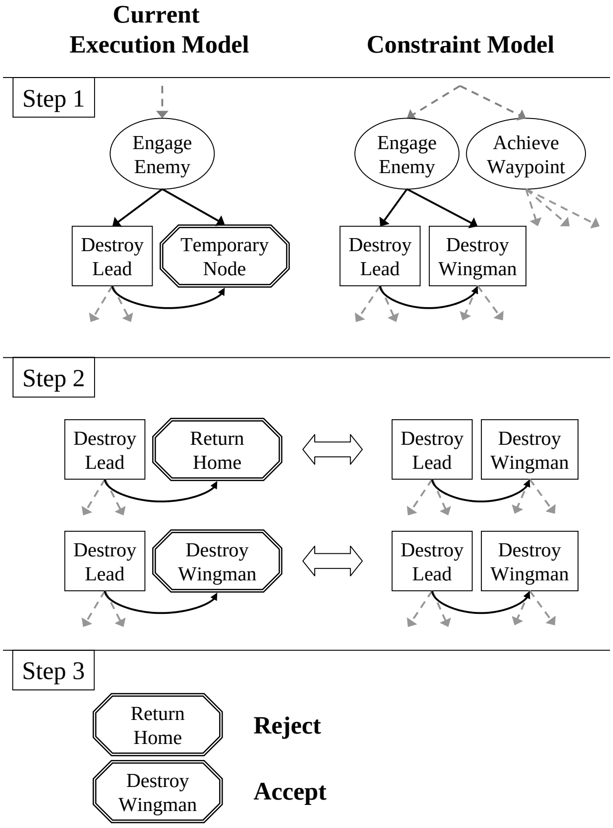

Figure 3: Model Comparison

Note, however, that there is no strict requirement that the constraint model be supplied by the designers. In fact, as we will see later in the discussion, the potential uses of the S-Assess library are greatly expanded by considering other scenarios.

As the agent performs its task, S-Assess traces which goals are selected and which actions are performed. In addition, it monitors when these events take place, and the relationships between goals, subgoals and primitive actions. As it does so, S-Assess builds a hierarchical execution model of the agent's behavior. The execution model (EM) is similar to the constraint model, but represents only behavior that has been executed up to the current point in time.

To evaluate potential operators, S-Assess intercepts the agent's decision making process as outlined in Section 4. The interception occurs just before a decision is made, when S-Assess augments the current execution model with a temporary node (Step 1 in Figure 3). Next, S-Assess iteratively associates one of the potential operators with the temporary node, comparing the resulting execution model with the constraint model to check for consistency (Step 2 in Figure 3). Note that this comparison is a simple process and does not require examining all nodes in each tree. Instead, we simply need to compare the temporary node and its siblings to their corresponding nodes in the constraint model.

The results of the comparison may reveal several situations. First, the execution model may be entirely consistent with the constraint model (i.e., siblings are ordered appropriately with respect to the ordering constraints, and all nodes of type Always are represented in the EM). In this case, the operator associated with the temporary node is consistent with the domain constraints and S-Assess takes no action. Second, the temporary node in the execution model may not have a corresponding mate in the CM. This situation indicates that the agent has suggested an inappropriate operator for the particular situation (i.e., for performing the parent-level goal). In this case, S-Assess rejects the agent's proposed operation. Last, the temporary node in the EM may create inconsistencies with the constraint model. For example, the EM may no longer uphold appropriate temporal constraints between sibling nodes. As before, this situation indicates the agent's suggested operator is inappropriate, and S-Assess must reject the candidate.

Each operator suggested by the agent is associated with the temporary node during the assessment process. In this way, the agent ascribes its own preferences to these operators and S-Assess can simply accept all operators consistent with the constraint model and reject all others (Step 3 in Figure 3). The net result is a (potentially) smaller set of operators which are consistent with the domain constraints and ranked according to the partial order specified by the agent. If, at the end of this process, S-Assess has excluded all operators suggested by the agent, it raises an exception to indicate that nothing the agent proposed can be performed safely. The library does not specify exactly how such exceptions should be handled, but a number of obvious possibilities exist. In particular, the agent may choose to do nothing; it may employ a new, more general, set of knowledge to suggest alternative operators; or it may delegate the decision to a human.

6.1 Interface

Leveraging the S-Assess library requires only the availability of a constraint model and minor commitments on the part of the Soar agent. Because S-Assess uses the same working memory space as the rest of the agent's knowledge base, the agent must be designed so as not to overwrite or in any other way modify the data structures used for assessment[2]. For most agents, this will be trivial to ensure, as the consistency computation uses an isolated area of working memory for performing its computations.

Additionally, agents must also use a specific mechanism for ranking potential goals or actions. Typically, in Soar, operators are proposed and then assigned architecturally recognized preferences that include unary relations such as "best" and "worst", and binary relations such as "better than". Once these preferences have been established, the architecture creates a partial ordering indicating the relative preference of each option. Normally, at this point the architecture automatically selects the best option to be pursued. As a result, S-Assess requires a minor modification to influence the selection of the agent's next operator. Specifically, we require that the agent assigns numeric preferences to operators instead of using Soar's built-in preferences. This has similar semantics, and easily allows one to establish a partial order of preferences over potential operations. However,

[2] As part of our future work, we plan to provide stronger guarantees that the data structures used by S-Assess cannot be modified by the agent proper. A conceptually simple approach would result from providing a read-only interface between S-Assess and the agent's standard working memory.

this modification also ensures that the architecture will not commit to a specific operator as soon as the (now numeric) preferences are asserted. As a result, the assessment framework has a chance to evaluate potential options and can then assign standard Soar preferences based on the finalized rankings. Thus, by this approach, all options are guaranteed to be ranked by both the agent and the assessment framework, and the final operator selection can still be performed via Soar's standard architectural mechanism.

The S-Assess library was built with generality and domain independence in mind. Although the preference scheme may require modest changes to existing Soar agents, it should not limit the engineer's ability to design agents for arbitrary tasks. Indeed, the CLIPS rule based system [3], uses numeric preferences called salience to make selections in a similar manner. Moreover, it is worth noting that agents designed to be used with S-Assess can still be run without the assessment framework. Specifically, we can use two simple Soar rules to convert the numeric preferences back into standard Soar preferences resulting in an agent that will select operations based entirely on its own beliefs. In this way, agents can easily be fitted to run with or without the execution-time self-assessment capability.

6.2 Evaluation

We tested the S-Assess Library on a simple agent with one high-level goal and six primitive actions. In our simulated world, the agent's goal is to satisfy its hunger by correctly getting, preparing and eating food, through the use of these six primitive actions. We modeled programming errors by allowing the agent's knowledge to suggest actions in any particular order, even if that order made little logical sense, or no progress towards the agent's goal. The result was an agent capable of trashing in the environment by pursuing action sequences that had no net effect (such as repeatedly preheating an oven) and that would achieve the goal only via a random walk. Given this agent implementation, we built six distinct constraint models that prescribed how the agent should perform its task. We included models with the following properties: total ordering over actions; partial ordering over actions; actions with the SOMETIMES property; partial ordering between SOMETIMES actions and ALWAYS actions; and restricting allowable actions to a subset of the agent's proposed set. We then examined the agent's behavior as it performed 50 instances of this "satisfy-hunger" task for each given domain constraint model. In all cases, S-Assess was able to ensure the agent's behavior was consistent with the constraint model eventhough the agent's knowledge alone would be insufficient to make this same guarantee.

6.3 Future Work

Although the current version of the S-Assess library provides enough functionality to illustrate its use and practical relevance, it is still in the early phases of design. In particular, the current implementation does not take maximum advantage of Soar's built in truth-maintenance system (TMS), opting instead to maintain data structures related to the quality of potential operators on its own. It is likely that leveraging Soar's TMS will have the added benefit of increasing the execution speed of the assessment subsystem while simultaneously making the library more compact.

Finally, we note that although we have only explored using the S-Assess library to perform run-time validation in the agent's operating environment, its potential uses are much broader. Briefly we introduce three such uses and leave further examination as future work.

First, consider a situation where the policies of the environment change over time (perhaps because new laws or rules are put into effect). Here, we might design an agent to consider performing its task in relatively general ways and use the S-Assess library to enforce the current environmental policy. This would allow the agent to adapt to new laws simply by obtaining a new constraint model from the environment. Second, a similar situation exists when agents must interact with one another. Here, we might use the S-Assess library to enforce a social policy that would establish safe and appropriate methods for agents to interact with one another. The policy might be obtained from the environment or from other agents involved in a specific transaction. Finally, we might allow callbacks from the assessment framework to either the agent or the external world. This would allow new approaches for dealing with constraint failure such as adjusting the agent's autonomy by seeking outside assistance. Although the details of these three applications differ, in each situation the self-assessment framework provides a mechanism for improving dependability and robustness of the agent it monitors.

7. CONCLUSION

In the near future, execution-time behavior monitoring will become an important tool for bridging the gap between the our willingness to trust new agent systems and our promised ability to produce agents that perform interesting and useful tasks. The work presented in this paper brings this goal one step closer by presenting a generalized framework for self-assessment and describing S-Assess, an initial implementation of this framework.

8. ACKNOWLEDGMENTS

The author would like to thank the anonymous reviewers for their insightful comments. The final version of this paper benefited from the questions raised by one reviewer in particular.

9. REFERENCES

[1] E. M. Atkins, T. F. Abdelzaher, K. G. Shin, and E. H. Durfee. Planning and resource allocation for hard real-time, fault-tolerant plan execution. *Journal of Autonomous Agents and Multi-Agent Systems*, March—April 2001.

[2] E. M. Atkins, E. H. Durfee, and K. G. Shin. Detecting and reating to unplanned-for world states. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 571–576, 1997.

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

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

[5] S. A. Hofmeyer, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[6] B. E. John and D. E. Kieras. The GOMS family of user interface analysis techniques: Comparison and

contrast. *ACM Transactions on Computer–Human Interaction*, 3(4):320–351, 1996.

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

[8] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, pages 41–50, January 2003.

[9] C. Ko, G. Fink, and K. Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of the Tenth Annual Computer Security Applications Conference*, pages 134–144, 1994.

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

[11] D. J. Musliner, E. H. Durfee, and K. G. Shin. CIRCA: A cooperative intelligent real-time control architecture. *IEEE Transactions on Systems, Man and Cybernetics*, 23(6), 1993.

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

[13] D. Weld and O. Etzioni. The first law of robotics (a call to arms). In *Proceedings of the Twelveth National Conference on Artificial Intelligence*, 1994.