Automated Monitoring and Validation of Synthetic Intelligent Behavior

Randolph M. Jones[1], Ben Bachelor[1], Webb Stacy[2], John Colonna-Romano[2], and Robert E. Wray[1]
[1]Soar Technology, 3600 Green Court, Suite 600, Ann Arbor, MI 48105, USA
[2]Aptima, 12 Gill Street, Suite 1400, Woburn, MA 01801, USA
rjones@soartech.com, ben.bachelor@soartech.com, wstacy@aptima.com, jcromano@aptima.com, wray@soartech.com

Abstract - *Validation of software models that emulate complex human reasoning has historically been informal, subjective, and difficult or impossible to scale to large numbers of models. This paper describes an approach to validation of intelligent behavior models (and semi-automated force, SAF, models more generally) that employs a formal knowledge representation (called a Behavior Envelope) to validate SAF behavior in both off-line and on-line modes of operation. The Goal Constraint System (GCS) employs constraint-based representations that enable requirement specification at different levels of abstraction and a penalty assessment approach that allows a subject matter expert to specify the relative importance of constraint violations. We describe the GCS language, interpreter, and several applications of the technology.*

Keywords: Constraint-based knowledge representation, software verification and validation, intelligent behavior modeling, synthetic intelligent forces

1 Intelligent behavior validation

Synthetic forces are an application of artificial intelligence that raise the quality of simulation-based Semi-Automated Forces (SAFs) to the level where they realistically emulate human tactical reasoning [1]. Because these models see use in realistic training and experimentation applications, validating the quality of modeled intelligent behavior is a key concern. The essential question to be addressed is “Given the richness and complexity of human behavior, how can we tell if any specific behavior produced by a synthetic force or SAF model is acceptably realistic?” Validation is one form of behavior evaluation that is useful for multiple reasons and at a variety of different levels of detail. For example, evaluation techniques are relevant to software verification, assessment of scientific contributions, assessment of student and SAF performance, runtime monitoring and intervention, evaluation of behavior-driven scenarios, and experimental assessment of new technologies. For the purposes of this paper, we will focus on software validation. In later sections, we will discuss approaches to extending our validation techniques to other types of intelligent-systems evaluation.

Validating human-level model behaviors (as well as actual human behaviors) is particularly challenging ([2] includes a thorough discussion of the challenges). Intelligent behavior is complex, and it is difficult to build automated validation tools that capture all the nuances of “good” and “bad” behavior. As a result, validation is often performed subjectively. Human subject-matter experts (SMEs) or training experts observe behavior and create assessments or scores based on their observations. Subjective validation allows the assessment of complex and nuanced behavior (to some extent), but it also has limitations:

- Subjective validation is inconsistent across different evaluators.
- Subjective validation is often inconsistent even when performed by a single evaluator.
- Validation criteria (i.e., requirements on behavior) themselves are often subjective and often not specified in detailed, archival, or formal form.
- Subjective validation is often qualitative or categorical, lacking nuanced assessment of individual decisions that led to a result.
- Subjective validation requires the availability of one or more human evaluators, which increases opportunity costs of validation.
- Depending on the complexity of the observed behaviors, validation may stress the cognitive capacity of the evaluator, especially for run-time validation.
- Validation may require deliberation that prevents the evaluator from keeping pace with the execution of the behaviors.

The overarching issues is that it is difficult and expensive to define objective requirements that capture all the nuances of human-level behavior. These limitations argue for an automated validation solution that provides rapid, consistent, and accurate assessments, using objective validation criteria than can be archived, inspected, and adjusted by humans. The primary gap for automation is having an inexpensive way to create formal requirements specifications to validate from.

We describe an automated behavior validation system called the Goal Constraint System (GCS). GCS works within a broader requirements-specification concept called *Behavior Envelopes*, being jointly researched and developed by Aptima and Soar Technology. Behavior Envelopes allow users to specify behavior validation criteria in the form of *behavior constraints* that hold within a particular *behavior context*. GCS comprises a formal representation language for Behavior Envelopes and a language for automatically scoring violations of constraints. It incorporates an implemented reasoner that infers the current behavior context for an entity, and monitors and scores observed behaviors for that context, either at run time or off line. We outline the need for formal behavior specification and the particular solution provided by the Behavior-Envelope concept. We then describe details of the GCS implementation and present several detailed examples of application of the GCS system, as well as higher-level descriptions of other GCS applications.

2 Expected behavior specification

From one perspective, behavior validation is the problem of measuring *observed* behavior against a formal characterization of *expected* behavior. There is a long tradition of formal behavior specification in computer science, particularly with respect to defining the requirements for the behavior of engineered software systems [3]. One of the primary advantages of a formal specification is that it requires unambiguous and objective descriptions of the desired behavior. This removes the elements of subjectivity, inconsistency, and ambiguity. Formal specification often provides the additional benefit that validation can be automated by computer software that is able to interpret the formal specification and match that specification to observable system behavior.

While these specification languages are formal, like computer programming languages, an additional advantage (usually) is that they provide a higher level of abstraction than a programming language does. This allows requirements builders to specify requirements for complex systems without having to duplicate all the work that would be required to build the system in the first place. However, for standard specification languages, this higher level of abstraction can leave the specification non-executable. This, in turn, can make system validation difficult to automate.

Property-oriented specification languages [4] facilitate automated validation. A property-oriented specification asserts particular relationships between elements of a system's data or behavior. A big advantage is that property-oriented specification does not need to be complete to be useful or to be automated. Automated software can monitor various assertions about properties and report any violations. Property-oriented specifications therefore have some appeal for application to the validation of intelligent behavior systems. However, there are two issues to be resolved in adopting a similar approach to validating complex intelligent behavior.

First, the context of a specification must be determined. In standard software engineering systems, assertions about behavior occur at the point in the code at which those assertions are applicable. This is feasible because, even for complex software, there are individual threads of execution that define the "location" of the execution logic at any point in time. In contrast, intelligent behavior involves much more loosely bound goals to be achieved, methods for achieving them, and processes for making sense of the world. In an intelligent system, all of these processes must interleave flexibly in ways that make it difficult to recognize a "state" for the system. (This behavioral flexibility explains in part why formalizations like state machines break down as behavior complexity increases.) We desire the ability to specify the context in which some property holds, including contexts that may not be entirely observable.

Second, intelligent behavior is rarely usefully classified as simply "correct" or "incorrect". Intelligent behavior is varied and flexible, and competence for accomplishing goals occurs in varying degrees. Thus, we desire a specification language that supports validation scoring functions that are not simply binary. The automated specification system should be able to indicate *the degree to which* an observed behavior meets the specification, rather than simply reporting that it fails to meet the specification.

3 Behavior envelopes

Aptima and Soar Technology have developed an approach to property-oriented specification languages that we term *Behavior Envelopes*. The approach builds and unifies work on scenario envelopes [5] and behavior bounding and variation [6][7]. A primary advantage of Behavior Envelopes is that they allow the specification of constraints to an arbitrary level of detail. As with property-oriented specifications for traditional software systems, this allows users to create inexpensive but useful behavior specifications, or to invest in more detailed specifications for particular intelligent behaviors.

Behavior Envelopes are a general concept to support a broad range of applications that all rely on the idea of a formal, declarative representation of behavior contexts and behavior constraints. For this paper, however, we focus on a particular implementation of Behavior Envelopes in GCS, targeted toward a smaller class of monitoring and validation applications (described below). A GCS Behavior Envelope consists of two primary components. The first component is a formal, relational representation of a situational context for an intelligent entity. The context can be considered a formula in predicate logic, composed of constituent predicates and propositions. The context may include observable features of the entity's situation (such as geographic location), as well as unobservable features describing the entity's internal state (such as a particular goal that the entity is trying to achieve). The second component is a formally specified set of constraints that the entity behavior should meet in situations where the behavior context applies. Each constraint is also a formal, logical predicate, often relying on relational predicates that related multiple properties together. GCS provides the machinery for making each logical predicate operational for a particular simulation system.

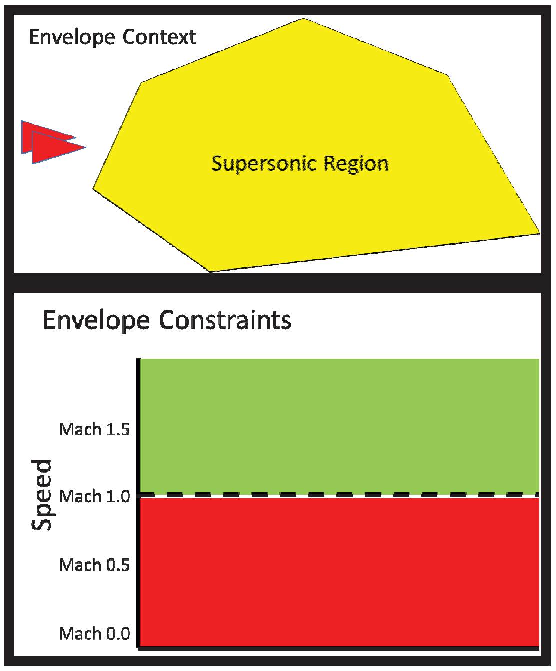

Figure 1. A Behavior Envelope consists of a Context and a set of Constraints. In this simple example, the Context specifies that the envelope only applies to entities who have a current position inside the Supersonic Region. The Constraints specify that any entities inside this region are considered to have acceptable speeds if the speed is above Mach 1.0, and unacceptable speeds if the speed is below Mach 1.0.

In order for the representations of context and constraints to qualify as "formal", a Behavior Envelope system must strictly define a set of domain-specific terms and predicates that compose to define each context and constraint. Thus, any particular instantiation of a Behavior Envelope system (such as GCS) must provide a well-defined language for specifying the conditions under which a particular envelope is relevant (Context conditions), as well as the conditions that must be met during the enforcement of a particular envelope (Constraints conditions). In addition, to operationalize these representations, they must be connected (via a programming interface) to the system generating the behavior. For example, a Behavior Envelope system that includes geographical regions for context must access a simulation interface that detects when entities are in particular geographical regions. GCS thus provides a level of formality necessary to enable objective, automated validation. We provide below examples of some of the formal representations built into GCS.

As a simple example, consider a Behavior Envelope that specifies a speed constraint over a geographic area. This type of constraint is common, for example, in training ranges. In such a case, the envelope context would consist of relations that specify the entity's position inside the controlled speed area. The envelope constraints would dictate the required speed limitations within this geographic area (See *Figure 1*).

A slightly more complex example envelope might describe the behavior expectations for an aircraft to fly a "racetrack" (oval) pattern. The envelope context in this case might be a complex set of conditions that specify when it is appropriate for the aircraft to be flying a particular racetrack pattern. These conditions would include mission specifications, the aircraft's geographic location, the position of the center of the racetrack point, the orientation expectations for the oval, and possibly some historical information about the entity (such as recent command-and-control messages).

Although these examples are simple, the GCS language supports arbitrary levels of complexity, for both envelope contexts and constraints. Users can create fine-grained envelopes when detailed validation is necessary, but complexity can be traded off in areas that do not require such detail. GCS thus supports a "spiral" approach to requirements, starting simple and increasing complexity as resources allow.

4 The Goal Constraint System

Thus, GCS provides a behavior validation capability (based on Behavior Envelopes) that enables user-specified validation of entity behavior at varying levels of detail. The GCS implementation consists of 1) the language that specifies Behavior Envelopes and 2) the executable system that interprets these envelops and uses them to monitor and validate observable intelligent behavior in a simulation. We describe each of these in this section.

4.1 The GCS language

The GCS language instantiates a specific version of the Behavior Envelope concept introduced above. The language supports the declaration of individual entities (i.e., the behavior generators), variables that bind at validation time to any of a predefined set of entities, and individual envelopes that define constraints on the behaviors of these individual entities or groups of entities. In GCS, the envelope context is primarily defined by a goal (or subgoal) that an entity (or group of entities) is attempting to achieve (note that this is the approach used in GCS, but Behavior Envelopes generally support a much wider range of context definitions). GCS provides methods to infer when an entity has spawned a subgoal from a previously active goal, or transitioned from

Table 1. A portion of an example GCS file for tactical air behaviors.

```
@fly_racetrack(=waypoint):
  **start -> @turn_to_inbound_racetrack_leg(=waypoint) within %time_delta minutes || inf
  **start -> @turn_to_outbound_racetrack_leg(=waypoint) within %time_delta minutes || inf


@turn_to_inbound_racetrack_leg(=waypoint):
  **start -> @fly_inbound_racetrack_leg(=waypoint) within %max_turning_time || inf

@fly_inbound_racetrack_leg(=waypoint):
  ^holds($has_racetrack_inbound_heading(=self)) || %medium_continuous_penalty per minute
  ^holds($at_cap_altitude(=self)) || %medium_continuous_penalty per minute
  ^holds($at_inbound_cap_speed(=self)) || %medium_continuous_penalty per minute
  $at_racetrack_distance(=self,  =waypoint)  ->  @turn_to_outbound_racetrack_leg(=waypoint)  ||
%large_one_time_penalty

@turn_to_outbound_racetrack_leg(=waypoint):
  **start -> @fly_outbound_racetrack_leg(=waypoint) within %max_turning_time || inf
```

achieving one goal to achieving a new goal. In contrast to alternative methods that assume agent goals are always explicit, GCS does not make that assumption. Thus, GCS can validate behavior generation methodologies that do not use explicit representations of goals, as well as those that do. This one of GCS' primary strengths, because it separates the representation of goals in GCS from the representation of goals in the behavior system. For example, GCS goals can be specified by SMEs who have no knowledge of a particular behavior system's implementation.

Each constraint set is a list of actions or goal transitions that the entity is expected to generate. A GCS Behavior Envelopes definition first declares the constants, variables, functional predicates, and goals relevant to the envelope. Then envelope constraint definitions specify relationships between series of functional predicates. GCS also elaborates the Behavior Envelope with a set of penalty functions. These penalty functions generate numeric scores when constraints are violated. Penalty values can be either "one shot", or they can accrue over time when envelope constraints are violated.

GCS defines functional predicates from an easily extensible set of native primitives. A GCS file that defines a set of constraints includes five sections: constant-value definitions, entity-variable definitions, predicate definitions, event definitions, and Behavior Envelope definitions. The Behavior Envelope definitions include the goal context for each envelope, the set of constraints for that context, and the penalty function for scoring constraint violations. Table 1 provides a snapshot of a portion of a GCS file describing envelopes for air combat behaviors. This snapshot does not include declarations or predicate definitions, but it gives two examples of Behavior Envelopes with constraints and penalty functions. One envelope's context is the "fly_racetrack" goal. This goal includes a subgoal constraint, indicating the conditions under which GCS infers an entity to be pursuing the subgoal "fly_inbound_racetrack_leg".

Constants provide symbolic references to constraint parameters that may take different values for different entities and scenarios. For example, a %radar_range parameter may specify the typical radar range of a particular aircraft, and the user could change this constant to validate a set of behaviors over a range of different aircraft. As another example, the %cap_orientation parameter can be changed when validating the behavior of different entities with different specific combat air patrol (CAP) missions.

Entities define the different types of objects over which GCS can specify constraints. Usually there is at least one entity variable representing the entity whose behavior is being validated. However, there can also be other "behaving" entities that interact with the primary entity of interest. Additionally, there can be "non-behaving" entities, such a geographical features (e.g., waypoints or boundaries) and physical systems (e.g., radar systems or weapons systems). Each type of entity has associated properties that can be included in envelope constraints. Another strength of GCS is that it can validate the aggregate behaviors of groups of entities or even entire scenarios, as well as individual-level behaviors.

Predicates define relations of interest that combine to build envelope constraints. Predicates are the primary form taken by individual constraints. A predicate is a relation that can either be met or unmet in a particular situation, and predicates can be constructed from other predicates, from events, and from primitive relationships (such as equalities and inequalities). An example simple predicate is "entity_is_airborne", which is met for an entity if the entity's altitude above ground level is greater than zero. A more complex example is "has_racetrack_inbound_heading", which is met if the reciprocal of the entity's heading is within some parameterized range around the mission-specified CAP orientation assigned to the entity (See *Figure 2*).

Events are similar to predicates, in that they also define relations to be included in envelope constraints. The difference is that the GCS Interpreter (described below) continuously checks whether predicates match. In contrast, for events, GCS only checks for transitions between being "unmet" and being "met". Once an event has been "met", the interpreter does not continue monitoring that event. An example event is "bandit_is_destroyed".

Goals define the contexts of the Behavior Envelopes, as well as the conditions under which GCS should transition to monitoring a new envelope (either via activation of a subgoal or transition from one goal to a subsequent goal). The current set of “active envelopes” (isomorphic to the current set of “active goals”) defines the constraints that should be monitored for a particular entity. An envelope is active as long as GCS assumes that the entity is still pursuing the goal that defines the context for that envelope.

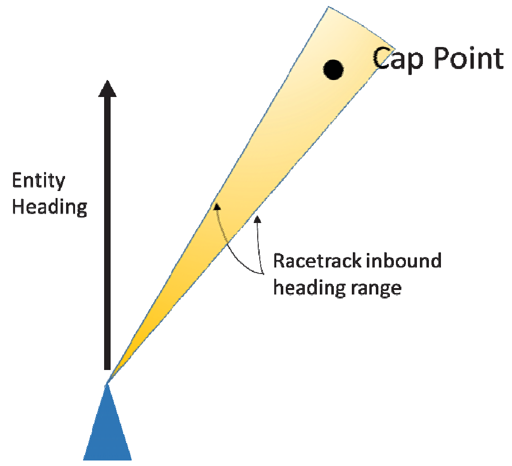

Figure 2. The "has_racetrack_inbound_heading" predicate is true if the entity's current heading lies within parameterized limits for approaching the Cap Point.

Each envelope constraint conjoins predicates and events, together with a penalty formula to use when the constraint is violated. Envelopes also include special “goal transition constraints”. That is, an envelope’s constraints can specify the conditions under which the entity “ought to” transition to a new goal. The GCS interpreter performs a search (using the penalty functions as heuristics) to infer whether the entity has started pursuing the behaviors associated with the new goal. The interpreter invokes the goal-transition constraint’s penalty formula if the constraints indicate that a goal transition should have occurred, but the entity’s behavior does not appear to be consistent with a transition to the new goal. Penalty formulas can have varied forms. Two primary forms are “one-time” penalties and “continuous” penalties. One-time penalties incur on the initial violation of a constraint. Continuous penalties accumulate over the duration of a constraint violation. For continuous penalties, it is also possible to specify a time period for the penalty accumulation (such as “per second”, or “per minute”). It is also possible to specify a penalty value of “inf”, meaning an infinite penalty, to represent constraints that are always expected to hold.

4.2 The GCS interpreter

The GCS Interpreter matches envelope contexts (goals) and constraints (specified in the constraint language) to streams of behavior data. The interpreter can run simultaneously with a dynamic behavior stream (e.g., during run time of a scenario) or offline on a collected data log file. The first job of the GCS interpreter is to infer, at any given time point, which goal(s) an entity is pursuing, indicating which Behavior Envelopes should currently be in force. The interpreter’s second job is to monitor which constraints are satisfied or violated for the set of active Behavior Envelopes. Finally, the interpreter uses the penalty scoring functions to compute penalty values for each violated constraint. Using heuristic search, the penalty values also allow the interpreter to infer whether the entity has made any changes in the goals it is pursuing. Ultimately, the interpreter generates a series of penalty values for each set of envelope constraints, reflecting how well the entity is meeting the behavior requirements for each active goal at each point in time. These penalty values aggregate into general validation measures of the behavior fidelity the entity generates for each Behavior Envelope.

The first implementation of GCS modeled entity behavior as a series of unknown goals with associated constraints. This version assumed that an entity could potentially transition from a current goal to any other goal, in response to some event. This approach naturally produced quite a large search space of potential goal sequences, which the interpreter navigated using the A* search algorithm. The interpreter used the GCS constraints and penalties associated with each envelope to compute a “best fit” score of the entity’s current behaviors to each particular set of envelope constraints. This allowed the interpreter to infer the goal sequence pursued by the entity during the course of a scenario. This approach was effective in analyzing off-line logs of behavior data. However, it was not efficient enough to produce timely validation of real-time behavior data streams.

For run-time validation, we refactored the GCS Interpreter, leaving the GCS language intact. Version 2 of the interpreter models entity goal sequences as a hybrid Hidden Markov Model / Finite State Machine. The interpreter tracks, in real time, the lowest possible penalty that could result in each possible goal being an active goal. Transition penalties between goals are recalculated each tick based on the current state of the simulation. Because a graph structure indicates the possible goal transitions, the GCS interpreter can efficiently track entity progress through potential goal sequences. This allows the solver to infer goals and compute penalties at low computational expense, allowing validation of both off-line data logs or run-time data streams. This approach does have the limitation that it cannot recognize goal transitions not specified in advance but, for synthetic force applications, such a limitation is typically acceptable.

5 Applications

The automated behavior monitoring, validation, and scoring that GCS implements can be used in a number of different applications. This section introduces potential applications and describes in more detail several examples of actual implementations.

5.1 Run-time SAF behavior corrections

If a constructive simulation scenario exhibits a run-time problem, such as an unexpected and incorrect SAF behavior, then an obvious remediation step would be to attempt to remove or reduce the violation while the scenario is still running. One way to use GCS would be to signal a human operator to respond to such a behavior violation. Alternatively, GCS could simply record violations in a log. We have been exploring a third option, in which violations inform adaptive technology that can attempt to "repair" the SAF behavior problem during execution (although this application can also do signaling and logging).

We are using violations detected by GCS to inform the Training Executive Agent (TXA), an adaptive operator-aiding technology [8]. The TXA is integrated with a distributed simulation, and it exploits the behavior representations of the simulator to modify the "native" behavior of its SAFs. The goal of the TXA is to reduce operator workload by automating the monitoring and management of a variety of interventions during the execution of a training scenario. GCS implements a major part of the monitoring and validation capability. As an example, one possible training objective is for trainees to intercept bandits moving at supersonic speeds. There is a geographic zone in which the bandit entities are expected to exhibit those supersonic speeds in order to support this training goal. Under "normal" conditions, a human operator might need to pay attention to aircraft entering this area and increase the speed if needed. The TXA reduces operator workload by automating these monitoring and management functions. GCS can be used to monitor entity behaviors and detect if any entity speeds are in violation of the expected speeds for the entity's geographical region and goals. If a violation is found, GCS uses its penalty functions to score the severity of the violation. This information is then passed to the TXA, which adjusts the entity's behavior in order to bring the behavior in line with the expected constraints.

In the longer term, the ability of GCS to assess penalties in real time, together with TXA's ability to permute SAF behavior, could be used in a more open-ended way ("emergent repair"). However, thus far we have used GCS only to identify violations of specific types and then trigger pre-defined TXA repairs. For example, an observed speed violation triggers a TXA directive that matches the prescribed SAF speed to range requirements at the entity's current location. The advantage of using GCS in this application is that the TXA can use the penalty score (and accumulating scores) to determine if/when to modify behavior, rather than simply recording and responding to a constraint violation. For example, for an aircraft briefly transiting thru the supersonic area, GCS can apply a different penalty function (based on context) than a bandit that is intended to engage the trainees.

5.2 Validating training scenario goals

SAFs may perform actions that are appropriate from a tactical perspective, but that may not match the specific required actions for a particular training context. For example, imagine a training goal intended to provide a trainee with the experience of two successive air-to-air intercepts. In this situation, the trailing bandits need to stay far enough away from the lead group to not interfere with the initial intercept but close enough that the trainee has to engage them immediately on successful prosecution of the first group. This is the instructor's intent for this scenario.

On the TXA effort, we are using the GCS to provide objective assessment of the "presentation quality" (from the perspective of instructor's intent) of a scenario as it evolves under a range of experimental conditions [9]. If all the various interactions in support of training goals execute as desired, then the scenario can be scored as having high quality. Various undesired interactions can impact the overall quality of the training scenario to different degrees. Subject-matter experts defined scoring criteria for each experimental scenario, and then we encoded the constraints and scoring functions into GCS. This allowed us to run a series of experimental variations to a training scenario, and to compute the presentation quality for each experimental condition in an automated and objective fashion.

5.3 Verification and validation of SAFS

If a SAF behavior results in some kind of (tactical or instructional) constraint violation, then there are additional potential responses, besides attempting to repair or adapt the current scenario. When GCS detects a poor behavior, this could be interpreted as a signal for repairing or refining the underlying behavior representation for future use (and thus eliminating the need for run-time repair in the future). We have investigated the potential for behavior refinement from several different perspectives. Most importantly, we have developed a test harness that enables systematic variation of parameters within a SAF behavior representation. The test harness is able to replay a given scenario with specific variations and combinations of behavior parameters. The test harness uses GCS to validate, score, and summarize the resulting behaviors. The results of this analysis indicate the range of "presentation qualities" that a SAF behavior definition can produce, as well as a sensitivity analysis of the behavior definition to various parameter settings.

Systematic variation of parameters helps modelers, operators, and instructors develop an understanding of the "topology" of a behavior model. Although such topology analysis has not seen significant application, the testbed investigations with GCS demonstrate that this type of analysis is feasible and worthwhile, especially for generalized behaviors. Using this approach, we foresee a much larger future role for GCS in facilitating behavior verification and validation. Systematic variation using GCS may be especially useful for learning systems that attempt to synthesize behavior representations from observation data [10].

5.4 Additional applications of GCS

There are additional potential applications of GCS that we have not yet implemented or explored significantly. For example, the search-based inference engine in the GCS interpreter can also be viewed as a form of plan recognizer. This type of plan recognition can be used to infer the intent of constructive force models without having access to the models' internal representations (as above), or also to generate explanations of observed behaviors when we do have some knowledge of a model's internal representations. The use of GCS to support the TXA experiments demonstrates the ability to go beyond validation of individual entity behaviors to validating aggregate behaviors at the scenario level. Additionally, GCS could assist in the creation of suitable scenarios or the run-time adjustment of scenarios (extending the functions of the TXA).

6 Summary and conclusions

We have introduced the problem of generating useful, objective, and automated validation of complex intelligent behaviors in modeling and simulation. The fluidity and complexity of human-level behavior requires validation solutions that extend techniques for standard software systems. We have also described and illustrated one implemented solution to this problem. The Goal Constraint System relies on the concept of Behavior Envelopes, which is itself an adaptation of property-oriented specification to the complexities of intelligent behavior systems. The GCS representation additionally extends the Behavior Envelope concept to accommodate unobservable features (entity goals) and quantitative scoring of constraint violations. The GCS interpreter exploits its penalty scoring functions to assist in inference about entity goals, as well as to produce quantitative scores of individual and aggregate entity behaviors. We have identified a number of application areas in which such automated and objective validation is useful.

7 Acknowledgments

This work is supported in part by the Naval Air Warfare Center Training Systems Division (Aptima Prime Contract: N68335-12-C-0146) and the Office of Naval Research project N00014-1-C-0170 Tactical Semi-Automated Forces for Live, Virtual, and Constructive Training (TACSAF). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Department of Defense or Office of Naval Research. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

We would like to thank sponsors at NAWC TSD and ONR who have provided insights and operational perspectives in the development of GCS: Beth Atkinson, Heather Priest, Melissa Walwanis, Amy Bolton, and LCDR Brent Olde. GCS also benefitted significantly from the contributions of J.T. Folsom-Kovarik, Leon Pryor, Marcus Huber, and Sam Wintermute at Soar Technology.

8 References

[1] 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), 27–41.

[2] Wallace, S. (2003). *Validating Complex Agent Behavior*, Ph.D. Thesis. The University of Michigan, Ann Arbor.

[3] Lamsweerde, A. V. (2000). Formal specification: A roadmap. In *Proceedings of the Conference on the Future of Software Engineering, ICSE '00*, 147-159.

[4] Dasso, A., & Funes, A. (2009). *Formalization process in software development*. IRMA International.

[5] Stacy, W., Picciano, P., Sullivan, K., & Sidman, J. (2010). From flight logs to scenario: Flying simulated mishaps. In *Proceedings of Interservice/Industry Training, Simulation, and Education Conference* (I/ITSEC 2010). Orlando, FL.

[6] Wallace, S., & Laird, J. E. (2003). Behavior Bounding: Toward Effective Comparisons of Agents & Human Behavior, *International Joint Conference on Artificial Intelligence*.

[7] Wray, R. E., & Laird, J. E. (2003). Variability in Human Behavior Modeling for Military Simulations. *Proceedings of the 2003 Conference on Behavior Representation in Modeling and Simulation*. Scottsdale, AZ. May.

[8] Wray, R. E., & Woods, A. (2013). A Cognitive Systems Approach to Tailoring Learner Practice. In J. Laird & M. Klenk (Eds.), *Proceedings of the Second Advances in Cognitive Systems Conference*. Baltimore, MD.

[9] Wray, R. E., Bachelor, B., Jones, R. M., & Newton, C. (to appear). Bracketing human performance to support automation for workload reduction: A case study. Accepted for publication in *Proceedings of the Human Computer Interaction International (HCII) Conference 2015.*

[10] Levchuk, G., Shabarekh, C., & Furjanic, C. (2011). Wide-threat detection: Recognition of adversarial missions and activity patterns in Empire Challenge 2009. In *Proceedings of the SPIE Defense, Security, and Sensing Conference*.