A Case-Based Behavior Design Aid for OneSAF

Douglas A. Reece
Jenifer McCormack
Jackie Zhang
Science Applications International Corporation
12901 Science Drive
Orlando, FL 32826
reeced, mccormackj, zhangj @saic.com

Keywords: case based reasoning, OneSAF, behavior composition

ABSTRACT: This paper describes an analysis of the behavior models in One Semi Automated Forces (OneSAF) and a proposed design aid tool that will use Case Based Design (CBD) to allow behavior designers to reuse existing models and find related models to adapt and modify to create new behavior. OneSAF has a large and growing repository of primitive and composed behaviors, and there are challenges for a behavior programmer faced with creating a new behavior. A design tool can help identify primitives which need to be modified and provide an example for how they are correctly used in a composition. Such a tool is expected to greatly reduce the effort of composing new behavior.

1. Introduction

This paper describes an analysis of the behavior models in One Semi Automated Forces (OneSAF) and a proposed design aid tool that will use Case Based Design (CBD) and other techniques to help behavior designers more easily reuse existing behavior models. The design aid will help find existing behavior components that are related to a target behavior and demonstrate how target behavior requirements can be implemented. Improved ease of reuse will not only make designing the target behavior less expensive, but will also help to maintain a consistent design style in the behavior repository.

1.1 OneSAF

The US Army is developing OneSAF as its next-generation Computer Generated Force. As described by the OneSAF Operational Requirements Document (ORD), OneSAF will be a composable, next-generation Computer Generated Force (CGF) that can represent a full range of operations, systems, and control processes from the individual combatant and platform level to fully automated BLUFOR battalion level and fully automated OPFOR brigade level. Unit behaviors will be modeled to the BLUFOR battalion and OPFOR brigade level for selected units, and command entities are to be modeled to the BLUFOR brigade and OPFOR division level. OneSAF will have a variable level of fidelity. It will accurately and effectively represent specific activities of combat, combat support (CS), and combat service support (CSS) and command, control, communications, computers, and intelligence (C4I) (Courtemanche and Wittman 2002). In addition to providing a next-generation simulation for the Army, OneSAF supports all services; it is being used in the Marine Corps’ Combined Arms Command and Control Training Upgrade System (CACCTUS) and is being considered by the Joint Forces Command to model ground forces in exercises such as Urban Resolve.

In order to support such a wide range of applications, OneSAF was designed to be highly composable. The application itself is composable from a product line of components, and the simulation works with repositories of entities, units, and behaviors. The OneSAF system includes graphical editors that are used to compose these entities, units, and behaviors. The focus of the work described in this paper is on the composition of behaviors using the language provided by the graphical behavior editor.

OneSAF has been under development for over two years, and is expected to be released early 2006. A project such as the behavior design aid effort described in this paper is faced with trying to analyze a system that is immature and undergoing continual additions and improvements. However, the behavior repository is becoming more stable and a number of design conventions for behavior models have been established. While some of the specifics of the analysis we have done to date will undoubtedly become obsolete over the next year, the larger patterns are expected to remain relevant. The work described in this paper was done using “block C” OneSAF software which is being released in April 2005.
1.2 The OneSAF Behavior Modeling Infrastructure

There are three types of software components that govern the behavior of an actor in OneSAF: physical agents, behavior agents, and behaviors (Henderson and Rodriguez 2002). Physical and behavior agents are components of an entity that perform low-level actions and controller functions. Behaviors are software objects that can be executed by the agents in an actor.

Behaviors comprise behavior elements such as primitive behaviors, predicate functions, message senders, and complex behaviors. The elements are arranged in program that may include sequential, parallel, conditional, and looping constructs according to a defined grammar. In OneSAF this program is called an execution timeline. Behaviors are a procedural description of doctrinal behavior. Procedural behavior definitions such as this have are used in many military and commercial simulations (Calder, Smith et al. 1993; von der Lippe and Courtemanche 1998; Fu, Houlette et al. 2003; Softimage 2005).

Several of these procedural behavior systems, including OneSAF, provide an interactive, graphical tool for creating behaviors out of the behavior elements. Behavior elements are selected and arranged with the desired control flow. In OneSAF, the end result is an XML file that is saved to a behavior repository. With a graphical user interface (GUI)-based composition tool, behavior designers can create complex behavior without having to write source code. With OneSAF, this means that military subject matter experts (SMEs) could create new behavior without having to program in Java. Software engineers (SWEs) are still required for creating new primitive behaviors when the target behavior is outside the scope of existing primitives.

Although OneSAF and other such systems define a structure for behavior and a tool for composition within this structure, behavior in the target domains is still complex and there are many details that must be addressed when composing a behavior. These factors continue to make behavior design a difficult process.

1.3 Reusing Behavior

OneSAF’s XML behavior description files are stored in a repository and made available to behavior designers for composing new behaviors. Ideally, designers will be able to reuse the behaviors in the repository—especially the primitives—when designing a target behavior. Unfortunately, this ideal is not always easy to achieve in practice. There are several obstacles to the easy reuse of the behaviors in a repository. For example, a behavior programmer must search through a large number of behaviors to find those that can implement or contribute to the target behavior. If the behaviors are not indexed well, the search could take a long time, and even after a long search the programmer might miss relevant behaviors. Furthermore, the approach that the programmer wants to use to decompose the target behavior might not match the decompositions supported by the existing behaviors. The programmer might in fact discard a relevant behavior factored in a different way and compose a new behavior that does much the same thing but looks different.

1.4 Case Based Design

We are investigating how a design aid could be used in conjunction with a behavior composition tool to make behavior design easier. One approach we are using is Case Based Design (CBD). A CBD system attempts to find existing designs that match the target design problem. A successful match provides a complete, working solution with all details filled in already. If the found design case does not match the target exactly, the system attempts to adapt it to the target problem. If the adaptation fails, the CBD system attempts to learn why it failed and how to anticipate the same problem in the future. Significantly different cases may be added to the case base for future use. (Aamodt and Plaza 1994) provide an analysis of Case Based Reasoning (CBR) characteristics and methods in general; (Summers, McLaren et al. 2004) and (Reece, McCormack et al. 2004) discuss the application of CBR to behavior composition. CBD is a form of CBR applied to design synthesis vice classification or diagnostic types of problems. In the behavior design application, the target problem is not expected to match an existing case, because (unless there is a mistake) target behaviors will be new behaviors not already in the behavior repository. The CBD system would thus have to assist in the synthesis of a new solution from existing cases or parts of cases.

Our intent in exploring CBD is to help find and reuse behaviors in the repository. A retrieved case would provide a behavior designer with a starting point that is close to the target behavior. The case would be a working behavior, providing an example of how all of the design details can be set. Behaviors built from existing cases would hopefully be more consistently designed and therefore more likely to support future reuse.

1.5 Paper Outline

The remainder of this paper describes our work in analyzing the OneSAF behavior modeling approach and developing concepts for a design aid. The next section discusses OneSAF behavior modeling and explains the
difficulties in composing new behaviors from primitives in the OneSAF repository. Section 3 describes our design aid concept, and Section 4 shows some of the prototype work that has been completed so far. The next sections discuss future work and conclusions.

2. Analysis of OneSAF Behavior Modeling

2.1 Design Objectives for OneSAF Primitives

One of the stated objectives for factoring military tasks into OneSAF primitives (Karr 2003) is to develop a language for non-programmers to develop executable representations of behavior. The primitives in the language should be part of the problem domain, i.e. military tasks. The primitives should allow military subject matter experts to create or read composite behaviors without having to know about data structures, data types, software interfaces etc. Thus primitives should not perform strictly software tasks such as manipulating data structures or converting data types. Another objective is to limit the number of primitives. The desire is to create primitives that are used in many composite behaviors, rather than having special primitives for each high level behavior. Thus many similar composite behaviors will use the same, recognizable primitives even though they do somewhat different things.

The first objective, no “programming” primitives, is being met by encapsulating low level data processing actions with other tasks into more complex primitives. For example, in an infantry behavior to mount a vehicle, a single PlanMount primitive combines several processing steps: listing unit subordinates, converting the vehicle name to an entity identifier, notifying the vehicle entity of the action, and converting a three dimensional entity location to a two dimensional move destination.

The goal of keeping “programming” concepts out of primitives extends also to predicate functions, used to select variations in behavior and to control repetitive behavior. Predicate functions generally test the truth of “facts” in OneSAF. Facts are really functions themselves, and can compute values from simulation events, terrain characteristics, entity status, and so forth. The details of the implementation can be hidden from the user, and a domain-relevant description such as “fired upon” presented instead. In some cases, conditional branches in a behavior test the truth value of a Boolean value computed in a primitive behavior; here again, the details of the computation are embedded in the primitive behavior, and are not presented to the user.

The second objective, creating a limited set of primitives, is also accomplished by combining related actions into single primitives. There are three approaches used in OneSAF (Karr 2004). The first is to identify generic actions that are independent of the task and make those primitives. A good example of this is TimeDelay, which is used commonly in low resolution behaviors to model an action just by allowing time to pass and then changing the status of the entity or environment. The second approach is to include variations of the same behavior into the same primitive. In OneSAF, such primitives are called “semantically rich” primitives. The different actions that these primitives take generally share many of the same steps and algorithms. The PlanCrossCountryMove primitive is an example of this approach; the core path planning and formation definition actions are augmented with variations that allow different bounding techniques, different observation schemes, etc. One or more selector inputs in a semantically rich primitive determine which variation the primitive performs. The third approach to limiting the number of primitives is to collect together primitive actions that have a common theme into one “centralized” primitive behavior. For example, one theme is placing objects in the environment; primitive ConstructEmplace is used to place obstacles, explosives, mines, and even holes in the environment. As with semantically rich primitives, these primitives use a selector input to determine which action is performed.

The desire to limit the number of primitives has led the OneSAF behavior modeling team to try to factor the tasks of the battlespace in such a way that most new behaviors can be composed of primitive actions that are already implemented in primitive behaviors, or that should logically be extensions to existing centralized or semantically rich primitives. In other words, it is expected that after most of the breadth of the domain has been touched by some implemented behavior, few if any new primitive behaviors will have to be added.

2.2 Challenges in Designing New Behavior with Existing Primitives

The OneSAF design approach of using complex primitives allows the system to meet the objective of keeping “programming” details invisible to the user who is building new behaviors with the behavior composer tool. However, this approach has another side: it also makes it difficult for the user to build new behaviors with the composer because the atomic steps in a behavior are all encapsulated inside the primitive behaviors. To change the atomic steps, it is necessary to change the Java code of the primitive containing the steps. To extend a semantically rich or centralized primitive behavior to perform a new function, it is necessary to define a new value for the selector input, and extend the Java code of the primitive to handle the new case. It may also be
necessary to add extra inputs to the primitive to handle new, different data.

The presence of selector inputs in primitives presents another challenge as well. Since one primitive may perform several functions, when a programmer is searching for an existing primitive to use in a new behavior, he or she is forced to examine each function in each primitive. Unlike the primitive itself, the different functions are not described by a name or by metadata. It is thus much more difficult to find all of the different behaviors available.

These observations about OneSAF primitives were confirmed in informal experiments that we performed in which we observed programmers creating a new behavior using the OneSAF behavior editor. At some point, the programmers invariably say that they need to look at the Java code for a primitive to determine what it does.

This problem for composing behavior using existing primitives is also present in existing composite behaviors; in order for a user to know just what a composite does, he or she usually has to view the behavior using the graphical editing tool. In fact, this problem is not particular to OneSAF, but is present in any system that defines behavior abstractions. The exact semantics of a behavior abstraction is determined by the lowest level of executable instructions in the simulation. Executing or viewing these instructions may always be necessary, even if the behavior has attached metadata describing what it does.

2.3 The OneSAF Behavior Repository

The previous section described the inherent difficulty in reusing behavior using only an abstract description. As noted in Section 1, relevant behaviors can be hard to find in a repository even when descriptions are available. The current repository in OneSAF is set up as follows: there are about 70 primitive behaviors, each described in an XML file (which itself refers to Java code). These files are spread among several directories, each corresponding to a different level of modeling “fidelity.” At this time it is not clear how behaviors are classified with respect to fidelity. Similarly, there are over 150 composite behaviors in OneSAF block C; they are also XML files arranged in different directories.

When a user wishes to look for a behavior that satisfies some criteria, he or she opens a dialog box in the editor. The dialog box displays files and directories in a manner similar to the familiar Windows Explorer. The user must select one directory at a time and then examine the file names to choose a behavior. The file names, which are chosen by behavior programmers, are fairly descriptive, but selecting behaviors this way is still often difficult. The files are arranged alphabetically rather than by topic or other relevant criteria; also, the names generally only encode the general topic of the behavior, but not other information needed by the designer.

3. A Design Aid Concept

3.1 The Design Process

We have noticed from our own design behavior experience and from observing OneSAF behavior modelers that a top-down approach is an effective way to develop behavior. It is also clear that behavior modelers attempt to use existing behaviors and that as a result they spend much of their design time examining behaviors to determine exactly what they do, and whether they would be appropriate targets for reuse with or without modification. As noted above, these examinations can include inspection of the Java code.

Our aim in creating a design aid tool is to support these design activities: top down design, search for reusable behavior, understanding what behaviors do and how they work, and modification of behaviors.

The approach is limited to the behavior design language of the behavior composer, i.e. it does not attempt to aid in Java programming. We had originally hoped to be able to aid an SME in constructing new behaviors entirely within the language of the graphical editor (Reece, McCormack et al. 2004), but the OneSAF primitive behavior modeling approach described above prevents this. The OneSAF architecture and behavior composer would support any set of primitives, but our goal is to support the reuse of the behaviors developed over the last two years rather than creating an entirely new set.

3.2 Design Tool Approach

Our approach to building a design aid is to help the user find existing behaviors that can potentially be reused and adapted, and to help the user understand what the behaviors do and how they do it. The first part of the approach uses CBD concepts to identify behaviors or parts of behaviors that match behavior specifications and either form the basis of the target behavior or provide an example of how part of the target behavior can be designed. The second part uses improved visual displays to more directly provide the user with the information they need to see how data flows through the behavior and identify changes that need to be made to behavior components.
The expected design process with the design aid is as follows: starting at the top level of behavior description, the user will use requirements for the target behavior (provided by manuals, SME personal knowledge, etc.) to determine if there is a behavior in the repository that satisfies the requirements. The design aid will allow the user to specify keywords and metadata values and search the repository for matches, ordering the results by closeness of match. The user can then browse the behaviors, examining their control flow, input variables, metadata descriptions, and so forth.

If a repository case does not match the target behavior (which is the expected situation), the design process begins. First, the user may determine that an existing case could be adapted to the target. This is the situation in which users commonly seek to understand exactly what the behavior and its components do. The design aid helps with this understanding by highlighting potentially key primitive behaviors, key behavior inputs, and data flows between behaviors. Second, the user could start from scratch composing a new behavior. In this case the user could query the design aid to find behaviors whose control flow structure matched the target, or which contained relevant primitives. The aid could also point out behavior inputs, such as TaskType inputs, which would likely have to be modified in a new behavior.

Once a behavior has been proposed, the user can identify components (primitive or composite behaviors) in the behavior that would require functional modifications; for example, a planning primitive might need to handle a new task, or a communication primitive might need to handle a new message type. If one of the components needed to be modified, or if a new component had to be created, the design process would recursively consider that component—looking for a match for that behavior, gaining understanding of the component behaviors, identifying component behaviors requiring modification, etc. Figure 1 illustrates the design process with the design aid.

3.3 Matching

Finding behaviors and behavior components that are relevant to the target behavior and that can be reused is a major challenge for CBD. The most obvious way to search the repository for behaviors is to compare the name and description (in metadata—when it is available) to keywords describing the target behavior. Among programmers who have been exposed to the design aid prototype, this is probably the most commonly used feature. In addition to keywords the design aid uses domain functional areas—based on battlefield operating system (BOS) areas, equipment areas (tank, aircraft, ), etc.—and the intended size of the unit using the behavior as indices for identifying matching behaviors. Each index that produces a match increases the match score of a behavior in the repository; the tool lists behaviors in order of their match score for the user to see. This approach is described in detail in (Reece, McCormack et al. 2004).

If a behavior-case from the repository does not match the target behavior requirements, the user has to modify it or create a behavior from scratch. The matching process may identify individual behaviors to be used in the target behavior (possibly with modification). In addition to identifying these components, the user has to specify the flow of control. Existing behaviors can also provide examples of how control flow can be designed to satisfy behavior requirements. The design aid can retrieve appropriate design cases by matching the control flow
design patterns in them to features selected by the user. We have identified several design patterns in the existing OneSAF behaviors that represent standard functions. For example,

- **Wrapper**—a composite is constructed to allow a user to invoke a primitive (users cannot order an entity to execute a primitive directly); it does nothing more than call the primitive. Or, for a primitive that has a selector function, a composite may always invoke a specific behavior by setting the selector to a constant value.

- **Unit and entity behavior**—some composite behaviors are designed to be run on either a unit or an entity. Inside the behavior there is a test to determine whether the actor is a unit or entity, and then the behavior either executes a primitive on the entity, or recursively commands subordinates to perform the behavior.

- **Repeat execution**—this is done in several ways. First, a list of items can be traversed using the GetNextItem primitive (which also outputs a flag indicating whether there are more items on the list). Second, some primitives such as PlanCrossCountryMovement work incrementally until the task is complete (i.e., bound until the destination is reached) and output a flag indicating whether they need to be called again.

- **Subordinates execute in parallel**—some composites have two groups of subordinates execute different tasks in parallel.

### 3.4 Understanding Behaviors

When a programmer is trying to understand how a behavior works, he or she typically examines the attributes of the component behaviors in the behavior. The attributes include metadata which describes what kind of units is intended to run the behavior and (in free text) what the behavior does. In block C, most of these descriptions are not filled in. (See comments on this below). The attributes also describe the input variables of the component behavior and where the input value comes from in the behavior. For example, the input may take its value from the behavior input parameters (appropriate for a movement destination, for example), from the behavior constants (appropriate for a TaskType input), or from the output of another component behavior (typical of an execution primitive that is preceded by a planning primitive).

The design tool can provide several aids to a user examining a behavior. First, “key” primitives can be identified. Primitives with inputs that take enumerated values—which, discrete values—and that get their values from a behavior constant are generally perform different functions with different input values, and are specialized in the given behavior by the constant value. This is especially true of primitives that have a TaskType input. These primitives would likely have to be expanded to support a new behavior. Our prototype tool can identify and highlight such key primitives in a behavior, and also display on the same screen the names of the input variables that have constant inputs.

Second, viewing the data connections between primitives can be useful to see how primitives work together and how they depend on one another. Data that flows to conditional branch components also affects control flow. The design aid can determine connections between primitives from the XML behavior description and display them in summary form.

Third, the design aid can show other examples of how selected primitives are use in behaviors. An examination of the current OneSAF primitives and composites reveals that the composite behaviors are often built in a particular way to match the inputs and outputs of a complex primitive. For example, the ClearRoom composite starts with the FindInteriorCombatPositions primitive, which has outputs specifying movement of a team to stacking positions, movement into a room, and movement within the room. There is also an output specifying how a grenade is to be used. Not surprisingly, in the ClearRoom composite the Find primitive is followed by primitives to execute the moves and the grenade toss. In our design aid prototype it is possible to select a primitive from the list and see composite behaviors that use it highlighted on another list; the user can select one of the highlighted composites to see how it uses the primitive.

There are undoubtedly other visual tools that can be developed to help the user understand behaviors. We have even considered parsing Java files to identify certain simulation actions and present them to the user. Further experiments with the tool are expected reveal useful design aids.

### 3.5 CBD Issues

The design aid concept we have described above can take behavior descriptions in the form of keywords, battlefield operating system identifications, etc. and find matching behaviors in a repository. The user can easily browse through the best matches, viewing their attributes and seeing a graphical representation of their control flow. This is the basis for a CBD system and is similar to other design systems such as CASECAD (Maher and Gomez de Silva Garza 1996).
Many CBR and CBD systems used representations of cases that include domain-specific attributes which are useful for matching, for selecting adaptation strategies, for detecting and analyzing solution defects, and for learning which new cases to add to the case-base. These case representations (and adaptation knowledge, etc.) are encoded by the system designers. Our goal is to make a design aid that works with an ever-broadening domain of cases and target behaviors, and without having the designers around to develop new domain-specific attributes. Therefore we have chosen to use the existing OneSAF representation for behaviors and the repository infrastructure for the case base. This means that all of the indices used for matching, all attributes used in visual behavior-understanding aids, and all attributes needed for adaptation, revision and learning have to be derived automatically from the OneSAF representation.

For example, it would be possible to annotate a primitive behavior with a bit of knowledge that enumerated what actions the primitive could perform, what data it needed as a precondition to executing, etc. An automatic reasoner could use such information to detect plan failures in a new plan. However, if a behavior designer creates new Java code for that primitive, then a designer would have to modify the knowledge by hand according to the function of the new code. This cannot be done automatically because there is no way to know the semantics of the Java actions.

The OneSAF behavior representation does include metadata for all behaviors, including the types of military unit the behavior is appropriate for, and even a free text description of the behavior. However, these are not scheduled to be populated with useful information until the OneSAF block D release next year. We note, however, that metadata descriptions are a form of software documentation, and documentation is often neglected in software projects. Even if the OneSAF development team fills in all of the missing data for the block D release, there is no guarantee that other OneSAF developers in the future will also do so. Therefore, we have chosen not to have the design aid depend on any specific piece of metadata for its operation.

4. Tool Prototype

We have created an extension to the OneSAF behavior composer that allows us to prototype design aid functions. Figure 2 shows the tool window with a query panel in the upper left and a list of composite and primitive behaviors below it (primitives in the bottom panel). In this case, the tool is using behavior metadata and several characteristics (beyond what has been described above) derived from the XML description to score and rank behaviors. One composite behavior has been highlighted, and that behavior is displayed graphically in the panel to the right.

5. Future Work

The design of this design aid is an ongoing effort. It will continue to evolve, both as we improve the analysis of the OneSAF behaviors and as OneSAF itself evolves. Our immediate goal is to complete the implementation of the tool prototype and to try it out with behavior programmers who have a variety of experience. We feel that the tool would be helpful to behavior developers on the OneSAF Models team in order to help maintain consistency in the models, as well as to engineers who are new to OneSAF. We would also like to try the tool with SMEs who have varying degrees of computer experience to see how effective it is in helping them encode new behaviors. Since the OneSAF behavior composer supports the specification of composite behaviors with placeholders for new primitive behaviors, an SME could potentially use a design tool to compose a behavior and create software requirements for primitive behavior modification.
There are several aspects of interactive behavior design using the CBR tool that could be improved. For example, our prototype does not allow a user to easily identify relevant primitives and composite patterns in several different behaviors and easily combine them together in one editor window. The design aid tool could be extended to support this. There is also no support in the composer for displaying which inputs and outputs of the primitives are being used, and how the data flows between primitives. We would like to extend our tool to provide this information graphically. A third extension would be to display the Java source code for a primitive to a programmer and indicate the places where it is likely to need modification, or even to propose the changes. For example, selector inputs are often tested in a switch statement and this could be identified automatically.

6. Conclusions

We have described a case-based design tool concept for building behaviors in OneSAF. OneSAF has a large and growing repository of primitive and composed behaviors, and there are challenges for a behavior programmer faced with creating a new behavior. The current set of OneSAF primitives is designed in such a way that a new behavior is likely to require some modification of the primitives, in addition to arrangement of the primitives in a composition. The case-based tool described here is intended to provide examples of working behavior that match the new behavior to a significant degree so that a programmer can quickly identify primitives to modify and
compose a working behavior that is consistent with other behaviors.

7. References


Author Biographies

DOUGLAS REECE is a senior scientist at SAIC in Orlando, Florida. His interests in AI are in intelligent agents, human behavior modeling, and computer vision. Dr. Reece has over 10 years of experience in modeling individual combatants in military simulations.

JACKIE ZHANG is senior software engineer at SAIC. She has been working as a software developer in the field of modeling and simulation for more than seven years. She has a Master’s degree in Computer Science from the University of Central Florida.

JENIFER MCCORMACK is a Chief Scientist and Program Manager at SAIC. Dr. McCormack has extensive experience in simulation training systems, instructional training systems, and rapid prototyping efforts for research and development. She specializes in information technology, case based reasoning, data mining, intelligent agents, knowledge-based systems, automated knowledge acquisition, and cognitive modeling.