Science Applications International Corporation
From Science to Solutions™

24 February 2006

Dr. Harold Hawkins
Program Officer
Office of Naval Research
Ballston Tower One
800 North Quincy Street
Arlington, VA 22217-5660

Subject: Case-Based Reasoning (CBR) Final Report

Reference: Contract No. N00014-03-C-0178

Dear Mr. Hawkins:

Science Applications International Corporation (SAIC) is pleased to submit the enclosed final report, CDRL A002, pursuant to the referenced contract. If, upon your review, you approve of the contents and agree that SAIC has met the final requirements, then your signature on the attached DD 250 is respectfully requested.

SAIC has appreciated the opportunity of supporting you on this vital program. Should you have any questions or require additional information on the enclosed subject matter, please contact me either by phone at (407) 243-3599 or by e-mail at don-ellen.ray@saic.com.

Sincerely,
SCIENCE APPLICATIONS INTERNATIONAL CORPORATION

Don-Ellen Ray
Sr. Contracts Representative

Enclosure(s)

cc: Brian Kehoe, ONR Contracting Officer
Director, Naval Research Lab
Defense Technical Information Center
Jenifer McCormack, SAIC Program Manager
Larry Campbell, SAIC Quality Assurance

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
24 FEB 2006	**N/A**	**-**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Case-Based Reasoning (CBR) Final Report	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **SAIC Science Applications International Corporation 12901 Science Drive Orlando, FL 32826-3014**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, distribution unlimited

13. SUPPLEMENTARY NOTES
The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**SAR**	**28**	

MATERIAL INSPECTION AND RECEIVING REPORT

Form Approved
OMB No. 0704-0248

PLEASE DO NOT RETURN YOUR COMPLETED FORM TO EITHER OF THESE ADDRESSES.

1. PROC.INSTRUMENT IDEN. (CONTRACT) NO. N00014-03-C0178 | (ORDER) NO.	6. INVOICE NO./DATE 2006Feb16	7. PAGE | OF 1 of 1	8. ACCEPTANCE POINT D

2. SHIPMENT NO. SAC0001	3. DATE SHIPPED 2006Feb16	4. B/L	5. DISCOUNT TERMS

9. PRIME CONTRACTOR CODE 52302	10. ADMINISTERED BY CODE| S0514A
Science Applications International Corporation 10260 Campus Point Drive San Diego, CA 92121-1578 c/o SAIC/Ellen Hatcher 12901 Science Drive Orlando, FL 32826-3014	DCMA San Diego 7675 Dagget Street, Suites 100/200 San Diego, CA 92111-2241
11. SHIPPED FROM (if other than 9) CODE| 52302 FOB: D	12. PAYMENT WILL BE MADE BY CODE| HQ0339
Science Applications International Corp. 12901 Science Drive Orlando, FL 32826-3014	DFAS-CO West Entitlement Operations P.O. Box 182381 Columbus, OH 43218-2381
13. SHIPPED TO CODE| N00014	14. MARKED FOR CODE|
Program Officer Office of Naval Research Ballston Tower One 800 North Quincy Street Arlington, Virginia 22217-5660 ATTN: Harold Hawkins, Code 342PS	

15. ITEM NO.	16. STOCK/PART NO. DESCRIPTION *(Indicate number of shipping containers - type of container - container number)*	17. QUANTITY SHIP/REC'D*	18. UNIT	19. UNIT PRICE	20. AMOUNT
CLIN Item A002	Final Report	1	ea		

21. CONTRACT QUALITY ASSURANCE

A. ORIGIN	B. DESTINATION
☐ CQA ☐ ACCEPTANCE of listed items has been made by me or under my supervision and they conform to contract, except as noted herein or in supporting documents	☒ CQA ☒ ACCEPTANCE of listed items has been made by me or under my supervision and they conform to contract, except as noted herein or in supporting documents
_____ _____ DATE SIGNATURE OF AUTH GOVT REP	_____ _____ DATE SIGNATURE OF AUTH GOVT REP
TYPED NAME AND OFFICE	TYPED NAME Harold Hawkins AND OFFICE ONR – Code 342PS Telephone: 703-696-4323

22. RECEIVER'S USE
Quantities shown in column 17 were received In apparent good condition except as noted.

_____ _____
DATE SIGNATURE OF AUTH GOVT REP

TYPED NAME AND TITLE

_____

* If quantity received by the Government is the same as quantity shipped, indicate by ( ) mark, if different, enter actual quantity received below quantity shipped and encircle.

23. CONTRACTOR USE ONLY

DD Form 250, NOV 92 *Previous edition may be used.*

Case Based Reasoning for Human Behavior Modeling

CDRL A002 for
Contract N00014-03-C-0178

February 16, 2006

Document Control Number: CBR-03000126-022406-1.0

Science Applications International Corporation

Prepared For

Office of Naval Research

Table of Contents

Table of Figures

Abstract

Over the past two years, we have investigated how the cost of behavior development for synthetic forces could be reduced through the use of Case Based Design (CBD). CBD has the potential for helping a developer quickly find useful behaviors (Reece, McCormack et al. 2004), understand how they work (Reece, McCormack et al. 2005) and reuse them. We investigated CBD for behavior in two different simulation environments. In the first year of the project, the most mature and readily available environment for military simulation was Composable Behavior Technology (CBT). The CBT environment included a behavior grammar with a graphical editor to compose behaviors (von der Lippe and Courtemanche 1999; von der Lippe, McCormack et al. 2000). In the second year of the project, another environment became available for use: the prototype version OneSAF, with its behavior composer component. OneSAF will be the main Army constructive simulation for battalion and below operations, and is being adopted by various Marine (e.g. CACCTUS) and Joint (e.g. Urban Resolve exercises) programs for simulating ground and combined arms operations (Surdu 2005). Behavior design advances made in the OneSAF environment will thus potentially have a large impact in military simulation development costs.

Our investigations of CBD for behavior design has produced several techniques for associating indices with behaviors and searching for relevant behaviors based on these indices. Some of these relevance indices are based directly on available metadata describing the behavior (e.g. echelon and unit type). Others are based on functional categories for primitive behavior that we assigned directly or inferred from the use of certain keywords in the behavior name or text description. We have also reusable structural patterns in OneSAF behaviors that can be identified automatically in the XML descriptions.

Our work has also yielded several promising approaches for helping a programmer understand existing behaviors. We have identified a number of information-gathering steps that programmers commonly take when analyzing behavior and investigated different ways to collect and present this information quickly and concisely. For example, programmers often look at the input and output variables of components they are analyzing; they determine how component inputs and outputs are connected in a complex behavior; they look at primitive source code; and they look at how components are used in other behaviors.

We have implemented some of these search and analysis techniques in a prototype tool based on the OneSAF behavior composer software. The tool has a query panel where the user can enter characteristics of the target behavior, a panel showing potentially relevant behaviors ordered by matching score, a graphical panel showing the execution plot of a selected behavior, and tabs for showing data flow in the behavior. We have prototyped some of the indices for matching behaviors and some of the input-output data flow displays that help analyze behavior. In preliminary tests, OneSAF behavior programmers have found the tool useful and have indicated that it could enable them to find reusable behaviors on the order of ten times faster than without it.

1 Problem

Military simulation exercises are typically populated by hundreds or thousands of computer-controlled entities. These entities are directed by human controllers at the unit level and perform tasks autonomously at lower levels using doctrinal behaviors encoded by programmers and military experts. The workload of the controllers can be greatly decreased if the semi-autonomous forces (SAF) can perform more complex behaviors and higher-echelon tasks; however, developing these behaviors is an expensive effort.

Modeling the behavior of humans performing military tasks is difficult and time consuming. Military tasks are often complex, and they are performed in a dynamic environment with other actors who may help or hinder the performance of the tasks. This behavior has always been difficult to for subject matter experts or software engineers to describe completely and accurately. Because of the complexity of the tasks, the dynamics of the environment, and the difficulty in specifying them, the software implementations of behaviors are difficult to verify, explain, and debug. This modeling challenge results in high costs for simulations that are intended for application across a wide range of equipment, battlefield operating systems, environments, echelons, and services.

Many military simulation exercises that use computer controlled entities use similar SAF software. This software includes Joint SAF, used by Joint forces; Close Combat Team Trainer SAF, used by U.S. Army mechanized units,

and the OneSAF Testbed Baseline (OTB), which is used in many applications. Behavior in these simulations is represented as a procedural flow of tasks—in effect, programs. The program representation is rich but often complex, as programs include conditional branches, hierarchical components, parallel threads, and synchronization constraints. This representation is derived directly from the way that military tasks are specified by military experts. In this behavior representation paradigm, creating new behavior has been expensive because software engineers were required to code the behaviors in a computer source language (e.g. C++ or Ada), because generating correct complex behavior programs is difficult, and because in order to reuse existing behaviors, the programmer has to search through voluminous code libraries and examine source code to find out what is available and what it does. This problem becomes increasingly difficult as the number of stored behaviors grows.

2 Background

2.1 Approaches to reducing modeling cost

There have been numerous efforts over the years to develop methods to reduce the cost of construction behavior models. These methods include high level languages for describing behavior (Lattimore and Riecken 1993; Crossman, Wray et al. 2004), knowledge acquisition tools (Goldman 1996; Pearson and Laird 2004), automatic learning of behaviors (Hieb, Tecuci et al. 1996; Rajput, Karr et al. 1996; van Lent and Laird 1998), and graphical behavior composition tools (von der Lippe and Courtemanche 1998; Fu, Houlette et al. 2003; Harvey and Griffith 2005; Softimage 2005). SAIC's AHBM research has explored how behavior can be constructed more efficiently by reusing existing models. Behavior reuse, especially across simulation architectures, has been a goal in the military modeling and simulation community for some time (see for example the Common Human Representation and Interchange System proposed by the Defense Modeling and Simulation Office (Bjorkman, Barry et al. 2001)). Reuse becomes more relevant as a simulation matures and accumulates a large number of behavior models. Reuse thus complements other approaches to model construction. Reuse has the potential to avoid significant costs associated with rebuilding parts of a new behavior that have already been created. These costs come from specifying, programming, debugging, and verifying the behavior.

In addition to reducing model construction cost, smart reuse can avoid a potential problem with large repositories of behavior. It may actually be difficult to use a simulation with a large behavior repository unless the many behaviors have been constructed with great care—i.e. they have been designed to cover the space of behavior with minimal overlap, their functions are clearly delineated, the behaviors were designed with a similar approach, and complex behaviors have been factored according to a good taxonomy. Without such design care, a user is faced with many similar behaviors and does not know which may perform the task of interest (correctly). Cohen et al have observed about Soar that "the usability of Soar in applied settings appears to be limited by the unstructured ontology of Soar programs" (Cohen, Ritter et al. 2005).

The benefits of reuse for avoiding modeling costs and maintaining a useful repository demand that reuse be supported for human behavior modeling even if other model construction aids are also available.

2.2 Issues for reuse

While behavior reuse, and software reuse in general, is desirable, it can be difficult to reuse behaviors because of several obstacles. First is the problem of finding existing behaviors that can be modified to make a target behavior, that can be components of the target behavior, or that can provide an example or pattern for designing the target behavior. A repository behavior could be relevant to the target behavior not only in domain function (attack, defend, supply, etc.) or unit type (including nationality, service, echelon, etc.), but in structural attributes such as how it coordinates subordinate behaviors or whether it repeats tasks for multiple objects.

A second obstacle to reuse is that relevant behaviors found in a repository must be understood so the behavior programmer knows if and how they apply to the target behavior. The understanding of behavior models is generally hampered by differences in programming styles across the repository, by the complexity of domain tasks in general, and by any details of the models that may be implemented in a low-level language. Cohen observed that "even experienced developers can have difficulty understanding Soar code written by others" (Aamodt and Plaza 1994;

Cohen, Ritter et al. 2005). We have seen that OneSAF behavior developers typically view Java code in behavior primitives to understand their function.

In addition to the obstacles described above, there is also a larger challenge for reusing behavior between different simulations. Different simulation (cognitive) architectures use fundamentally different behavior representations and behavior language semantics, so different architectures might not be capable of executing a behavior description the same way.

2.3 Case-Based Design

We have explored Case Based Design as an approach to reducing behavior modeling cost through software reuse. The CBR process typically involves four tasks (see Figure 1). First, *retrieval* finds past cases that may be appropriate in the new problem and validates them with respect to the new problem requirements. A "case" is typically an abstracted description of the problem (for example, symptoms of an error) together with a solution (for example, a repair procedure). Second, *reuse* adapts and combines the cases to fully satisfy the new problem's requirements. Third, *revision* involves testing, and possibly repairing, the new solution. Finally, *retention* updates the case base with the new case, if appropriate. CBR systems often have four elements: a case base (which stores previous cases for retrieval), a case retrieval engine (which allows users to express retrieval queries), an adaptation module (which encapsulates problem knowledge to partition or combine retrieved cases), and a validation module (which is used to help evaluate and ensure the validity of combining previous problems when solving a new problem). Example CBR systems include KRITIK (Goel and Chandrasekaran 1989), HICAP (Munoz-Avila, McFarlane et al. 1999), SIROCCO (McLaren 2003), and FABEL (Gebhardt, Vob et al. 1997).

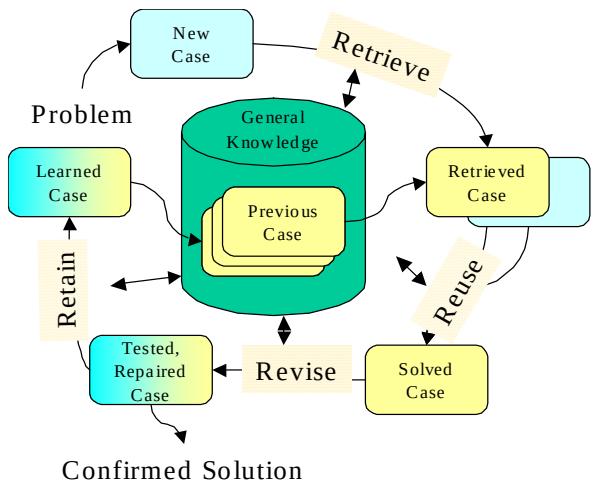

Figure 1. The Case Based Reasoning cycle (Aamodt and Plaza 1994)

Behavior authoring can be viewed as a case-based design (CBD) task (Summers, Lacroix et al. 2002; Summers and Shah 2003); the objective is to synthesize a new behavior by composing existing behavior elements. While the normal behavior design task is generative, i.e. requires the designer to create a behavior from scratch, the CBD approach will support variant design—modification of existing solutions. Our goal is to develop a CBR-based authoring tool that will provide for retrieval, reuse, and retention of behavior cases. It will be semi-automated in that it will allow behavior modelers to interactively describe a desired behavior, view relevant existing behaviors, and select one or more behaviors for reuse. The tool will suggest the relevant behaviors and suggest adaptations to satisfy the requirements of the desired behavior. The modeler will indicate when a new behavior has been evaluated and validated, and the tool will add it to the case base.

2.4 Behavior Design Environments

In order to effectively apply CBD principles to behavior design, we sought to find an environment in which behavior was expressed in abstractions that were related to the domain (military tasks) rather than in a low level computer language. In the first year of the project, the most mature and readily available environment for military simulation was Composable Behavior Technology (CBT). The CBT project, sponsored by the Simulation, Training and Instrumentation Command (STRICOM), explored the concept of primitive and composite behaviors and developed a behavior grammar with a graphical editor to compose behaviors (von der Lippe and Courtemanche 1999; von der Lippe, McCormack et al. 2000). In the second year of the project, another environment became available for use: the prototype version OneSAF, with its behavior composer component. The project switched to OneSAF in the second year because unlike CBT, OneSAF is in the beginning of its operational lifetime and is expected to be used by a large simulation community across all services in the coming years. Behavior design advances made in the OneSAF environment will thus potentially have a large impact in military simulation development costs.

2.4.1 The Composable Behavior Technology Environment

There are several behavior elements that a CBT programmer uses to compose a behavior. The main elements are primitive behaviors, predicate functions, communication primitives, and other composite behaviors. Primitive functions are implemented in low level software code in the underlying simulation. In fact, CBT runs as a separate application from a supporting SAF application, and the primitives are actually software components of the SAF (not CBT). Predicate functions test conditions in the simulation and provide for conditional branching in the composed behaviors. As with primitives, the actual test is implemented in low level code in the underlying simulation. Communication elements are actions that send specific messages between and within command echelons that allow synchronization of tasks between entities or units. Finally, composite behaviors may also include other composite behaviors; CBT thus supports abstraction hierarchies in behavior.

The behavior elements described above are all associated with particular *categories* of simulation entities. A category in CBT includes a domain, force, and echelon. There is no formal definition of domain, but it is used to distinguish aircraft, helicopters, tanks, infantry fighting vehicles, infantry, etc. Force refers to a nationality or other group that would have a unique doctrine and behavior set.

There are two other important elements that are used to compose behavior in CBT. The first is a set of roles that subordinates play in a unit's behavior. For example, if a programmer is defining a unit attack behavior, he/she might include actions that subordinates would play in roles of maneuver, fire support, and reserve. The subordinate roles used in the behavior and the (lower level) behaviors are part of the unit behavior.

The remaining element is the connector, which defines the order of tasks. Connectors can have temporal constraints associated with them that determine when the behavior actually transitions from one element to the next. The constraints can be based on a time delay, the completion of a behavior in another role, or on a communication. Thus connectors can synchronize the actions of entities in a unit.

The CBT system has a graphical editor that allows a user to select behavior elements and place them in a composition window. Typically the user selects and places composites, primitives, predicates, and communication actions and then orders them with connectors. The editor only provides access to elements that are valid for the category of entity for which the behavior is being created. The editor presents parameter lists for each element selected and allows the user to fill in values or specify that values will be provided at run time. Figure 2 shows this editor.

2.4.2 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

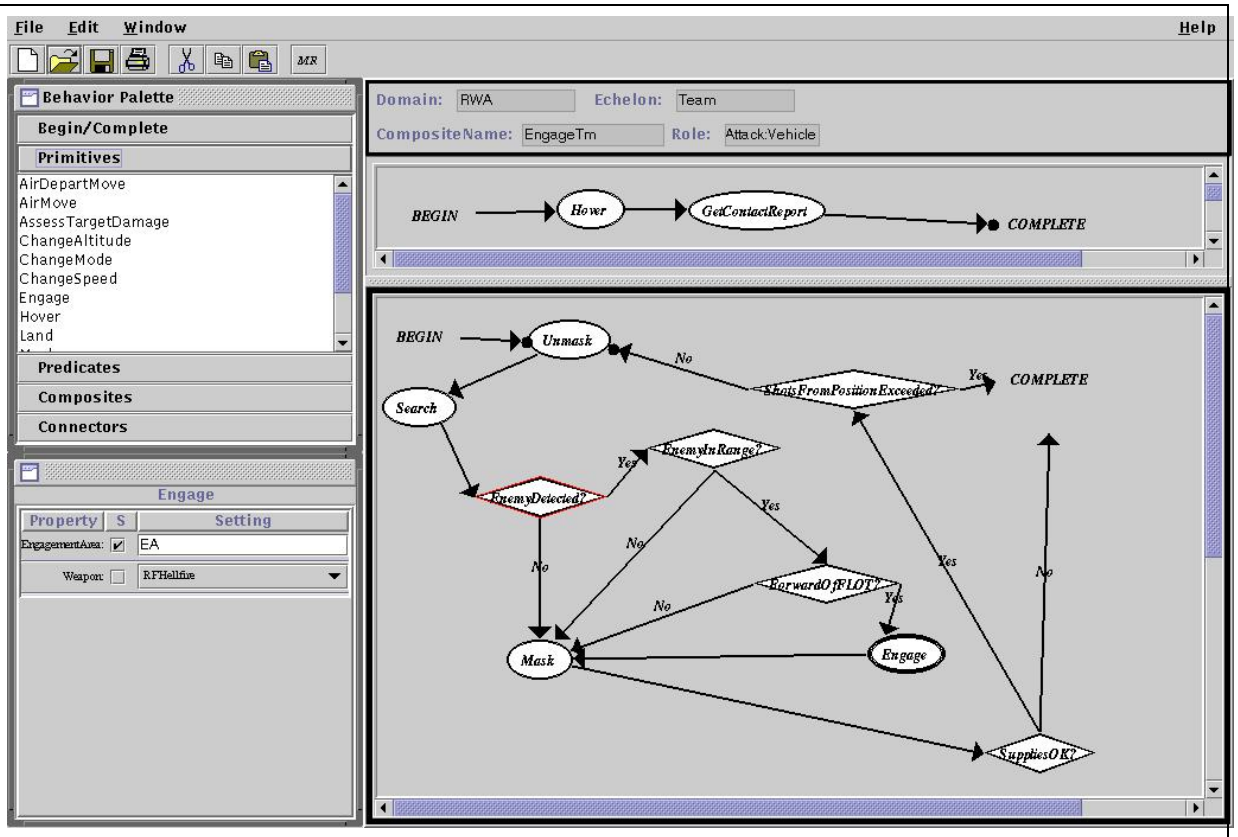

Figure 2. CBT behavior editor graphical interface. Menus on the left allow selection of behavior elements which are used to compose a behavior in the windows on the right. Oval shapes represent primitive behaviors, while diamonds represent predicates.

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 was released in April 2005.

2.4.2.1 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.

2.4.2.2 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.

2.4.2.3 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.

2.4.2.4 The OneSAF Behavior Repository

The previous section described the inherent difficulty in reusing behavior using only an abstract description. 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 Approach

3.1 Year 1: Composable Behavior Technology

In the first year of the project, the most mature and readily available environment for military simulation was Composable Behavior Technology (CBT). The CBT project, sponsored by the Simulation, Training and Instrumentation Command (STRICOM), explored the concept of primitive and composite behaviors and developed a behavior grammar with a graphical editor to compose behaviors.

3.1.1 The CBR-CBT Tool

For CBT we developed a new graphical tool that works in conjunction with the CBT behavior editor. The CBR-CBT tool displayed a panel which presented the user with queries about the desired new behavior and allowed the user to enter responses. Another panel in the tool displayed the names of all of the composite behaviors in the case repository and a matching score for each. The cases were listed in order of decreasing score. As the user entered a response to each query, the matching scores are updated and the behavior names were redisplayed in the correct order. At any time, the user could select one of the behavior names and that behavior would be displayed graphically in third panel.

Figure 3 shows the CBR-CBT tool after a behavior has been selected.

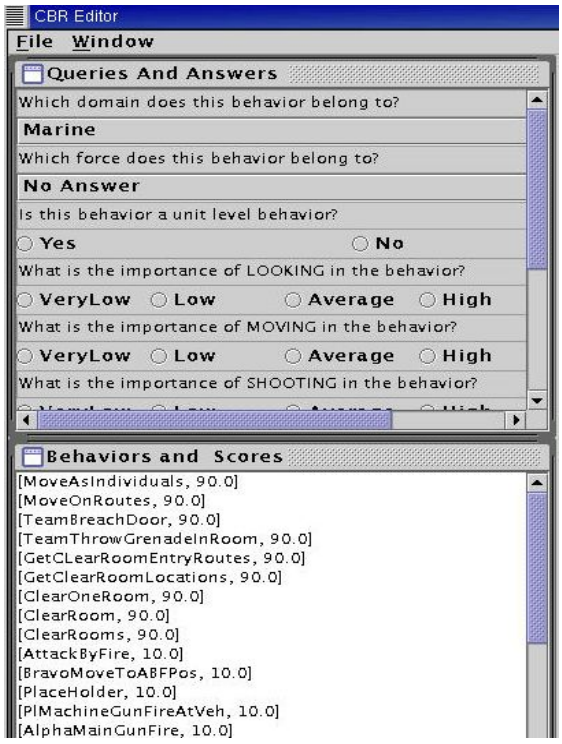

Figure 3. CBR-CBT query panel

3.1.2 Case Representation

In CBT, composite behaviors were stored in a relational database along with definitions of behavior elements, lists of attribute names, relations between entity categories and behaviors, etc. As mentioned above, this representation of behavior as data objects instead of embedded software code is already a great step forward towards reusing behavior in CGF. For CBR, it meant that part of the infrastructure for a case repository was already available.

Figure 4 shows the architecture of the combined CBR-CBT system. The CBT repository provided the infrastructure for the "solution" part of the case. The "problem" part of the case was based on specific behavior features, described below, whose values were derived from the behavior description in the database. The values for the behavior features were computed off-line and saved in an XML file along with a pointer to the actual behavior in the repository. Thus at run time the case base was read from a file and the matching process was reasonably fast. Only when a user wished to see the graphical representation of a behavior was the actual behavior retrieved from the relational database.

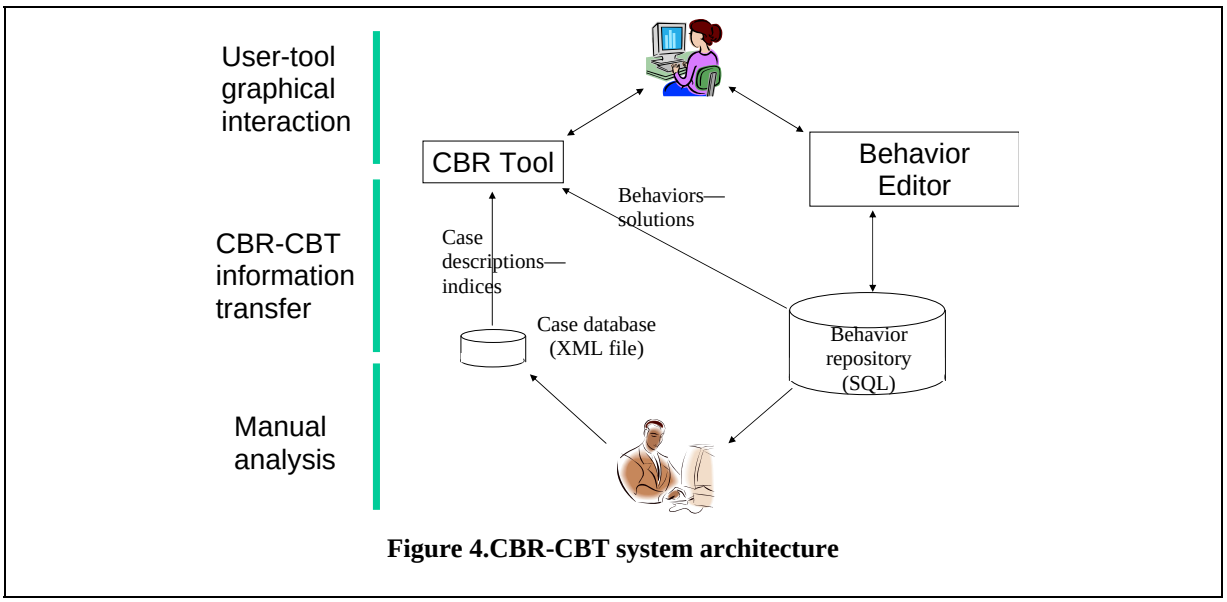

Figure 4.CBR-CBT system architecture

The problem part of the case was presented to the user as a set of queries about the desired new behavior. The user answered the queries to describe the problem. The responses were multiple-choice selections which were available as pull-down menu selections or radio buttons (see Figure 3). The user did not have to answer all queries; matches were computed based on the questions that were answered.

3.1.3 Case Retrieval

Since the user answered queries by selecting from discrete choices, the CBR-CBT tool had an easy time parsing the response and could directly look up feature values for the new behavior. These feature values are called case *indices*. The corresponding indices were then retrieved from the case base (from the XML file, but in memory at run time) for each behavior and a similarity score was computed. This score was normalized to a value between 0 and 100 for each index. The normalized scores were multiplied by a weight and then summed together to form an overall matching score for the behavior. In other words,

$$Score_B = \sum_i \omega_i ns_i$$

$$ns_i = N_i(match_i(I_i, I_{Bi}))$$

where
B is the behavior,
i is the index number,
ω_i is the weight for index i,
ns_i is the normalized score for index i,
$match_i$ is the matching function for index i,
N_i is the normalization function for $match_i$,
I_i is the index i value for the desired behavior, and
I_{Bi} is the index i value for behavior B.
Note that I may have a non-numerical value such as a domain name.

We have created five indices. Three were based on entity category—domain, force, and echelon; another was based on behavior function, and the fifth was determined by the need for subordinate coordination in the behavior.

3.1.3.1 Category Indices

Unit category was a natural basis for finding similar behaviors; in fact, in the CBT behavior editor, the user began the behavior creation process by specifying a category, and then [only] elements in that category were made available for selection. However, there were cases in which two categories were very similar, or in which some aspect of a behavior in one category were similar to a behavior in another category. For example, U.S. Army infantry is different in organization, equipment, and tactics from U.S. Marines, but nevertheless there are strong commonalities. Infantry and tanks are quite different domains, but both domains make use of bounding overwatch movement techniques requiring alternating movement and fire support by subordinates.

To allow behaviors to be used across domains, we used the category to measure similarity rather than as a pruning feature. For domain and force, we created a similarity table that gave a similarity score for each <case domain, new behavior domain> pair. The match function for these indices was a table lookup. Echelon was more difficult to deal with since echelon names vary between domains and forces, and because organizations are different at different echelon levels. However, there was clearly a difference between unit behaviors that control subordinates and individual-level behaviors that have no subordinates; therefore we used a predicate “is-unit” to provide some indication of similarity.

3.1.3.2 Function Index

The category-based indices characterized the military organization taxonomy. For the next index, we attempted to capture the fundamental function of the behavior. Function was classified according to the traditional military functional taxonomy: look, move, shoot, and communicate. To use function as an index we first classified behaviors in the case repository according to function, and then elicited the function of the desired behavior from the user, and finally computed a match between them.

For the function index we did not select a single military function for each behavior. Rather, each behavior had a fuzzy membership in each military function. For example a reconnaissance behavior might have had a high membership in move and look, a moderate membership in communicate, and a low membership in shoot. To compute these membership values, we first assigned (manually) membership values for each primitive CBT behavior. Next, raw function scores (not fuzzy membership values) for composite behaviors were computed by summing the membership values of the primitives and other composites that comprise the behavior. Finally, the ratios of the raw scores for each function to the sum of all four function scores were used to estimate the membership value for that function. For example, a behavior whose function score was 50% of the sum of scores was assigned a “very high” membership value. This computation is summarized below:

Let $M_{P,f}$ be the membership of primitive behavior P in military function f; $S_{B,f}$ be the raw score of composite behavior B for function f; then

$$S_{B,f} = \sum_{\text{Primitives } P \text{ in } B} M_{P,f} + \sum_{\text{Composites } C \text{ in } B} S_{C,f}$$

$$M_{B,f} \approx \frac{S_{B,f}}{\sum_{All\ functions\ f} S_{B,f}}$$

This ratio-based algorithm was an effort to normalize the function values across composite behaviors with different numbers of behavior elements and at different echelons. Other approaches are also possible, such as an average membership value based on the number of elements.

The function membership value for the desired behavior was obtained by querying the user directly about how strongly each function class characterizes the desired behavior. The user was not given any specific guidance about how to characterize different types of behavior; the characterization was based only on the user's intuition.

The behavior case function index was matched against the desired behavior function index by using a square-error function. For each function, the error was the difference between the case membership value and the desired membership value (each a number between 0.0 and 1.0). The raw match score was the sum of the squares of these errors for the four functions. This computation is summarized below:

$$match = \sum_{f} \left[1.0 - \left(M_{B,f} - M_f\right)^2\right]$$

where M_f is the membership of the desired behavior in military function f.

3.1.3.3 Subordinate Coordination Index

The final index was intended to capture some of the structure of the behavior rather than application or functional class. Some unit behaviors specified actions that subordinates must take, and make use of specific communication primitives to synchronize the subordinate actions. The subordinate coordination index was a predicate that indicated whether the behavior made use of this specific communications primitive.

3.2 Years 2-3: OneSAF

In the second year of the project, another environment became available for use: the prototype version OneSAF, with its behavior composer component. The project switched to OneSAF in the second year because unlike CBT, OneSAF is in the beginning of its operational lifetime and is expected to be used by a large simulation community across all services in the coming years. Behavior design advances made in the OneSAF environment will thus potentially have a large impact in military simulation development costs.

3.2.1 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.

3.2.2 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 (see Figure 5): 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

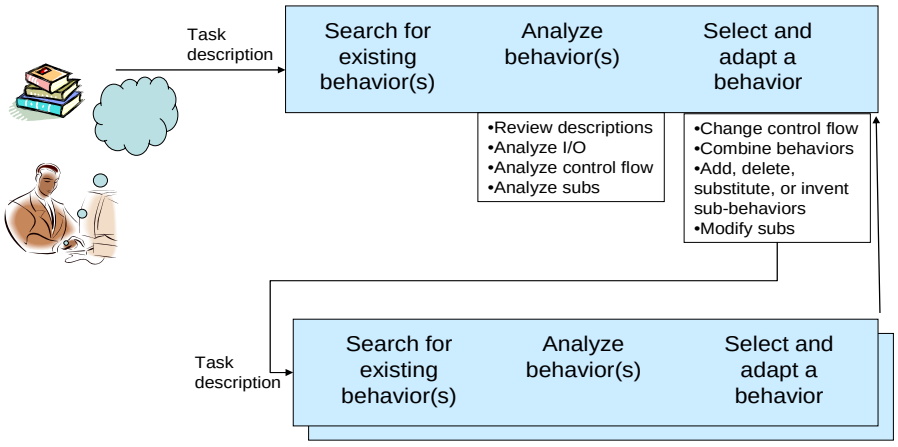

Figure 5. Behavior design process with design aid.

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.

3.2.3 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., bounding 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.

3.2.4 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 the OneSAF Block C release, 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—i.e., 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 the composite behavior 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.

3.2.5 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 until the OneSAF Block D release later in 2006. 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 metadata 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.

3.2.6 Tool Prototype

We have created an extension to the OneSAF behavior composer that allows us to prototype design aid functions. Figure 6 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 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. This figure shows how the user can perform the first steps of the design process (Figure 5), finding existing behaviors.

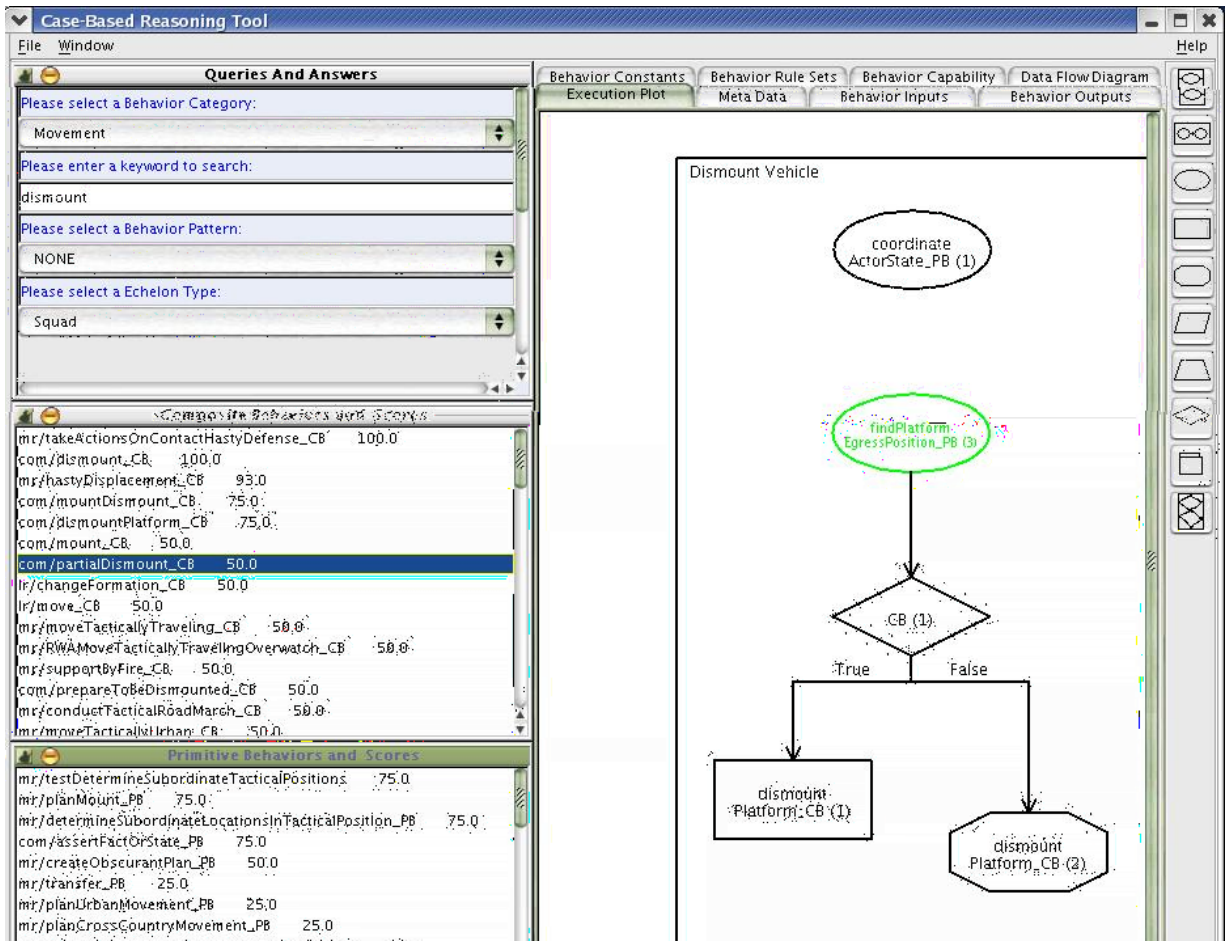

Figure 6. CBR tool in OneSAF

The next step of the design process is to analyze candidate behaviors. Figure 6 shows a primitive behavior in the behavior flow graph with highlighting; this indicates that the primitive has symbolic, enumerated-value inputs,

which may indicate that the primitive's function is variable and depend on a data input.

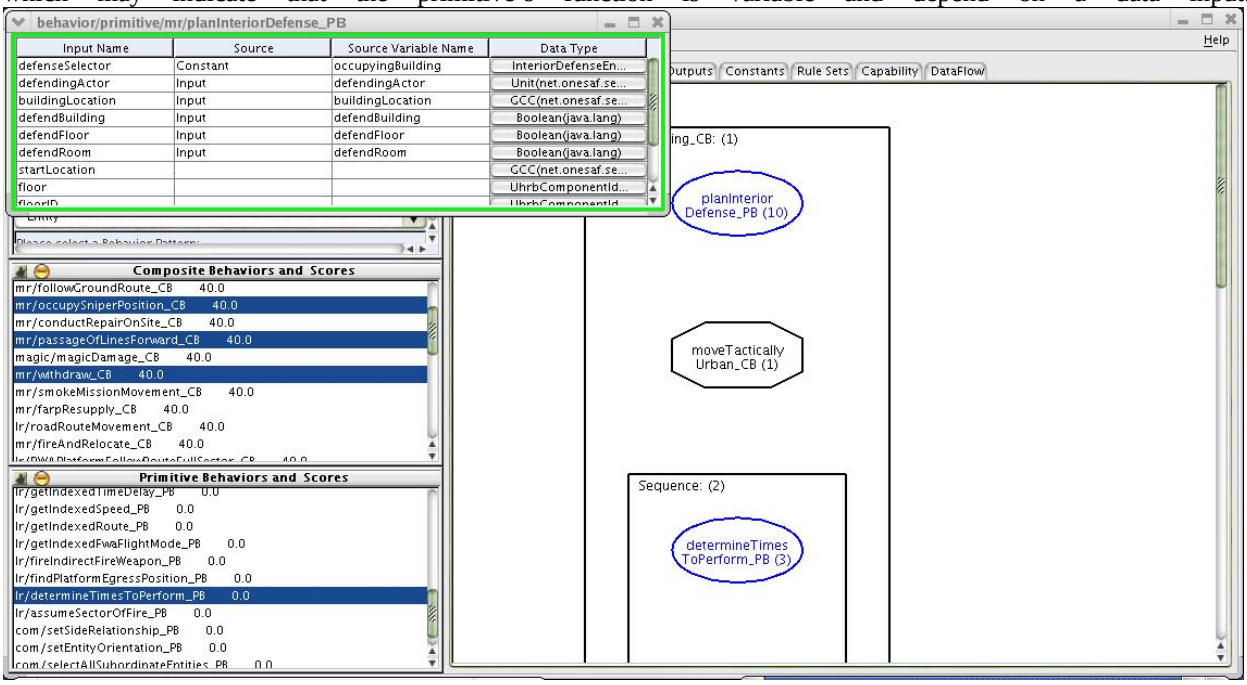

Figure 7 shows a component behavior that has been selected and window for that behavior showing all of the input variables for this component behavior. The table indicates the type of the input and the source of the data. If the type is Enumeration and the input is a behavior constant, then this input *selects* a particular primitive function appropriate for the behavior. If the input comes from another component, the user can see how the two components are interdependent. The names of the inputs may also provide clues to how the component works.

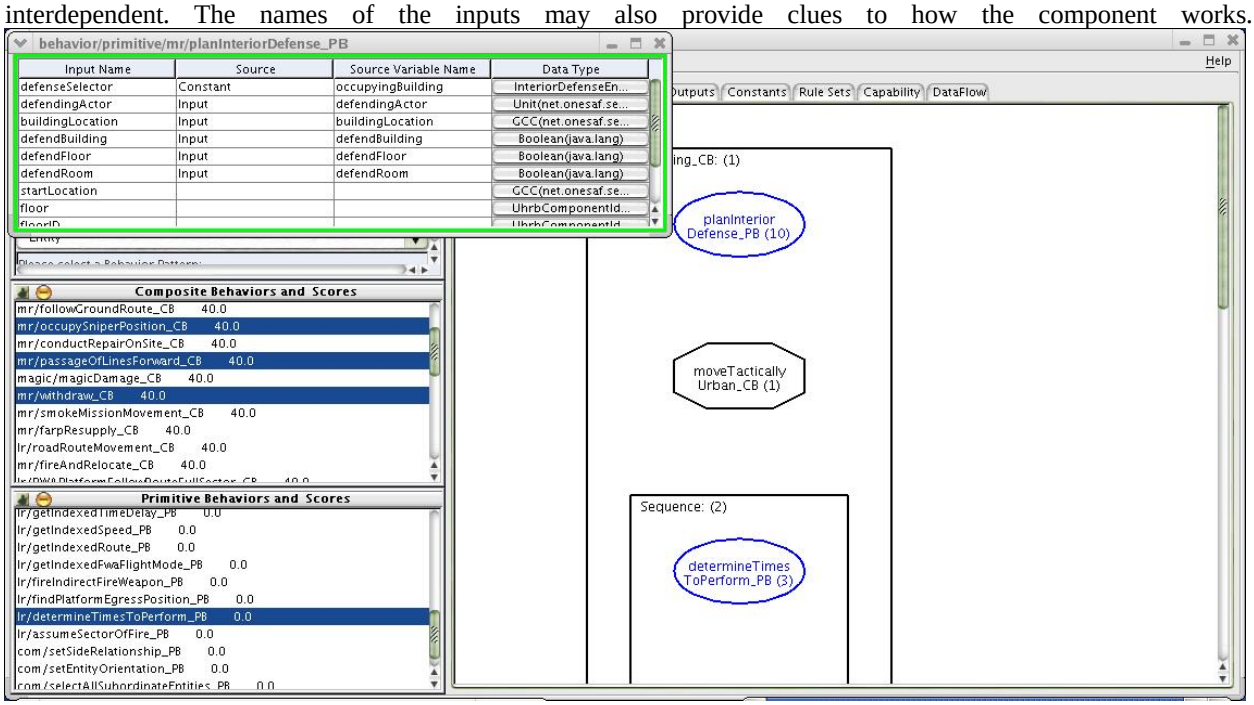

Figure 7 also shows that the component name can be selected in the lower-left scoring panel, and the tool will highlight other components that use the component.

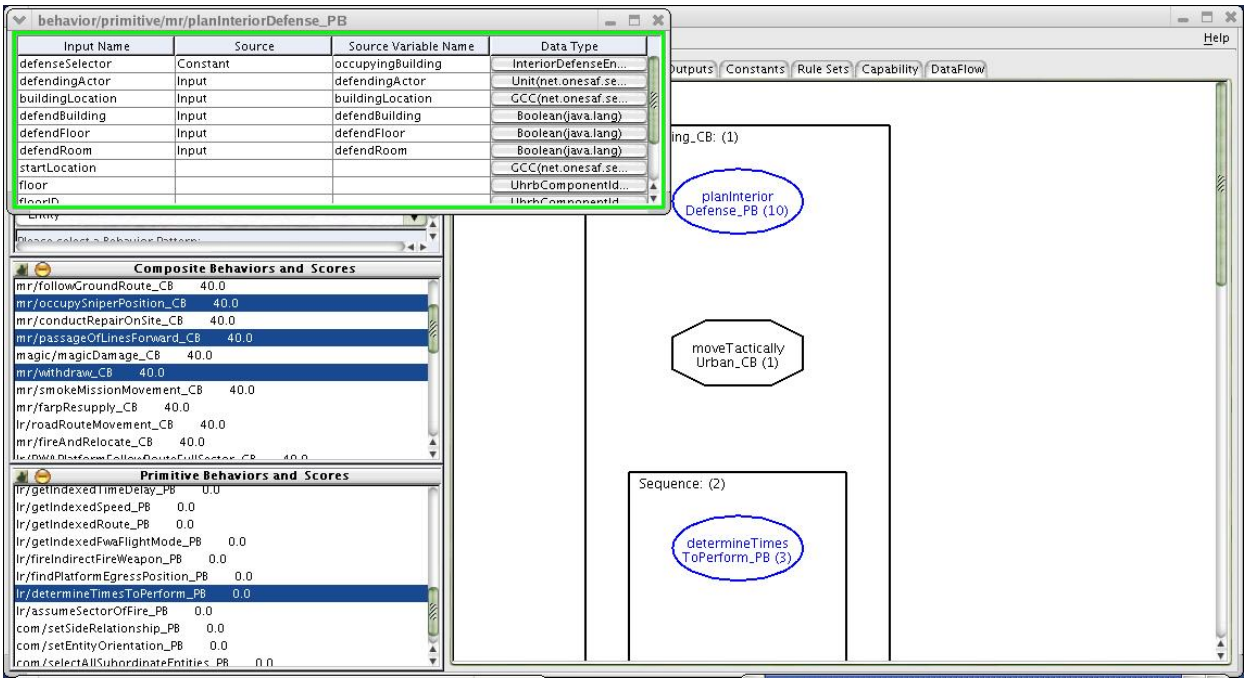

Figure 7. CBR tool display of input data characteristics for a behavior

This kind of input window is designed for all different types of behavior components, such as primitives (as shown above), composites, Order Senders and Conditional Branches (see below). The design was based on the characteristics of each type of behavior component. For example, an Order Sender input window shows the subordinate(s) that will perform the orderable task; a Conditional Branch input window shows the predicate(s) that are used within the conditional and their inputs. (Figure 8)

In addition, to help understand the inputs/output data flow within the behavior, CBR tool also have a DataFlow diagram that helps to analyze the behavior input data in a quick and convenient way. This diagram shows where all the behavior inputs and constants are going within the behaviors amongst all behavior components. This diagram is especially useful when enumerated input/constant values are used by various behavior components because different values can make the same behavior do different tasks. (Figure 9)

4 Results

We have conducted an experiment with several test subjects. These test subjects each had different level of programming experience and familiarity with the OneSAF behavior building process. The first one was a SME/SAF operator, with no programming experience at all. The second one was a programmer with lots of OTB behavior programming experience but very limited knowledge of the OneSAF behavior building process. The third subject was an experienced OneSAF behavior programmer. We gave these three people a task to design two new behaviors from informal task descriptions–"USMC version of enter and clear building" and "OPFOR irregular shoot and scoot."

These two tasks are chosen for their complexity and familiarity. We wanted to see how our test subjects make the best use of our tool to simplify their behavior modeling process. First, we hoped to see that the subjects can use the CBR's searching capability to find similar and familiar behaviors within the repository and then use the various input tables and data flow tables to understand the existing behaviors first before they even tried to start building the new behaviors. Secondly, we hoped to see that the structural complexity of the new behaviors can be simplified by using CBR's behavior structural pattern matching facility. We understood that with different programming experiences and background, these test subjects would be able to use the CBR tool from just simple searching to its

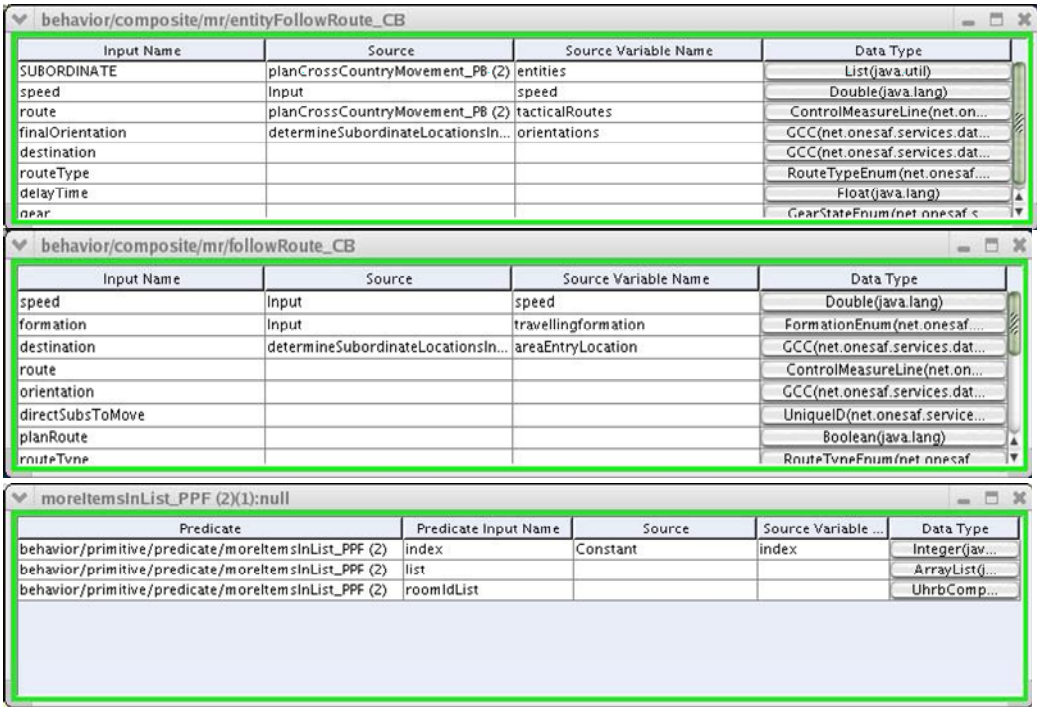

Figure 8. CBR tool display of input tables of various behavior components. From top down: OrderSender component, Composite component, Conditional Branch component

fullest extend, therefore the experiment would collect more accurate data on how much we can make the behavior modeling process easier and faster.

During the experiment, we found out that our CBR tool's searching and browsing features were very effective to help all of them quickly find existing behaviors that had similar types of actions. All test subjects frequently examined behavior inputs using the data tables. Structural behavior matching was less commonly used, but our most experienced OneSAF programmer found such examples to help him building the new behaviors. One programmer said "I could do in 15 minutes what it would otherwise take all day to do", which was almost 32 times faster than a normal behavior building process without using our tool.

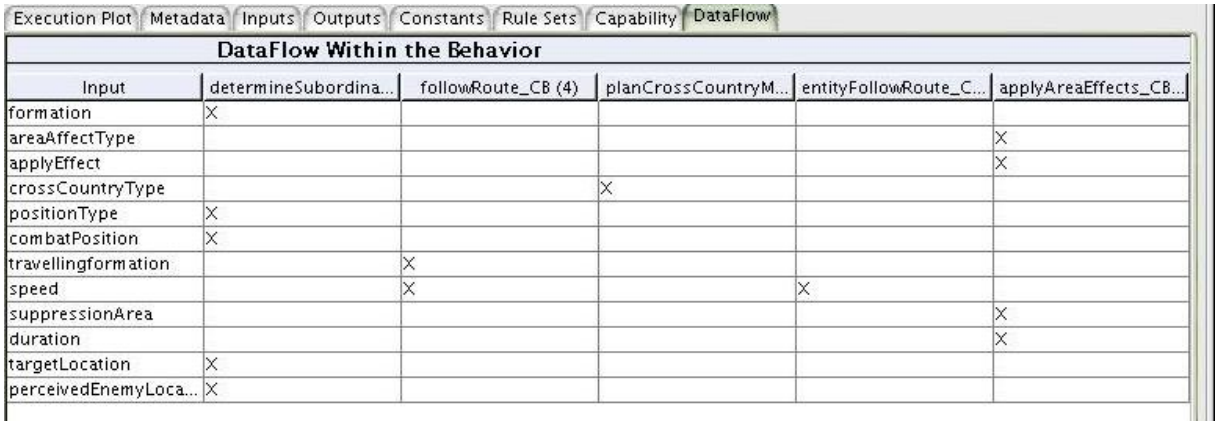

Figure 9. CBR tool display of behavior internal data flow

We also realized that although the CBR design process is promising, there should be some extra training in using the OneSAF Behavior Composer and the CBR tool in order to make the behavior modeler more efficient.

5 Conclusions

5.1 Future Work

There are many areas of research that could enhance the capabilities of the CBD tool in both retrieval and analysis tasks. The goal is to make these tasks more robust in the face of inadequate metadata, and automatically generate some of the new behavior.

5.1.1 Case Retrieval

Develop new domain and structural indices. Useful indices will be identified both through further analysis of existing OneSAF behaviors, and feedback from users in evaluation tests. Both types of indices will be implemented in our CBD tool in order to perform these tests. For domain indices, this will often require manipulation of behavior metadata, while for structural indices, which indicate how components of a behavior are related to one another, will require analysis of parts of the XML behavior representation. The relations between components reflect whether the behavior is performing some action repeatedly, for example, if it is coordination actions of subordinates, or if it is performing two tasks simultaneously, etc. We believe that the most effective design process may be to identify the target structure first, and then separately find behavior components that match the domain characteristics of the target behavior.

5.1.2 Case Analysis

Extend the information displays to other component types in the OneSAF behavior design language. The information displayed will include the following:

- Conditional branch and conditional loop: the names of predicate functions used in the conditional expression; and the input variables for the predicate functions and their sources.

- Order sender: the input variables of the behavior being sent to subordinates, their types and sources; and the name of the list of subordinates and the source of the list

We will implement these extensions and evaluate them in experiments with behavior modelers. Feedback from these evaluations will also be used to identify additional analyses displays.

5.1.3 Inherent Behavior Attribute Analysis

Analyze the infrastructure interface used by OneSAF behavior primitives and determine if and how the function attributes of the primitives can be assigned based on their use of the infrastructure. The function types of the primitives will be estimated by examining the Java code for the execution method of the primitives and looking for key method calls into the simulation infrastructure. Essentially, these method calls will be treated as even lower-level primitives that are more closely grounded in the world. For example, the simulation infrastructure provides information about or modifies the environment—the terrain skin, nearby features such as trees and water, and features of buildings. Other methods are used to get information about the situation, change the physical state of the entity, send messages to other entities, etc.

5.1.4 Behavior Adaptation

Design an extension to the CBD prototype to identify a skeleton behavior, to analyze the components of the skeleton in light of the domain characteristics of the target behavior, and to identify inappropriate components in the skeleton, and to suggest appropriate replacements.

5.1.5 Low Level Code Generation

Analyze the Java code changes needed to extend primitives and identify which of these changes are highly regular and which are unique to each function. We will then create a CBD tool capability to generate the highly regular portions of the code automatically.

5.1.6 CBD for Cognitive Architectures

Investigate the application of the tools and techniques for behavior reuse developed for OneSAF to cognitive architectures. We will identify a promising candidate architecture based on the use of a high-level language to describe behavior, the representation of a behavior as data in a form such as XML, and the availability of a large number of behavior models for the cognitive architecture in a repository. We will then develop a detailed concept for how the CBD technology can be applied to the selected architecture.

5.2 Summary

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.

6 References

Aamodt, A. and E. Plaza (1994). "Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches." AI Communications **7**(1): 33-59.

Bjorkman, E. A., P. Barry, et al. (2001). Results of the Common Human Behavior Representation And Interchange System (CHRIS) Workshop. Fall 2001 Simulation Interoperability Workshop, Orlando, Florida, Simulation Interoperability Standards Organization.

Calder, R., J. E. Smith, et al. (1993). ModSAF Behavior Simulation and Control. Third Conference on Computer Generated Forces and Behavioral Representation, Orlando, Florida, University of Central Florida.

Cohen, M. A., F. E. Ritter, et al. (2005). Herbal: A High-Level Language and Development Environment for Developing Cognitive Models in Soar. Behavior Representation In Modeling and Simulation, Universal City, Ca., Simulation Interoperability Standards Organization.

Courtemanche, A. J. and R. L. J. Wittman (2002). OneSAF: A Product Line Approach for a Next-Generation CGF. Eleventh Conference on Computer Generated Forces and Behavioral Representation, Orlando, Florida, Simulation Interoperability Standards Organization.

Crossman, J., R. Wray, et al. (2004). A High Level Symbolic Representation for Behavior Modeling. Behavior Representation In Modeling and Simulation, Alexandria, VA, Simulation Interoperability Standards Organization.

Fu, D., R. Houlette, et al. (2003). A Visual Environment for Rapid Behavior Definition. 2003 Conference on Behavior Representation in Modeling and Simulation (BRIMS), Scottsdale, AZ, IEEE Computer Society Press.

Gebhardt, F., A. Vob, et al. (1997). Reasoning With Complex Cases. Norwell, MA, Kluwer Academic Publishers.

Goel, A. and B. Chandrasekaran (1989). Functional Representation of Designs and Redesign Problem Solving. 11th International Joint Conference on Artificial Intelligence, Detroit, Michigan, Morgan Kaufman.

Goldman, S. R. (1996). Knowledge Acquisition and Delivery: Constructing Intelligent Software Command Entities. Sixth Conference on Computer Generated Forces and Behavioral Representation, Orlando, Florida, University of Central Florida.

Harvey, E. and T. Griffith (2005). Affordable KA/E for Human Performance Modeling. Affordable Human Behavior Modeling Contractor's Meeting, Office of Naval Research.

Henderson, C. and A. Rodriguez (2002). Modeling in OneSAF. Eleventh Conference on Computer Generated Forces and Behavioral Representation, Orlando, Florida, Simulation Interoperability Standards Organization.

Hieb, M. R., G. Tecuci, et al. (1996). Training a ModSAF Command Agent Through Demonstration. Sixth Conference on Computer Generated Forces and Behavioral Representation, Orlando, Florida, University of Central Florida.

Karr, C. R. (2003). Writing Behaviors. **2005**.

Karr, C. R. (2004). Primitive Behavior Analysis. **2005**.

Lattimore, P. J. and M. E. Riecken (1993). A Virtual Battlespace Language. Third Conference on Computer Generated Forces and Behavioral Representation, Orlando, Florida, University of Central Florida.

Maher, M. L. and A. Gomez de Silva Garza (1996). "Developing Case-Based Reasoning for Structural Design." IEEE Expert **11**(3).

McLaren, B. M. (2003). "Extensionally Defining Principles And Cases in Ethics: An AI Model." Artificial Intelligence **150**: 145-181.

Munoz-Avila, H., D. McFarlane, et al. (1999). Using Guidelines to Constrain Interactive Case-Based HTN Planning. 3rd International Conference on CBR, Seon, Germany, Springer.

Pearson, D. and J. E. Laird (2004). Redux: Example-Driven Diagrammatic Tools for Rapid Knowledge Acquisition. 2004 Behavior Representation in Modeling and Simulation, Alexandria, VA, Simulation Interoperability Standards Organization.

Rajput, S., C. R. Karr, et al. (1996). Learning the Selection of Reactive Behaviors. Sixth Conference on Computer Generated Forces and Behavioral Representation, Orlando, Florida, University of Central Florida.

Reece, D. A., J. McCormack, et al. (2004). A Case-Based Reasoning Tool for Composing Behaviors for Computer Generated Forces. 2004 Behavior Representation in Modeling and Simulation, Alexandria, VA, Simulation Interoperability Standards Organization.

Reece, D. A., J. McCormack, et al. (2005). A Case-Based Behavior Design Aid for OneSAF. Behavior Representation In Modeling and Simulation, Universal City, Ca., Simulation Interoperability Standards Organization.

Softimage (2005). Softimage|Behavior. **2005**.

Summers, J., Z. Lacroix, et al. (2002). Case-Based Design Facilitated by the Design Exemplar. 7th International Conference on Artificial Intelligence in Design, Cambridge, U.K.

Summers, J. and J. Shah (2003). Developing Measures of Complexity for Engineering Design. ASME DETC-2003, Chicago, IL.

Surdu, L. J. R. (2005). One Semi Automated Forces (OneSAF) Program Overview. Urban Operations Workshop. Portsmouth, Va.

van Lent, M. and J. E. Laird (1998). Learning by Observation in a Tactical Air Combat Domain. Seventh Conference on Computer Generated Forces and Behavioral Representation, Orlando, Florida, University of Central Florida.

von der Lippe, S. and A. J. Courtemanche (1998). Interim Results in the Development of User Composable Behaviors. Seventh Conference on Computer Generated Forces and Behavioral Representation, Orlando, Florida, University of Central Florida.

von der Lippe, S. and A. J. Courtemanche (1999). Advances in the Execution Engine for Composable Behaviors. Eighth Conference on Computer Generated Forces and Behavioral Representation, Orlando, Florida, Simulation Interoperability Standards Organization.

von der Lippe, S., J. McCormack, et al. (2000). Embracing Temporal Relations and Command and Control in Composable Behavior Technologies. Ninth Conference on Computer Generated Forces and Behavioral Representation, Orlando, Florida, University of Central Florida.