

Static Analysis and Validation of Composite Behaviors in Composable Behavior Technology

Jackie Zheqing Zhang
Bill Hopkinson, Ph.D.
12479 Research Parkway
Orlando, FL 32826-3248
407-207-0976

jackie.z.zhang@saic.com, william.c.hopkinson@saic.com

Sheau-Dong Lang
School of Electrical Engineering & Computer Science
University of Central Florida
Orlando, FL 32816
407-823-2474
lang@cs.ucf.edu

ABSTRACT: *Compute Generated Forces (CGF) have long been used in combination with human players as semi-automated forces (SAF) in the DIS and HLA contexts to represent opposing or friendly forces. The Composable Behavior Technology (CBT) was developed to provide the capability of representing complex behaviors in terms of primitive behaviors, allowing new behaviors to be easily defined to meet simulation objectives in the SAF systems. The resulting compound behaviors are stored in XML format in the Behavior Repository for future use in defining more complex composite behaviors and missions. A drawback of the current CBT implementation is that non-doctrinal behaviors can be created using the Behavior Editor, since the CBT does not provide behavior validation. Behaviors developed by the end users using the Behavior Editor may be illogical, incomplete, contradictory, and possibly leading to deadlocked situations. The goal of our research was to implement a tool that performs static analysis and validation of the behaviors defined using the CBT for ModSAF. Using advanced XML and JAVA technologies, together with graph algorithms, we developed a tool called LogicChecker to provide static behavior validation for behaviors created using the CBT methodology. Our static validation techniques perform assessment on the basis of the characteristics of the static model design and source code, prior to machine execution.*

1. Background

To overcome the difficulties of building complex simulation behaviors in ModSAF and CCTT SAF, CBT was developed to provide the capability of representing complex behaviors in terms of primitive behaviors, allowing new behaviors to be easily defined to meet simulation objectives in the SAF systems [3]. As a prototype, CBT allows an end user to successfully combine various primitive behaviors and decision points (predicates) to create tactically realistic composite behaviors for relatively complex missions, as well as to create

reactive behaviors and command and control behaviors for simulated entities and units at various organizational levels. The development process combines an analysis of the behaviors implemented in the current ModSAF and CCTT SAF systems, and a careful study of the existing CCTT SAF Combat Instruction Sets assisted by the subject matter experts, ensuring that the behaviors are realistically developed, tested, and validated through simulation.

For readability and portability purposes, as well as to formally represent the behaviors, the CBT uses the Extensible Markup Language (XML) to

describe the properties and actions of the composite behaviors. The XML representation of the behaviors is an important part of the CBT by empowering the end users with the flexibility in creating behaviors.

A significant drawback of the current CBT implementation is that invalid behaviors can be created using the Behavior Editor, since the CBT does not provide behavior validation. Behaviors developed by the end users using the Behavior Editor may be illogical, incomplete, contradictory, and possibly leading to deadlocked situations. Some of these commonly made errors such as unreachable actions and erroneous data usage can easily be detected before behavior executions, leading to significant savings in time and effort of the model development process. At the same time, a formal validation of the models would increase their credibility. Our research is focused on developing a static analysis and validation tool for user-composed behaviors.

This paper will review related works in behavior modeling and validation, discuss an approach to statically validate CGF behaviors, detail the implementation of the approach and provide insights and conclusions on the on-going research and development.

2. Related Work

2.1 Behavior Representation and Modeling

The primary goal of behavior representation in CGF is to display realistic behaviors in the complex simulation environments. A significant amount of resources are available that describe how to create complex behaviors using various modeling techniques such as the finite-state machine, rule-based representation as in CCTT SAF and ModSAF, as well as advanced AI techniques. According to [4], a recent report by the National Research Council (NRC) stated that neural networks hold promise of providing a powerful learning model. Researchers have experimented with using neural networks in modeling battlefield behaviors for CGF systems, and reported the development of a neural network-based movement model illustrating improved performance through the use of a modular context paradigm.

Another important behavior representation method is using Intelligent Agents to model the CGF entity behaviors through rule learning in an interactive simulation environment [6]. With an agent, new tactics are learned from trial and error using a performance feedback function as reinforcement. The authors claim that by playing against a computer controlled aircraft which has some pre-determined, fixed rules of behaviors for an air intercept task, their study shows that the learning system controlled aircraft is able to learn and automatically acquire rules of behaviors that outperform the fixed rules.

2.2 Behavior Validation Techniques

While the behavior models used for CGF are being developed, various validation techniques have also been proposed to ensure the validity of the models. Validation techniques have been classified into four main categories: *Informal*, *Formal*, *Dynamic*, and *Static* [1].

Informal techniques are among the most commonly used. They are called informal because the tools and approaches used rely heavily on human reasoning and subjectivity without stringent mathematical formalism. Examples of Informal techniques include documentation checking, face validation, reviews, and Turing test.

Formal techniques are based on mathematical proof of correctness. Although proof of correctness is the most effective means of validation, current available proof techniques are simply not capable of being applied to even a reasonably complex simulation model. Examples of Formal validation techniques include induction, inference, logical deduction, predicate calculus, and predicate transformation.

Dynamic techniques require model execution and are intended for evaluating the model based on its execution behavior. Examples of Dynamic techniques include acceptance testing, compliance testing, execution testing, functional testing, interface testing, statistical techniques, and structural testing.

Static techniques can be applied to perform conceptual model analysis, control flow analysis, data analysis, and interface analysis. Static techniques are concerned with accuracy assessment on the basis of the characteristics of the static model design and source code.

Simulation developers can apply static analysis at different stages of the model validation process, prior to machine execution of the model.

2.3 Behavior Creation – The CBT Approach

The CBT is developed to improve the efficiency and the ease of creating behavior models. In CBT, the behaviors are created using a graphical Behavior Editor [3]. Figure 1 is a screen capture of the editor's main interface. The top left panel is the Behavior Palette from which the user can choose various types of the behavior nodes and connectors. The bottom left panel is the Attribute panel where the user defines Behavior Attributes such as the "Route and Speed for a Move" behavior.

The Behavior panel is where the user can graphically define a Composite behavior using the available behaviors defined in the Behavior Repository. These primitive behaviors form a base set of behaviors in the Behavior Repository that are available as basic building blocks for users to build complex behaviors. Users can choose behaviors from the repository based on the behavior definitions, such as category, domain and echelon, etc. When the user finishes building the behaviors, the composite behaviors can be saved for execution or as building blocks for other behaviors. The behaviors defined using the graphical interface are saved internally in the XML format.

In [3], the CBT implementation team gave a list of validation rules that should be implemented (but were lacking), including checking for unreachable behavior nodes, un-decidable decision point, valid constraints, behavior consistency, etc. Therefore, a comprehensive tool for behavior model validation is indeed desirable for future releases of the CBT.

3. The LogicChecker

The LogicChecker is the tool we developed to perform static analysis and validation of user-composed behaviors using CBT's Behavior

Editor. The tool adopted various static analysis methods to validate the behaviors before the behavior executions.

3.1 User-Created Errors in CBT behaviors

Based on the experiences we gained by building behaviors using CBT's Behavior Editor, and the input from our customers that used the CBT in the past, we generated a list of potential errors that may occur during the behavior building process. We categorize the errors into four types: Element Errors, Constraint Errors, Attribute Errors, and Reachability Errors.

Element Errors are related to missing behavior elements, such as edges, behavior nodes, and predicate branches; Constraint Errors are related to incomplete conditional constraints or incomplete/incorrect timing constraints; Attribute Errors are related to missing behavior attributes, wrong attribute types, or uninitialized attributes; and Reachability Errors are related circular dependencies or unreachable behavior nodes.

3.2 Static Analysis Methods

Our goal was to implement a tool that performs static analysis and validation of the behaviors defined using the CBT's Behavior Editor. As alluded to earlier, *static* validation techniques are concerned with accuracy assessment on the basis of the characteristics of the static model design and source code, prior to machine execution. The LogicChecker tool applies the following techniques for static analysis:

Data Usage Analysis: This analysis ensures that each data item defined within the model is used in a way compatible with its definition and with its description in the requirement. As a result, this type of analysis can be applied to check Element Errors, Constraint Errors, and Attribute Errors.

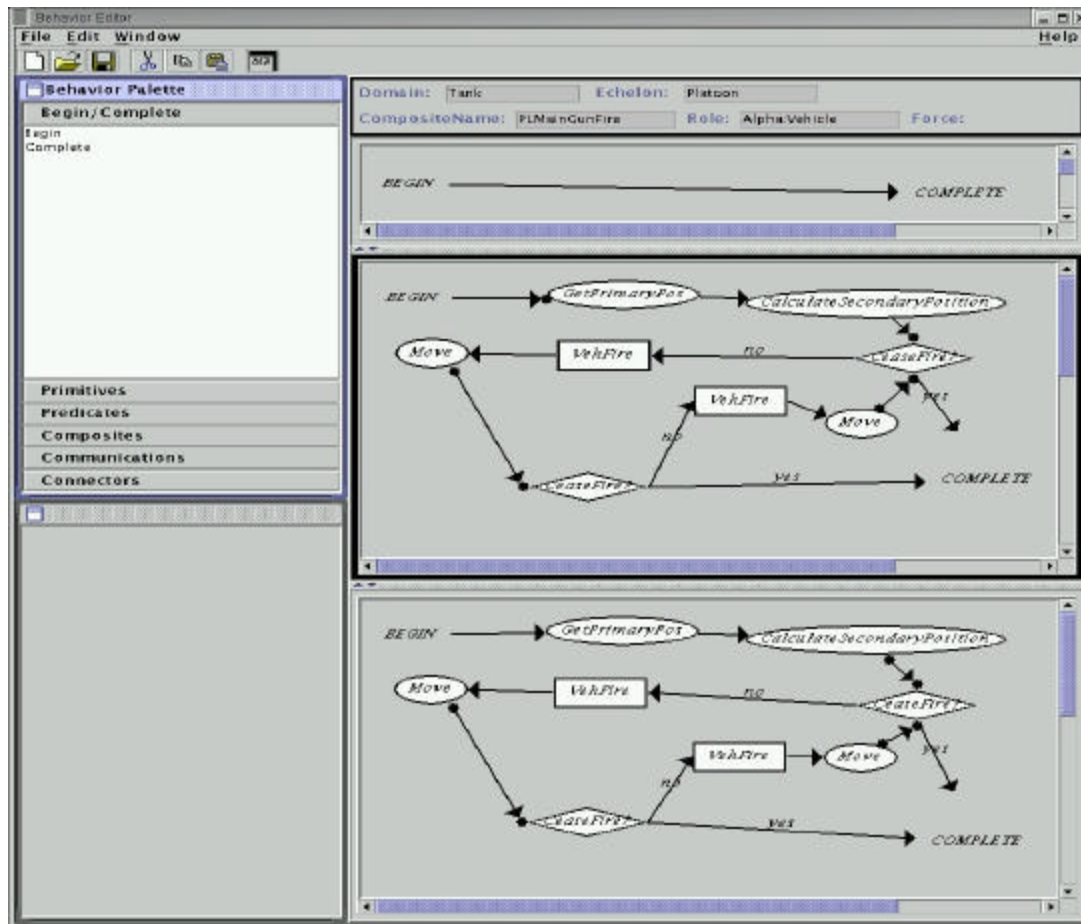


Figure 1: CBT's Behavior Editor

Data Flow Analysis: This analysis scrutinizes both the logical structure of the behavior model and the data usage properties of each XML statement. This analysis can be used to check Constraint Errors, Attribute Errors.

Control Flow Analysis: This analysis is used to ensure that the complicated resulting logics of composite behaviors are correctly represented and organized. It can be applied to check Timing Constraint Errors, Conditional Constraint Errors and Reachability Errors.

Constraint Analysis: The task of constraint analysis provides methods to ensure all the constraints are defined and used correctly. It is used to detect Timing Constraint errors and Conditional Constraint Errors.

3.3 Architecture Design of the LogicChecker

The architecture design of the LogicChecker is based on a model that captures the essential aspect of available static analysis techniques, as well as the real world usage of the model. We adopted an Object-Oriented analysis technique called Scenario Analysis, as defined in [2].

The Scenario Analysis is the central activity of any Object-Oriented analysis. In this analysis, we first identified the classes and objects, we then identified the primary function points of our error-checking tool and grouped them into clusters of functionally related behaviors.

We categorized the logic errors into two different types: Data errors and Control errors. We also identified the main sources needed to track the errors, which are CBT's Behavior Editor and

CBT's Repository Manager. The reason is that some errors can be identified directly from the Behavior Editor, and others can only be caught if more behavior information from the Behavior Repository is provided. For example, in order to catch a missing branch error for any Predicate behaviors, we have to access the Repository Manager to find out how many branches this predicate is defined to have.

3.3.1 Identify Classes and Objects

In order to better define the classes, we also made use of the concept of Design Patterns. The concept of Design Patterns has been used to solve recurring problems in software design. Among the different types of Design Patterns, structural patterns, which deal primarily with the static composition and structure of classes and objects, is used to help us in factorizing and generalizing the components in the architectural design and analysis. By applying these design techniques, we devised an object-oriented decomposition of our application and generated a set of candidate classes and objects, as well as the relationships among these them.

This main class LogicChecker is where the CBT composite behaviors are input from the XML files and the different types of error checking are kicked off. Making the LogicChecker class the place to kick off error checking gives us the flexibility of turning on or off any types of error checking. The LogicChecker application is designed to be instantiated by the CBT Behavior Editor, but it is also a generic component that can be adapted to incorporate additional types of error checking.

3.3.2 Identify the Relationships Among Classes and Objects

After carefully defined the patterns of error checking behaviors, we also clarified the common roles and responsibilities for each type of checking. By doing so, we derived four instantiated classes from the LogicChecker class: ElementChecker class, ConstraintChecker class, AttributeChecker class and ReachabilityChecker class, which share the common operations and methods on retrieving data and checking for errors. We made these classes as flexible as possible so they can be adapted to different contexts for more specific error checking. And we used the same design patterns to define more specific classes according to their class

categories: EdgeChecker, NodeChecker, BranchChecker, are subclassed of the ElementChecker; ConditionalConstraintChecker, TimingConstraintChecker are subclasses of the ConstraintChecker; Reachability Checker and CircularDependencyChecker are subclasses of the ReachabilityChecker.

The milestone of this design phase is a specific class category definition for the whole application, which contains a three-layered hierarchy according to the result of both architectural analysis and class identification analysis, shown in Figure 2.

We then validated the architecture through a prototype where we created an executable release that partially satisfied the semantics of some fundamental scenarios. The details of the implementation and testing are described next..

3.4 Implementation of the LogicChecker

Since the composite behaviors we are validating are stored in the XML format, the first step of our implementation is to obtain an XML parser. In order to keep the development cost down, as well as to achieve speed, conformity and validation purposes, Apache's Xerces DOM Parser was chosen for its simplicity and stability. Next, a parallel effort led by the Institute of Simulation Technology (IST) of the University of Central Florida produced a Document Type Declaration (DTD), which defines how the behaviors should be represented, such as what kind of data and its attributes are allowed in the document, the acceptable attribute values for each data element, the nesting and occurrences of each element, and whether there are any external entities needed. It should be noted that a well-formed behavior representation doesn't guarantee valid behaviors, which is why a validation tool like LogicChecker would be useful.

It is our goal to achieve a good balance between structure and efficiency in the implementation of the tool. Our object-oriented analysis and design have already provided us with a reasonable structure that we tried to keep as much as possible. The implementation starts from the LogicChecker class.

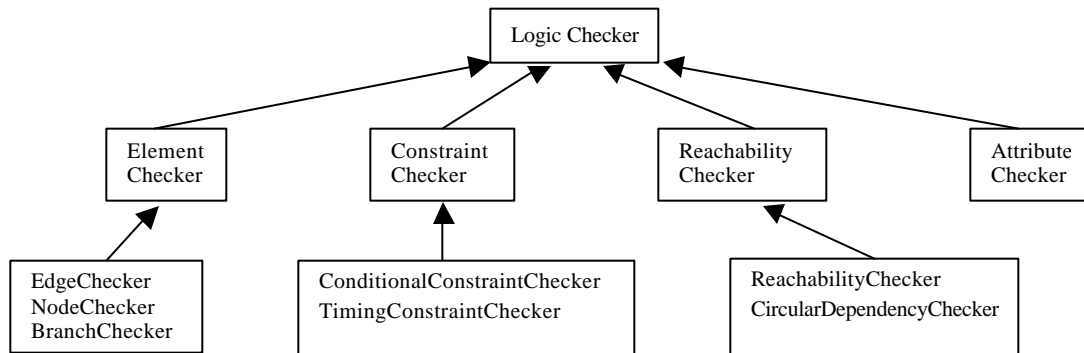


Figure 2: LogicChecker Class Hierarchy

The immediate subclasses ElementChecker, ConstraintChecker, ReachabilityChecker and AttributeChecker are defined based on the common characteristics of the error types. Each of these main error-checking classes encapsulates operations and methods that can be used to perform the specific type of behavior validation. And their subclasses can easily inherit and overload the operations and methods for more specific uses.

The detailed implementation information can be found in [6].

4. Testing Results

By analyzing and validating the XML representation of the composite behaviors, the LogicChecker successfully checks data related errors and control related errors we proposed to detect. The test results of our application LogicChecker have proven that this tool successfully performed the static analysis and validation on the CBT behaviors. The benefit of the tool is that it can detect most of the human errors and logic errors, so that the behavior developers can correct the errors before the behavior execution, therefore saves lots of trouble and headaches in the runtime behavior validation process. The test results also proved that the tool is stable and reliable. The use of Object-Oriented analysis and design provided the modularity and reusability for our application.

5. Conclusion

Our research and development successfully prototyped an application tool that uses static analysis methods to validate user-composed behaviors developed using Composable Behavior Technologies. We not only utilized the classic static analysis methods and Object-Oriented programming concept, but also adopted advanced JAVA and XML technologies to achieve usability, reliability and stability of the tool.

In the meantime, we also discovered additional areas for further research and development. Recommended areas are:

- Integration of LogicChecker into the CBT. Although the LogicChecker can be used as a stand-alone application, by integrating into the CBT system, it would be able to access the CBT behavior database, therefore to check any errors that need information from the database, such as the behavior mission attributes and predicate branch definition. It is possible to integrate the LogicChecker as part of the CBT Behavior Editor so that it can be turned on or off by a button click.
- Advanced Error Report Mechanisms. The current error report is very limited and preliminary. It only reports what the error is, and which behavior generates it. It would be much more useful for the

users if the error report shows the lines where the errors are generated in the XML behavior representation, by highlighting the lines, as well as giving a suggestion on how to fix it. In addition, the report can also flashes the behavior components that are related to the errors on the CBT's Behavior Editor.

- Behavior Input/Output Data Validation. Some CBT behaviors can be built to return some data values for the next one to use. Although the primitive behaviors are the ones being executed in the SAF and return values, and SAF enforces the type of the return data, the value of these data needs to be analyzed and validated to ensure the next CBT behavior get the correct and usable data.
- Behavior Mission Role Validation. The concept of using Mission Roles in the CBT is a must when building behaviors for units that contain several levels of force hierarchies. Since CBT's Behavior Editor only supports two levels of hierarchy in each behavior building process, it is very important to validate the consistency of the mission role definitions. This analysis and validation would involve the behavior database access.

6. Acknowledgements

Part of the work described in this paper was carried out by SAIC for STRICOM.

7. References

- [1] Balci, O. Verification, Validation and Accreditation of Simulation Models. In *Proceedings of the 1997 Winter Simulation Conference*, Dec. 1997, pp. 135-141.
- [2] Booch, Grady. *Object-Oriented Analysis and Design with Applications*. 1994
- [3] Courtemanche A., von der Lippe, S. and McCormack, J. Developing User-Composable Behavior. In *Proceedings of 1997 Fall Simulation Interoperability Workshop*, Sept. 1997.
- [4] Henningger, A., Gonzalez, A., Georgipoulos, M. and Demara, R. Modeling Semi-Automated Forces with Neural Networks: Performance Improvement through a Modular Approach. In *Proceedings of the Ninth Conference on Computer Generated Forces*. May 2000.
- [5] Pew, R. and Mavor, A., editors. *Modeling Human and Organizational Behavior*. National Academy Press, 1998, pp. 40-41, 217-219.
- [6] Zhang, Jackie Z. *Static Analysis and Validation of User Composed Behaviors in CGF*. Master's Thesis, 2001. UCF Library.

8. Author's Biographies

Jackie Zheqing Zhang has five years of software development experience in ModSAF/OTBSAF related projects. She was the technical lead for the Advance Concept Techenology II/CBT project. Ms. Zhang received her Master's Degree of Computer Science from University of Central Florida in 2001.

Bill Hopkinson has over 16 years of experiences in system engineering in various areas. He has been the project lead for numerous Modeling and Simulation experiments. He received his Ph.D. from University of Central Florida in 1997.

Sheau-Dong Lang is an associate professor in Computer Science at the University of Central Florida. His research interests include algorithm designs in information storage and retrieval, databases, and simulation and training systems. He received his Master's degree and Ph.D. both from the Pennsylvania State University.