

ENHANCING UNDERSTANDING OF MODEL BEHAVIOR THROUGH COLLABORATIVE INTERACTIONS

*Dr. C. Michael Overstreet
Dr. Irwin B. Levinstein
Old Dominion University
Norfolk, Virginia USA
{cmo,ibl}@cs.odu.edu*

ABSTRACT:

We believe model understanding can be facilitated through interactions between model users and models if, during execution, the model can help explain the reasons for the actions it has taken. We report on a prototype in which an existing simulation was modified in an attempt to help both developers and users better understand the causes of the behaviors exhibited during a simulation. During an execution, an animation shows what the model is doing while a second animated graphical representation, here in the form of statecharts, helps observers understand the reasons for particular model actions. We discuss some additional techniques not present in the prototype such as support for interactive interrogations of models to help observers understand the reasons for sequences of model actions.

Keywords: Interactive simulation, Model understanding, Model verification

1. INTRODUCTION

Reduced hardware costs and increased hardware performance have allowed the development of more complex software systems, including simulation models. As these simulations have become more complex, it can be more difficult to determine whether behaviors observed during executions reflect behavior of the system being simulated or are the result of errors in modeling or a reflection of implementation errors. We describe implementation details of an approach intended to assist modelers, developers, and users of simulations to more easily understand and verify model behavior. Displaying model actions and the reasons for those actions during execution is the approach taken.

We describe our implementation, provide samples of outputs generated by the implementation and after a brief evaluation of the strengths and weaknesses of the approach, discuss concepts for extending the concept to enhance its utility. We also discuss some additional ideas for improving a model's ability to explain its behavior by

enabling an observer to interactively explore the reasons for a sequence of actions.

1.1 MOTIVATION FOR APPROACH

Our original motivation for this work was to explore alternate graphical model specification forms that could be used to specify model behaviors and which might be easily understood both by people responsible for implementing the models and by people responsible for verifying the model specifications. We anticipated that animating the graphical specification during execution would be beneficial for both uses

A common approach to help verify simulation models, particularly in the U.S. military, is the use of "subject matter experts" (SMEs), who are often retired military personnel, to observe model executions to confirm correct model behavior or to identify incorrect behaviors. We have observed some situations in which SMEs had initially believed that an observed behavior was erroneous until they could later determine that the model action was due to situations not readily apparent to the observer (for example, an aircraft was responding to a radar signal). Thus animation of the graphical specification might help these individuals more easily understand the reasons for particular model actions.

2. IMPLEMENTATION DESCRIPTION

ModSAF (MODular Semi-Automated Forces) is a simulation that has been widely used in war games and military simulations [1]. ModSAF simulates the movement and behaviors of many types of entities, where an entity can be anything from an aircraft, ship or tank to a single dismounted infantryman. It also supports aggregates of entities such as squadrons and battalions. Each entity type typically has several associated behaviors that are selected depending on the mission and circumstances of the entity involved. For example a tank moving from one location to another may execute behaviors that cause it to follow a prescribed route, avoid collisions, and watch for enemy vehicles. ModSAF is written as a distributed simulation

Our implementation is also based on BetterState [3], a tool developed by a student of David Harel, that facilitates software development using statechart models. This tool helped with the project since it has the following capabilities:

- Statecharts are constructed by drawing them.
- The tool can generate the code for the statechart and its transitions.
- The tool can animate the statechart used for the design of an application while the application is executing.
- The tool can capture the state transitions occurring during a run and allow them to be replayed.

3. IMPLEMENTATION DETAILS

This project was a proof of concept only of the last idea, that of animation. To this end, we selected an existing behavior and extracted its FSM. From the FSM we reverse engineered a more easily understood statechart that specifies the same behavior as the FSM. We devised means to semi-automatically collect and transmit state transition information as the simulation model executed in the DRE environment.

The communications so triggered were received on another workstation. This machine displays an animation of the model, textual information, and a second animation of the corresponding statechart. BetterState displays the statechart in DDE (Direct Data Exchange, a precursor to OLE) as it interacts with our communications server program. The server itself consists of a communications manager and a module specialized to the particular statechart and FSM involved. To simplify the recoding task, we devised tools by which code for the specialized module could be automatically generated by a suite of programs using a point and click interface.

3.1 MODIFICATION OF THE AWK PREPROCESSOR.

One aim of the project was to animate the state changes in the statechart whenever the DRE module's FSM changed state. A second was to display additional explanation. Consequently it was necessary to modify the DRE module so that the programs on the BetterState computer would be notified of the state changes and pass additional debugging information. Rather than

modify the DRE module directly, we sought to automate the process.

As mentioned above, the FSM of the DRE module is written in "fsm," a language that is a superset of C. It also contains constructs to simplify the coding of the module's FSM and to create the boilerplate code needed to incorporate the module into the DRE environment. A module in FSM form is transformed into pure C code by the fsm2ch.awk preprocessor. Since the preprocessor already understands the FSM, modification to decorate its output with appropriate calls to communication routines were relatively simple. Wherever the preprocessor was programmed to recognize a transition, we modified it so that a call to a communication routine is added.

One of the parameters to that communication routine is a text string. In the original implementation code, since most state transitions are implemented using an if-else statement whose boolean expression describes the condition causing the transition, we included this expression as the text string. We found that in most cases, the variable names chosen by programmers were sufficiently suggestive to describe the reason for the state transition. Fsm2ch.awk was further modified to collect this information as it processed the input file. While not always ideal for explaining the reason for the state change and accompanying actions, the information collected is passed to the communication routine and was generally helpful in explaining model behavior. Only information statically available at the point of call is passed. For example, should the state change in the if part of an if-else statement, only the if part (including the condition) is passed to the communication routine. We found that awk is less than adequate for this kind of task in general. If this sort of code analysis for explaining behavior proves desirable, the construction of an additional tool is warranted, perhaps using more appropriate tools such as lex and yacc. However this modification illustrates a useful possibility in the proof of concept.

3.2 COMMUNICATIONS

Communications were implemented via a server on the BetterState side and a client on the DRE side using TCP/IP sockets. The client establishes a session on the first call and thereafter sends the transition information to the server at each state change. The server displays the information in a list box and saves it for replay. It also translates the transitions on the FSM of the DRE module into transitions on the statechart. BetterState is

notified via DDE of the transition and thereby animates the displayed statechart diagram.

3.3 THE BETTERSTATE SIDE SOCKET SERVER

The BetterState server consists of two parts. The first communicates with any DRE module as described above. That part listens for the client and receives the transition and textual information that it displays and also saves for replay. The second part is specific to the particular DRE module required to translate the transitions in the DRE FSM into transitions in the BetterState statechart. We developed a tool to automate the construction of the second part of the server program. This tool accepts as input a set of state names and a set of transition names. It then displays an empty matrix of possible transitions that can be completed using a point-and-click interface. From this information, the tool generates the code for the translation module of the server.

Figure 2 is part of a screen shot of the BetterState machine during a simulation. This is the animation that allows someone to observe the behavior of a simulated entity (in this case, a tank). In our prototype, as the simulation proceeds, the animation shows the positions and action of the two tanks displayed in the image.



Figure 2: Animated Representation of Vehicle Move Behavior,

Figure 3 is also taken from a screenshot of the system; it contains the statechart that defines the tank's behavior. During execution, the current states are shown in red. For example, figure 3 shows the tank to be in the "checking," "scanning," and "moving." states. If a "detect" event occurs, it switches from "scanning" to "attacking" and from "moving" to "disabled." On a "we-win" event, it returns to the "moving" and "scanning" states

4. PROTOTYPE EVALUATION

We feel that this proof-of-concept was successful. It demonstrated the simplifying nature of statecharts, the remote display of real time explanations, and the remote animation of a statechart corresponding to a behavior. We showed that it is possible to automate the conversion of behavior modules to make them displayable and to automate the construction of translators on the server side.

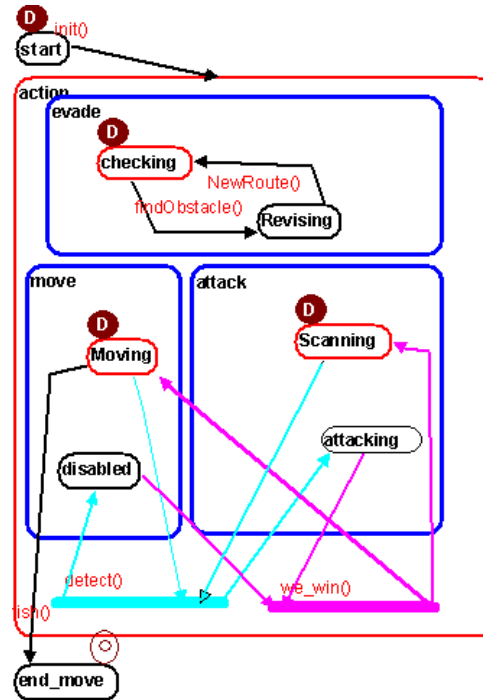


Figure 3: Statechart Representation of Vehicle Move Behavior

4.1 POTENTIAL IMPROVEMENTS

Since this effort was simply a proof-of-concept prototype, several improvements would increase its utility:

1. Improved information gathering. As indicated above, the parsing of the source code is less than adequate. An additional preprocessor built with compiler constructors such as lex and yacc is needed to do the job properly. In addition, complete explanations of behavior require access to the changes in data values that determine the state changes.

2. Multiple entities under examination simultaneously. It would be a relatively minor elaboration to make it possible to observe the statecharts of multiple entities at the same time.

3. Multiple behaviours under examination simultaneously. Entities are governed by a collection of behaviors. It would be illuminating to examine the interaction of several behaviors on the same entity. BetterState provides a means (“threads”) for constructing interacting, semi-independent statecharts.

During model execution, this software makes three types of information are available to observers: 1) an animation to show what the model is doing, 2) a statechart representation to provide information on why the model behaves as it does, and 3) boolean expressions, as text, providing additional information regarding why the model does what it does. Clearly, the utility in displaying boolean expressions of program variables depends on the programmer choosing variable names carefully.

4.2 DISPLAYING ACTION CAUSES

Our hope is that the information provided will be useful to several audiences: programmers (for debugging), subject matter experts (responsible for verifying the simulation), and in some circumstances, to users of the simulation (to more easily understand the reasons for system behaviors).

One commonly expressed benefit by users of simulation is the enhanced understanding of the system of interest gained through the process of specifying and verifying the simulation. We hope these explanations can be useful for this purpose.

Many of the representational forms used by model builders may not be immediately useful to users or subject matter experts. However, we suspect that if a model is easily accessible to explain the notation during execution, these graphical forms can still be helpful.

5. CAUSAL SEQUENCES

In this proof-of-concept activity, we started with an existing implementation that was not designed to provide the new animations. This strongly influenced what was done and the graphical representations used. We were able to take advantage of this implementation’s use of FSMs. Many other graphical representational forms are widely used in simulation; several might provide a more easily understood basis for explaining why the simulation chooses the actions it takes. Graphical representation forms include, among many others, the icons introduced very early with GPSS [4], Event Graphs [5], finite state machines, and statecharts [2]. For reasons

discussed above, we chose to use statecharts as a means of showing the reasons for model actions.

From a conceptual view, a simulation consists of a sequence of actions that occur over a period of time. In a pure discrete simulation world view, each of these actions is called an event. Extending the discrete even view slightly, each event can be either “determined” or “conditional.” Determined events are characterized as “time-based” and are scheduled to occur at a particular simulation time by some preceding event. Conditional events are characterized as “state-based” and occur because some event has just occurred that changed the state of the simulation so that the conditional event should occur at the current simulation time. In either case, each event is caused by one or more particular events. In addition, each conditional event occurs based on a controlling condition (a boolean expression) that has changed from false to true due to the actions of some model components. Nance [6] and Overstreet [7] discuss and illustrate the benefits of recognizing these causal relationships.

In an approach similar to what was done in the prototype described in section 3, it should be a straightforward task to gather and display information that shows the cause for each event as a simulation progresses. Since the prototype both displayed and saved the information generated during execution, these animations could be replayed without the simulation reexecuting. And if stored appropriately this information could be searched for interesting situations and then replayed with VCR-like controls, allowing an observer to interactively explore the causes for interesting, unexpected, or confusing sequences of events. This is similar to a trace, provided as a debugging tool by many simulation languages, but with the advantages that causality could be interactively explored to answer questions about the simulation behavior.

If this easily obtained information is stored appropriately, an observer could then explore, in an interactive fashion, the events that occurred and the causes for each event. For determined events, the cause reported could be the event that scheduled it at an earlier simulation time and the time that causal event occurred. For conditional events, the cause displayed can be both the condition (as a boolean expression) controlling the occurrence of the event and the event or events that changed the value of one of variables used in that boolean expression along with the times at which each event occurred. If desired,

the observer could then examine the cause for each of these events.

This approach has a flavor reminiscent of Weiser's concept of program slicing in which, in response to a user's request, only the subset of a program that influences the value of specified variables at particular point in the program is displayed [8]. However, this apparently simple idea has both run-time complexity problems and has proven to be difficult to implement completely and has led to deep research issues; it is an active research area. In contrast, recording the sequence of events and their causes presents no computation problems as they can be captured as a trace during actual execution.

6. OTHER ISSUES

We have proposed the use of providing explanations as a way of helping people cope with the complexity of large or long running simulations. Additional support will be necessary to accomplish this. In a large complex and long running simulation, it is infeasible for individuals to observe all behaviors that occur during a simulation run. This issue is a recognized problem in the task of verifying simulations. As is common with testing complex systems, a set of important and revealing situations to check must be identified and efforts focused on checking those.

It is often too easy to get lost and confused by details. Hiding distracting details is necessary, but equally important to supporting understanding is the ability to show the right details at the right time. Some support for showing only relevant information can be provided by allowing users to interactively explore the reasons for model actions as they see fit. Another approach to reducing the volume of details could be to allow a user to identify particular behaviors to be observed (just as testers identify particular situations to test). This could be supported by a capability such as allowing a user to identify a situation of interest, a letting the simulation run until that situation occurs, then allow the user to interact with the simulation in order to explore what led to the situation of interest. A similar approach could be to allow a modeler to identify situations that should never occur (invariants); in some situations is easily done. If they should occur, an observer could then explore the sequence of events and the causes that gave rise to the incorrect situation.

Abstraction is a key tool in enhancing understanding. If the statechart representations of

the prototype are useful, it is because they provide a higher degree of abstraction than the C code that is the implementation. They hide many uninteresting details (for some purposes) about the code and display information more helpful for enhancing understanding.

While displaying the reasons for model actions can enhance understanding, many other capabilities could also prove helpful. Discovering those of general benefit to a wide class of users remains a challenging problem.

7. SUMMARY

We built a prototype by modifying a complex existing simulation so that it displays three additional types of information: an animation, a statechart showing the current state of the system and the state changes, and a textual display of the boolean expressions that were used in the implementation code to cause model behavior. This idea is similar to that long used in the artificial intelligence community with expert systems where, as systems became more complex, it was quickly recognized that a system must be able to explain the reasons that led it to particular decisions [9]. The explanations are crucial for both debugging and verification.

While we believe that providing a mixture of the types of information identified is useful for understanding system behavior, the added complexity present in many modern simulations requires additional approaches to support users as they attempt to understand the behavior of these complex systems.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the support and the interesting discussions with Brent Kornman of Lockheed Martin Corporation for this work. Additionally, Mohamed Owis and Yan Fu, former computer science graduate students at Old Dominion University, made significant contributions to this work.

REFERENCES

- [1] ModSAF, <http://www.modsaf.org>.
- [2] D. Harel: "A Visual Formalism for Complex Systems," *Science of Computer Programming*, Vol. 8, pp. 231-274, June 1987.
- [3] BetterState, <http://www.windriver.com>.
- [4] G. Gordon, 1966 "Simulation Languages for Discrete Systems," In *Proceedings IBM Scientific Computing Symposium on Simu-*

- lation Models and Gaming, White Plains, New York, pp. 101-118, 1966.
- [5] L. Schruben, "Simulation Modeling with Event Graphs," *Comm. ACM*, Vol. 26, pp. 957-963, November 1983.
 - [6] R. E. Nance, "The Time and State Relationships in Simulation Modelling," *Comm. ACM*, Vol. 24, No. 4, April 1981, pp. 173-179.
 - [7] C. M. Overstreet, "Model Specification and Analysis for Discrete Event Simulation," Ph. D. dissertation, Virginia Tech, Blacksburg, VA, 1982.
 - [8] M. Weiser, "Program Slicing," *IEEE Transactions on Software Engineering*, Vol. SE-10, pp. 352-357, July 1984.
 - [9] W. Swartout, C. Paris, J. Moore, "Explanations in Knowledge Systems, Design for Explainable Expert Systems," *IEEE Expert*, Vol. 6, pp. 58-64, June 1991.

AUTHOR BIOGRAPHIES

C. MICHAEL OVERSTREET is an Associate Professor of Computer Science at Old Dominion University. A member of ACM and IEEE/CS, he is a former chair of SIGSIM, and has authored or co-authored around 90 refereed journal and conference articles. He received a B.S. from the University of Tennessee, an M.S. from Idaho State University and an M.S. and Ph.D. from Virginia Tech. He has held visiting appointments at the Kyushu Institute of Technology in Iizuka, Japan, and at the Fachhochschule für Technik und Wirtschaft in Berlin, Germany. His current research interests include model specification and analysis, static code analysis and support of interactive distance instruction. Dr. Overstreet's home page is www.cs.odu.edu/~cmo. He can be reached by e-mail at cmo@cs.odu.edu.

IRWIN B. LEVINSTEIN is assistant professor and assistant chair of the Department of Computer Science at Old Dominion University. He has done research in the areas of human-computer interaction, privilege management, and simulation animation. His current research is in intelligent tutoring and document management. Dr. Levinstein's home page is www.cs.odu.edu/~ibl and his e-mail address is ibl@cs.odu.edu.