ENHANCING UNDERSTANDING OF MODELS THROUGH ANALYSIS

Kara A. Olson and C. Michael Overstreet
Department of Computer Science, Old Dominion University, Norfolk, Virginia, U.S.A.

Keywords: Discrete event simulation, Model analysis, Model understanding, Static analysis, Dynamic analysis.

Abstract: Simulation is used increasingly throughout research and development for many purposes. While in many cases the model output is of primary interest, often it is the insight gained through the simulation process into the behavior of the simulated system that is the primary benefit. This insight can come from the actions of building and validating the model as well as observing its behavior through animations and execution traces or statistical analysis of simulation output. However, much that could be of interest may not be easily discernible through these traditional approaches, particularly as models become increasingly complex.
The authors suggest several possibilities of how to obtain such insights. These analyses have other obvious uses including aid in debugging, verification and documentation. The authors, however, are primarily interested in how these analysis techniques can be used to help modelers gain additional insights into the models they are using or constructing.
The discussed techniques are used with significant benefit within computer science and software engineering; the authors believe these techniques can also serve simulation well. The authors' experience with these techniques thus far has involved discrete event simulations; their potential benefit with other model representations and implementation approaches has not yet been explored.

1 INTRODUCTION AND MOTIVATION

We begin with an unusual frame of mind about model specifications, but one that the authors find intrinsically appealing. In mathematics, an axiomatic system is any set of axioms from which some or all axioms can be used in combination to logically derive theorems. A mathematical theory consists of an axiomatic system and all of its derived theorems (Wikipedia, 2011). In our view, one's assumptions about a simulation can be viewed as a set of axioms, with the properties of that system being our derivable theorems. Several approaches are used to discover these properties including observing a simulation during execution or analyzing data generated during simulation runs: but another approach is to reason about the system. With this in mind, consider then that our goal is to complement existing analysis techniques by "deducing" what we can about our system of interest.

It is often stated by users of simulation that its primary benefit is not necessarily the data produced, but the insight that building the model provides. Paul et al. discuss this in (Paul et al., 2005), noting that "simulation is usually resorted to because the problem is not well understood," and more often than not, the simulation is no longer of interest once the problem is fully understood. We believe that techniques such as those discussed here can be used to enhance both modelers' and model users' understanding.

A prime problem with model descriptions, whether in textual or graphical notations, is that even in simple models, descriptions are often difficult to fully comprehend. Paul accurately states (Paul and Kuljis, 2010):

> Even when we think we know what we are modeling there are many problems: we do not have the software skills to know if the software is doing the right thing; we cannot be certain that the logic of the problem is faithfully represented in the model; we cannot be sure that the assumptions built into the model, the uses it was designed to be put to and not put to, will be adhered to by future users etc. And then with the passage of time, and probably with some model updates, corrections, and possible changes of logic, we cannot be sure of the way the model works at all.

A. Olson K. and Overstreet C..
ENHANCING UNDERSTANDING OF MODELS THROUGH ANALYSIS.
DOI: 10.5220/0003597603210326
In *Proceedings of 1st International Conference on Simulation and Modeling Methodologies, Technologies and Applications* (SIMULTECH-2011), pages 321-326
ISBN: 978-989-8425-78-2

Insights can arise from many different sources. One can be surprised to learn that one event causes another seemingly-unrelated event. One can also gain insight when something that is expected to happen does not occur. Sometimes events can happen with regularity or as clusters which may not be noticed by a modeler and may reveal important aspects of the simulated system. Often these facts are not immediately obvious, particularly in large simulations (Overstreet and Levinstein, 2004; Nance et al., 1999). Anecdotal reports from modelers support the frequent difficulty of detecting important aspects of their models which when pointed out are quite useful.

Indeed, in one of our experiences, a model-coder was studying trace data produced during simulation executions. It was noted that the events that occurred could be divided into a small number of groups based on the number of times each event occurred; every event in each group occurred the same number of times. This observation revealed a structure of the model (and the system it represented) that had not been previously recognized; this fundamental system property may not have been apparent through more traditional techniques.

We suggest many possibilities and types of possibilities of how to obtain such insights throughout the next section.

2 DISCUSSION POINTS

What kind of analyses might reveal insights into a given model? First, let us consider potential beneficiaries of such analyses.

Certainly at least two groups could benefit from insightful discoveries: modelers and model users. A modeler, by our definition, is one who realizes the model into the computer: perhaps in terms of a programming language such as C++ or GPSS, or in terms of a visual product like Simio or Arena. A model user, then, is one who uses the simulation to meet some objective such as training, evaluating different scenarios, or perhaps trying to understand better the system at hand.

The analyses we discuss herein can benefit both groups in possibly different ways.

The types of analyses in which we are most interested are *static analysis* and *dynamic analysis*. With static analysis, an object (such as code or a list of specifications) is analyzed without executing it. With dynamic analysis, data is collected during execution of the object of interest (normally code). Dynamic analysis often requires inclusion of additional statements into the code to enable data collection during code execution, such as output or profiling statements. These two techniques have a long history of use in both the computer science community and the software engineering community to support, for example, code optimization and automated generation of some types of system documentation.

Previous model analysis research has used multiple model representation/specification forms such as Petri nets, DEVS and event graphs. An excellent survey of Petri net analysis can be found in (Murata, 1989). These techniques, though, require that a Petri net model be constructed and not all simulations lend themselves to this representational form. Zeigler's DEVS formalism (Zeigler, 1990) has been the basis of significant analysis work. However, the focus of much of this work concerns model verification – for example, (Hwang and Zeigler, 2006). Event graphs (Schruben, 1983) are also used as a basis for model analysis, though as our objectives differ from Schruben's, the information in our graphs differ. Each of these representational forms could support the types of analyses we discuss, but we do not explore their potentials here.

2.1 Insights – Static Code Analysis

Observing and analyzing the behaviors produced by a simulation are the usual techniques for improving understanding of a system being simulated. However, static analysis of the model specification itself can often reveal characteristics of a model not readily apparent from observing merely its run-time behavior.

In compiler optimization, several techniques are used routinely that could potentially provide useful insights to the modeler. Data flow analysis is used to help identify data dependencies – that is, to help identify relationships among different parts of the code. This analysis also can help determine interactions among variables in different model components, unrealized relationships, both causal and coincidental, relationships among different code modules, or relationships among different simulation components – relationships of which the modeler might not be so aware.

For example, using data flow analysis, one can potentially identify both the events which can cause a specific simulation event, and those events which can be caused by a specific event. If these lists can be generated, they can be informative by possibly identifying unanticipated effects previously unrecognized by the modeler. They can also serve a diagnostic purpose if the list omits events the modeler knows should be included, or includes events the modeler knows should not be included.

Another benefit of static code analysis: no finite number of runs can deduce what is possible. Not testing, execution traces or animations can necessarily discover all things that are possible in some simulation run. However, from static analysis, one can reveal the possibility of infrequent situations occurring.

Data flow techniques also can be used to determine if race conditions exist. In a common implementation of a discrete event simulation, an events list is checked before the clock is advanced. Ideally, the order in which events that are scheduled to occur at the same time happen to be on the list should not matter to the simulation results; however, analysis can flag this possibility – or point out a potential surprise to the modeler.

2.2 Insights – Dynamic Code Analysis

Static code analysis has its limitations; it can identify possible actions but since the possible complexity of code is unbounded, the results of the analysis can on rare occasions be misleading. From static analysis, one may discover that event *A* can cause event *B*, but dynamic analysis often can reveal specifics of which events caused which events – that is, which event caused a particular event, and which event(s) a particular event caused – which cannot always be determined prior to run-time. In addition, if static analysis suggests that event *A* can cause event *B*, but dynamic analysis reveals that this never happens, this may be of interest to a modeler or user of the simulation.

The ancedote in Section 1 about the grouping of events by frequency of occurrence is another example of useful information revealed through dynamic analysis.

2.3 Insights – Examining the Model Differently, Interactively

Alas, source code tends to involve many issues unrelated to the model itself, such as data collection, animation, and tricks for efficient run-time behavior. Unless the modeler is an expert programmer, this other code tends to obscure the model as implemented. Consider if one were able to be shown only what one considered currently relevant.

Weinberg identified the importance of program locality (Weinberg, 1971), the property obtained when all relevant parts of a program are found in the same place. He noted that "...when we are not able to find a bug, it is usually because we are looking in the wrong place." Since "issues of concern" vary widely, no single organization of a program can exhibit locality for all such concerns. Additionally, as the problem of interest changes, the information considered relevant also changes.

Consider if there were multiple model "views," where one could see only the aspects of (current) interest, and as the aspects of interest changed, so could what was shown to the modeler or model user. Perhaps these "slices," not unlike Weiser's program slices (Weiser, 1984), could aid in allowing model characteristics to be more easily understood.

An additional interactive possibility – a zoomable action cluster interaction graph – is discussed below.

2.4 Insights – Visualizing the Model

When given an unfamiliar model to modify or use, modelers and model users traditionally examine text output, source code, and perhaps animations if available. Animations aside, most analyses are not particularly visual, a shame since pictures help us build mental models (Glenberg and Langston, 1992) – in this context, a mental model of the encoded model.

Some work has been done on the "visuality" of models. Program visualization (which is distinct from visual programming – "the body of techniques through which algorithms are expressed using various two-dimensional, graphic, or diagrammatic notations," such as Nassi-Schneiderman diagrams (Nassi and Schneiderman, 1973)) has similarly stated goals: "to facilitate a clear and correct expression of the mental images of the producers (writers) of computer programs, and to communicate these mental images to the consumers (readers) of programs" (Baecker, 1988). However, the approach to doing so "focuses on output, on the display of programs, their code, documentation, and behavior" and "[displaying] program execution in novel ways independent of the method of specification or coding" (Baecker, 1988).

A prime problem with model descriptions, whether in textual or graphical notations, is that even for simple models, the descriptions are often difficult to fully comprehend. Even in relatively simple cases, the wealth of data available can easily obscure useful information in the volume of what is presented. The type of tools for which we argue may help with the problem of having "too much information" by allowing the interactive exploration of a model so that only relevant information is presented.

An information mural (Jerding and Stasko, 1998) is a technique for displaying and navigating large information spaces. The goal of the mural is to visualize a particular information space, displaying what the user wants to see and allowing the user to focus quickly on areas of interest. As Jerding and Stasko aptly state, "A textual display of such voluminous in-

formation is difficult to read and understand. A graphical view [...] could better help a software developer understand what occurs during a program's execution." We find this notion particularly appealing, since the source code is the true specification of the model as executed.

The action cluster interaction graph (Overstreet, 1982) is a type of dependency graph that can assist in model decomposition. The main purpose of this graph, derived from source code, is to show which events can cause which events. In these graphs, a solid line indicates that event *A* can cause event *B* to occur at the same instance in time; a dashed line indicates that event *A* will cause event *B* at a future instance in time.

Consider several modifications of this dependency graph. First, combining the static nature of this graph – which events can cause which events – with dynamic analysis – which event did cause which events – could prove insightful to a modeler. Second, imagine if the graph were "zoomable" – where one could see only one or a few steps at a time, rather than be overwhelmed with a graph of several hundreds, thousands or more events. We feel that exploring this graph interactively – perhaps as the simulation progresses, perhaps as the curiosity of the moment changes – could be insightful as well.

3 VISION

Our vision involves a unique product that can incorporate these types of analyses into one, useful, user-friendly tool, enabling both modelers and model users to gain additional insights into their models. This tool could start as something similar to CodeSurfer (Anderson et al., 2003), a commercially-available software static analysis tool based on prior Defense Advanced Research Projects Agency (DARPA)-sponsored research at the University of Wisconsin. CodeSurfer can create system dependences graphs – similar to our action cluster interaction graphs – based on static analysis, and allow those graphs to be queried in multiple ways.

Since these graphs explode as the number of events increases, our envisioned tool would allow the user to see the graph of all events – perhaps presented in a way akin to an information mural – and allow the user to select a particular event to follow, either statically (event *A* could cause events *B* or *C* to occur), or dynamically within a particular run at a particular time (in this execution trace, at time *t*, event *A* caused event *B*, which then caused events *C* and *D*, ...).

CodeSurfer also has multiple views that allow the user to view the parts of the code that s/he finds relevant at the time – analogous to our vision of showing the user only the currently relevant information.

3.1 Example

Consider a standard harbor model from (Nance et al., 1999): ships arrive at a harbor and wait for both a berth and a tug boat to become available; a ship is then escorted to a berth, unloaded, and escorted back to sea. This model is used to study tug boat utilization and ship in-harbor time.

A sample view of the model's action clusters might look like a subset of Figure 1. An action cluster is a collection of model actions that must always occur atomically. For example, an *initialization* action cluster in this context might consist of setting the mean interarrival time of ships; the unload time of a ship; how long a tug boat takes to travel while encumbered and while unencombered; how many tug boats and berths will be available; and so on. These actions will occur as an indivisible unit every time *initialization* occurs.

Figure 1: List of a sample model's action clusters.

With the harbor C code [Figure 2], using static analysis techniques, one can obtain its action cluster interaction graph [Figure 3]. As mentioned above, in these graphs, a solid line indicates that event *A* can cause event *B* to occur at the same instance in time; a dashed line indicates that event *A* will cause event *B* at some instance in time. Here, the simulation is started with the event *initialization*. An *initialization* event can cause an *enter*, *deberth*, *move_tug_to_ocean*, *move_tug_to_pier*, or *termination* event to occur at start-up, and will schedule an *arrival* event to occur; and similarly for the other nodes in the graph. As one can see, this graph is already non-trivial for a model of merely 12 events.

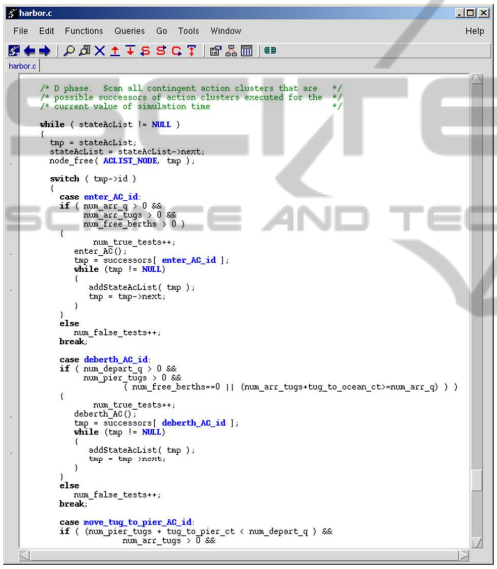

Figure 2: Sample source code.

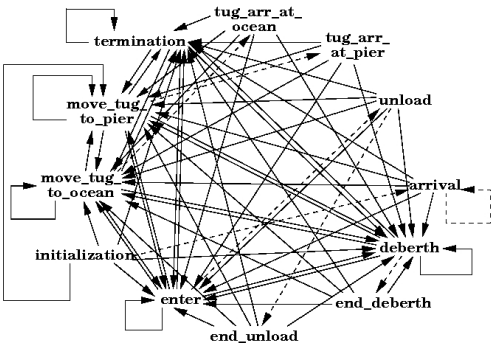

Figure 3: Action cluster interaction graph.

Let us take an example of a 26-event model. Static analysis could yield something like Figure 4; however, using an interactive tool, one could study, say, event *V*: first, the user could click on event *V* and see (more clearly) that it could cause events *E*, *R*, and *Q*; then, selecting event *Q*, s/he could see that event *Q* can cause events *F*, *W*, *G*, *R*, and *L* [Figure 5]. Given a specific execution trace and timeline of events, this type of visual could be used to traverse through the entire run, seeing which events caused which events.

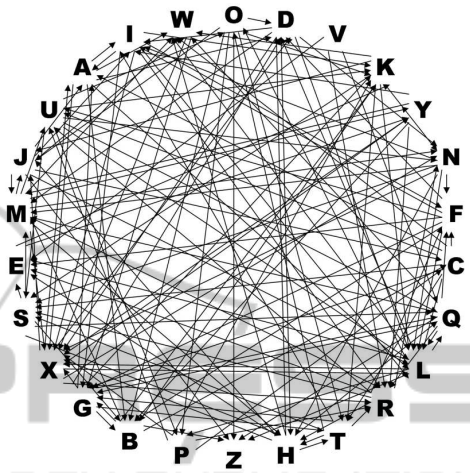

Figure 4: A sample action cluster interaction graph for a model with 26 events.

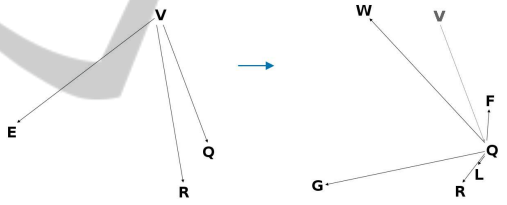

Figure 5: Exploring event *V*.

4 CONCLUSIONS

This work presents several challenges. Identifying a model specification form that works for model builders and model users and can be the basis for analysis techniques is challenging and will require a compromise due to conflicting needs. Additionally, identifying only the additional information likely to be useful to modelers and model users is a challenge. Use of modeler-directed interactive tools can assist.

In this paper, we have discussed a many-faceted approach to obtaining insights into a given model based on both source code and model execution. Types of analyses considered include static analysis, dynamic analysis, looking at the model differently, interactively, and visually – perhaps all at once, as different activities, or some combination thereof.

We propose a tool that would help with the problem of having “too much information” while incorpo-

rating these types of analyses. Indeed, our long interest is in enhancing model understanding – helping modelers gain additional insights into the models they are using or constructing.

REFERENCES

Anderson, P., Reps, T. W., Teitelbaum, T., and Zarnis, M. (2003). Tool support for fine-grained software inspection. *IEEE Software*, 20(4):42–50.

Baecker, R. (1988). Enhancing program readability and comprehension with tools for program visualization. In *Proceedings of the 10th International Conference on Software Engineering*, pages 356–366.

Glenberg, A. M. and Langston, W. E. (1992). Comprehension of illustrated text: Pictures help to build mental models. *J. Mem. Lang.*, 31(2):129–151.

Hwang, M. H. and Zeigler, B. P. (2006). A modular verification framework based on finite & deterministic DEVS. In *Proceedings of the 2006 DEVS Integrative M&S Symposium*, pages 57–65.

Jerding, D. F. and Stasko, J. T. (1998). The information mural: A technique for displaying and navigating large information spaces. *IEEE Trans. Vis. Comput. Graph.*, 4(3):257–271.

Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proc. IEEE*, 77(4):541–580.

Nance, R. E., Overstreet, C. M., and Page, E. H. (1999). Redundancy in model specifications for discrete event simulation. *ACM Trans. Model. Comput. Simul.*, 9(3):254–281.

Nassi, I. and Schneiderman, B. (1973). Flowcharting techniques for structured programming. *ACM SIGPLAN Notices*, 8(8):12–26.

Overstreet, C. M. (1982). *Model Specification and Analysis for Discrete Event Simulation*. PhD thesis, Virginia Polytechnic Institute and State University, Blacksburg, VA.

Overstreet, C. M. and Levinstein, I. B. (2004). Enhancing understanding of model behavior through collaborative interactions. Operational Research Society (UK) Simulation Study Group 2nd Two Day Workshop.

Paul, R. J., Eldabi, T., Kuljis, J., and Taylor, S. J. E. (2005). Is problem solving, or simulation model solving, mission critical? In Kuhl, M. E., Steiger, N. M., Armstrong, F. B., and Joines, J. A., editors, *Proceedings of the 2005 Winter Simulation Conference*, pages 547–554.

Paul, R. J. and Kuljis, J. (2010). Problem solving, model solving, or what? In Johansson, B., Jain, S., Hugan, J. R. M.-T. J. C., and Yücesan, E., editors, *Proceedings of the 2010 Winter Simulation Conference*, pages 353–358.

Schruben, L. (1983). Simulation modeling with event graphs. *Comm. ACM*, 26(11):957–963.

Weinberg, G. M. (1971). *The Psychology of Computer Programming*. Computer Science Series. Van Nostrand Reinhold Company, New York, NY.

Weiser, M. (1984). Program slicing. *IEEE Trans. Softw. Eng.*, SE-10(4):352–357.

Wikipedia (2011). Wikipedia. *Axiomatic system*, http://en.wikipedia.org/wiki/Axiomatic_system.

Zeigler, B. P. (1990). *Object-Oriented Simulation with Hierarchical, Modular Models*. Academic Press, Boston, MA.