

# ENHANCING MODEL UNDERSTANDING THROUGH STATIC ANALYSIS

Kara A. Olson and C. Michael Overstreet  
Department of Computer Science  
Old Dominion University  
Norfolk, VA 23529-0162  
{kara, cmo}@cs.odu.edu

**ABSTRACT.** Simulation is used increasingly throughout the sciences for many purposes. While in many cases the model output is of primary interest, often the insight gained by the modeler through the simulation process into the behavior of the simulated system is the primary benefit. This insight can come from the action of building the model as well as observing its behavior through animations, execution traces, or statistical analysis of simulation output. However, much that could be of interest to a modeler may not be easily discernible through these traditional approaches, particularly as models become more complex.

Static code analysis techniques can reveal aspects of models not readily apparent to the builders or users of the models, even when applied to relatively modest simulation models. Using a commercially available static code analysis tool, we were able to find documentation errors in the published paper, “Redundancy in Model Specifications,” by Nance, Overstreet and Page [8]. This additional information about model properties is unlikely to be detected in executing the models and contributes to the insights gained by modeling a complex system.

## INTRODUCTION & MOTIVATION

It is often stated by users of simulation that the primary benefit is not necessarily the data produced by the model, but the insight that building the model provides. At the most recent Winter Simulation Conference (December 2005), Paul *et al.* discussed this [10], 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 the static analysis techniques such as those discussed here can be used to enhance their understanding and complement insights gained through model execution.

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 [9, 8]. Anecdotal reports from modelers support the frequent difficulty of detecting important aspects of their models which when pointed out are quite useful. Static analysis of models has the potential to help a model builder interactively explore different properties of the model such as causal relationship among model components.

Automatically generated information about models has the potential of overwhelming users with too much information; using interactive static code analysis, it may be possible to offer the right information at the right time. For example, in an experience of Overstreet, a model-coder was studying the implementation of a model. Upon running it, it was noted that every event in one group occurred exactly the same number of times. Once this was observed, it became obvious that the model structure dictated this; however, having the right information at the right time could have enhanced understanding earlier than at the end of the implementation. As will be described shortly, in a project whose objective

was to explore the ability of existing code analysis tools to provide model information to modelers, a static slicing tool was used to analyze a model [8] and errors were found in the published documentation of the model. Although the goal was not to check for errors, the tool clearly could be used to provide better documentation and hence a more complete understanding of the model.

These analyses have other obvious uses including aid in debugging, verification and documentation. We, however, are primarily interested in how these analysis techniques can be used to help a modeler gain additional insights into models s/he is using or constructing.

### SIMILAR WORK

Program visualization and abstraction are similar notions [3, 7]; however, our objectives are often different from those of the program visualization community. In our work, the modeler may benefit from understanding the details of the model as it is realized in source code since the source code is the true specification of the model as executed. Alas, source code involves many issues unrelated to modeling, 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. The analysis techniques we discuss can help a modeler better understand a model as it is being constructed. By providing feedback, such as casual relationships, they can also help modelers better understand existing models.

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 – something of which the modeler might not be so aware.

An example of information which can be produced through data flow analysis techniques (though the ease with which this can be done depends on the model representation used) is the identification of both the events which can cause each simulation event, and those events which can be caused by each 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.

Control flow analysis, also from compiler optimization, can be used to determine which variables control which behaviors, something a modeler could possibly overlook. Control flow is especially useful for parallelization, something being done more often due to the size and complexity of newer models.

Using even these kinds of analyses, one can detect unrealized relationships, both causal and coincidental: relationships among different code modules, or relationships among different simulation components.

Some of the graphs we consider are similar to Schruben’s event graphs [12], though as our objectives differ from Schruben’s, the information in the graphs differ.

### CODESURFER

CodeSurfer is a software static analysis tool based on more than ten years of Defense Advanced Research Projects Agency (DARPA) sponsored research on system dependence graphs [2]. CodeSurfer is based on The Wisconsin Program Slicing Tool, a research project at the University of Wisconsin. It is designed to enhance program understanding using the system dependence graph of a program [1].

A *program dependence graph* [5] is a directed graph that represents one procedure in a program. It uses directed edges to indicate dependencies among different parts of the code. A *system* dependence graph [6] represents an entire program and is usually a combination of multiple program dependence graphs. A *dependence graph*, as used in our work, is a directed graph that shows dependencies among the code used to define the behavior of each event. That is, if event  $b$  is caused by event  $a$ , then there is an arrow from event  $a$  to event  $b$  in the dependence graph. A solid line indicates that event  $a$  can cause event  $b$  at the same instance in time; a dashed line indicates that event  $a$  can cause event  $b$  at a future instance in time.

CodeSurfer allows the system dependences graphs it makes to be queried in multiple ways. It also has multiple *views* that allow the user to view the parts of the code that s/he finds relevant at the time. These viewers are connected through hyper-text links. CodeSurfer obtained its name since it “allows surfing of programs akin to surfing the Web” [1]. One such view looks at a *backward slice*.

Program slicing [13] is a technique for finding a subset of a program that relates to the parts of the program that are of current interest. For example,

say one were to have a statistical program that finds mean, standard deviation and variance. If the latter two were found correctly but the former was not, it could be helpful to the programmer to only concern him or herself with only the bits of code that deal with obtaining the mean. This subset of the program would be a *slice*.

A *backward slice*, as articulated in [1], is a slice of a program with respect to a starting point  $S$  that answers the question, “What points in the program does  $S$  depend upon?” In our work, multiple backward slices are used to construct the dependence graph of the model.

## RESULTS

Using CodeSurfer and working with backward slices, Olson was able to find several uncaught errors in [8], involving a simulation of a harbor model with a simple dependency graph as most models go. The main purpose of this graph, derived from source code, is to show which events can cause which events.

In the harbor model from [8], 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. This is a version of the harbor model which appears in [4, 11].

In Figures (a) and (b), a *move\_tug\_to\_ocean* event can cause a *deberth* event to occur; also, a *deberth* event can cause a *move\_tug\_to\_ocean* event to occur. As mentioned earlier, a solid line indicates that event  $a$  causes event  $b$  to occur at the same instance in time, whereas a dashed line indicates that event  $a$  will cause event  $b$  at a future instance in time. A dotted line in Figure (b) indicates a correction or addition.

These figures also illustrate a prime problem with model descriptions whether in textual or graphical notations: even in simple models, the descriptions are often difficult to fully comprehend (even when automatically derived from source code). The type of tool we hope to develop may help with the problem of having “too much information” by allowing the interactive exploration of a model so that only selected information is presented.

## CURRENT & FUTURE WORK

As we have noted, it is difficult to separate code that defines the model – and hence is likely of primary interest to a modeler – from code that is present in order to run the model – for example, the details of adding events to lists.

We would like to be able to generate representations of different aspects of simulations such as: model representations which exclude implementation aspects; data collection representations which show only those parts of a model that effect particular statistics produced by the simulation; and model component interaction measures which might be useful for distributed simulations.

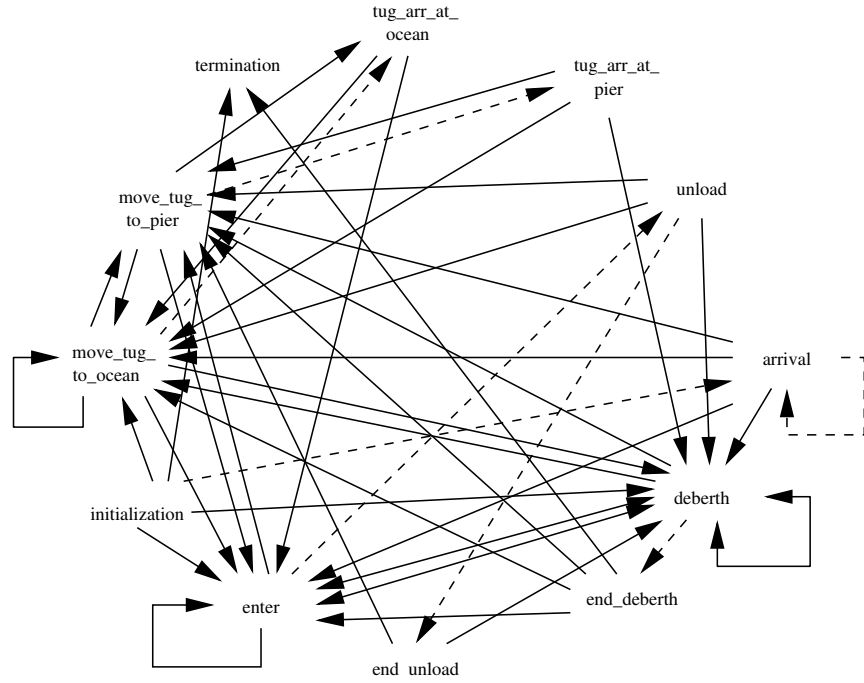
One instantiation of this may use CodeSurfer. CodeSurfer facilitates extraction of a wealth of information from a program. However, most model users are not computer program experts. CodeSurfer is yet another tool that would have to be learned, remembered, and manipulated. Seldom does it seem worth the time costs and effort to use such a tool, regardless of what insight it may offer, and such tools are soon cast aside or forgotten.

However, CodeSurfer comes with a Scheme (language) interpreter for extensibility. We hope to extend CodeSurfer such that a model user would only have to run our CodeSurfer script and obtain the dependence graph in a user-friendly, graphical representation. Certainly this would be more likely to be useful and to be used. With this information extracted and readily available, both understanding of the model and better evaluation of model correctness could be enhanced.

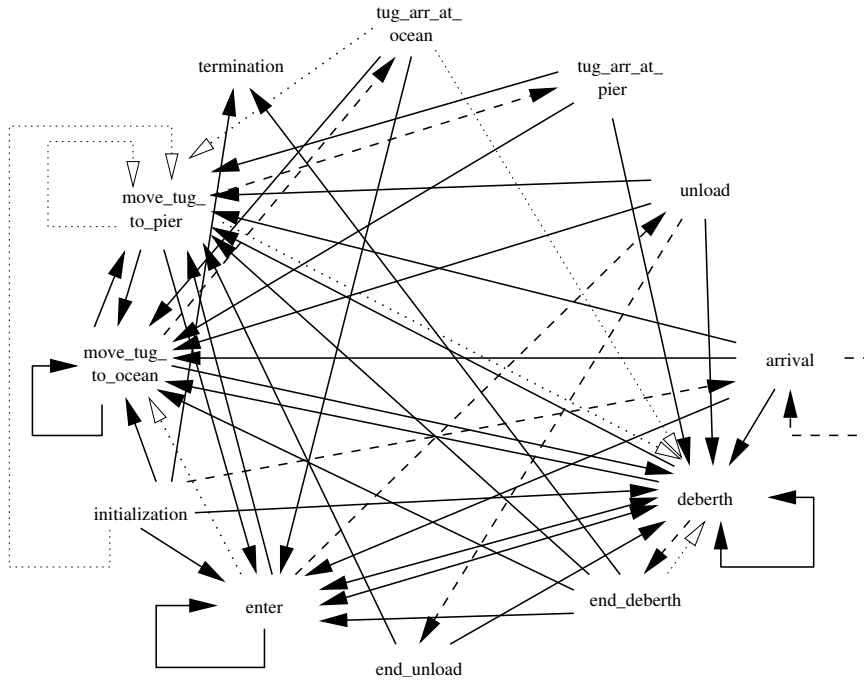
## CONCLUSION

Our experience has been that often model information obtained through analysis seems embarrassingly obvious after it has been pointed out – sometimes even obviating the need for the model, as noted by Paul *et al.* [10]. Our goal is to provide modelers with new information about their models – not just facts of which they are already aware. Selecting information likely to be useful is a challenge. Providing a tool which encourages interactive exploration of model characteristics should support identification of new information about a model of interest.

In this paper, we have discussed some of our initial work using CodeSurfer and how its ability to generate model representations directly from source was used to correct a manually-produced harbor model dependence graph published in [8], hence correcting erroneous model documentation and thus enhancing model understanding. We have also suggested one way CodeSurfer could be extended to be useful to others and enhance their understanding of their models.



(a) The original dependence graph



(b) The corrected dependence graph

## REFERENCES

- [1] P. ANDERSON, T. REPS, AND T. TEITELBAUM, *Design and implementation of a fine-grained software inspection tool*, IEEE Trans. Software Engineering, 29 (2003), pp. 721–733.
- [2] P. ANDERSON, T. W. REPS, T. TEITELBAUM, AND M. ZARNIS, *Tool support for fine-grained software inspection*, IEEE Software, 20 (2003), pp. 42–50.
- [3] R. BAECKER, *Enhancing program readability and comprehensibility with tools for program visualization*, in Proceedings of the 10th International Conference on Software Engineering, April 1988, pp. 356–366.
- [4] J. N. BUXTON AND J. G. LASKI, *Control and simulation language*, Comput. J, 5 (1963), pp. 194–199.
- [5] J. FERRANTE, K. J. OTTENSTEIN, AND J. D. WARREN, *The program dependence graph and its use in optimization*, Trans. Programming Languages and Systems, 9 (1987), pp. 319–349.
- [6] S. HORWITZ, T. REPS, AND D. BINKLEY, *Interprocedural slicing using dependence graphs*, Trans. Programming Languages and Systems, 12 (1990), pp. 26–60.
- [7] T. JONES, *Diagrammatic presentation of software engineering documents*, Tech. Rep. 95-6, Software Verification Research Centre, Department of Computer Science, The University of Queensland, Queensland, Australia, February 1995.
- [8] R. E. NANCE, C. M. OVERSTREET, AND E. H. PAGE, *Redundancy in model specifications for discrete event simulation*, ACM Trans. Model. Comput. Simul., 9 (1999), pp. 254–281.
- [9] C. M. OVERSTREET AND I. B. LEVINSTEIN, *Enhancing understanding of model behavior through collaborative interactions*. Operational Research Society (UK) Simulation Study Group 2nd Two Day Workshop, March 2004.
- [10] R. J. PAUL, T. ELDABI, J. KULJIS, AND S. J. E. TAYLOR, *Is problem solving, or simulation model solving, mission critical?*, in Proceedings of the 2005 Winter Simulation Conference, M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, eds., 2005, pp. 547–554.
- [11] T. J. SCHRIBER, *An Introduction to Simulation Using GPSS/H*, John Wiley & Sons, Inc., New York, NY, 1974.
- [12] L. SCHRUBEN, *Simulation modeling with event graphs*, Comm. ACM, 26 (1983), pp. 957–963.
- [13] M. WEISER, *Program slicing*, IEEE Trans. Softw. Eng., SE-10 (1984), pp. 352–357.