Proceedings of the 2015 Winter Simulation Conference
L. Yilmaz, W. K. V. Chan, I. Moon, T. M. K. Roeder, C. Macal, and M. D. Rossetti, eds.

ENHANCING UNDERSTANDING OF DISCRETE EVENT SIMULATION MODELS THROUGH ANALYSIS

Kara A. Olson
C. Michael Overstreet

Department of Computer Science
Old Dominion University
Norfolk, VA 23519, USA

ABSTRACT

This work extends current research in model analysis and program understanding to assist modelers in obtaining additional insight into their models and the systems they represent. Given a particular simulation implementation, this research demonstrates the feasibility of automatically-derived observations that could potentially enhance a model builder or model user's understanding of their models. One significant point of this research is that the newly-created tools do not necessitate that a modeler be able to encode the model, modify or add code, or even have a technical background. Another key point is focus on *model* aspects rather than *simulation* aspects: the model itself is detailed rather than the simulation implementation code. Results indicate these tools and techniques, when applied to even modest simulation models, can reveal aspects not previously apparent to builders or users of the models. This work provides modelers with additional techniques that can enhance understanding.

1 INTRODUCTION

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, Eldabi, Kuljis, and Taylor 2005), noting that "[s]imulation is usually resorted to because the problem is not well understood."

Simulation is used increasingly throughout research, development, and planning for many purposes. While model output is often the primary interest, insights into the system gained through the simulation process can also be valuable. These insights can come from building and validating the model as well as analyzing its behaviors and output; however, much that could be informative may not be easily discernible through these traditional approaches, particularly as models continue to increase in complexity.

A prime problem with model descriptions, whether in textual or graphical notations, is that even in simple models, embedded descriptions are often difficult to fully comprehend. Source code involves many issues unrelated to the model itself, such as data collection, animation, and tricks for efficient run-time behavior; coupled with difficulties in programming language, model details can be particularly opaque to most modelers.

For some systems and the models that represent them, recognizing interactions among components provides useful information about the systems. Often these interactions occur indirectly and usually with time delays between cause and effect. These interactions might not be easily noticed when observing animations of the simulations and are often not captured by the data typically collected and reported at the conclusion of simulations. Understanding the reasons for behaviors is an often unstated goal of simulation activities. This additional insight may also reveal modeling errors and implementation errors (the implemented model is inconsistent with the conceptual model), though these are not a focus of this research.

Insights can arise from many sources. One can be surprised to discover relationships among seemingly unrelated events. One can gain insight when something that is expected to happen does not occur, or when something that is not expected to happen does occur. Sometimes events can happen with regularity or in groupings that 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, Overstreet, and Page 1999). Anecdotal reports from modelers support the frequent difficulty of detecting important aspects of their models which when pointed out are quite useful.

This work describes research to better inform modelers about their models, extending current work in model analysis and program understanding with the goal of assisting modelers in obtaining more insight into their models and the systems they represent. To explore these potentials, we use a simulation implementation that facilitates and enables the use of several beneficial analysis techniques. A primary technique for model understanding is analysis of model output; this research has developed new, complementary techniques. Some of these techniques are known but have not been applied to modeling issues in the simulation community.

Results indicate these code analysis techniques, when applied to even modest simulation models, can reveal aspects of those models not readily apparent to the builders or users of the models. These analyses can often reveal important aspects of systems that are not readily observable in model-driven animations or even in examining data produced during simulation execution. This work has provided both model builders and model users with additional tools that can give them improved understanding of their models.

2 SOLUTION FRAMEWORK

It is often 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 a simulation – for example, the details of adding events to lists. A modeler, as defined here, is the curator of the model; s/he may or may not be the programmer who realizes the model into the computer, and may or may not have programming expertise. A model user is one who uses the simulation to meet some objective, perhaps such as trying to better understand the system at hand, designing, training, or evaluating different scenarios.

One significant point of this research is that the created tools do not necessitate that a modeler or model user be able to encode the model or have any coding expertise, but simply supply the original model definition file and execute a command. Some of the information presented here could be produced by existing software development tools but many modelers today do not have the technical background to use these tools or to make use of the reports such tools can produce.

These newly-created tools – detailed in Section 3 – can help modelers, model builders, and model users better understand their models by showing what causes what as well as producing concise summaries of key model structures that allow modelers to direct their attention to aspects not generally discernible from current simulation output.

2.1 The Condition Specification

Different ways of describing a model lend themselves more easily to different types of analyses. The condition specification is a way of describing a model that lends itself to and is the basis for many model analyses (Nance and Overstreet 1987, Page and Nance 1999). It was created to facilitate automated transformation among the classical world views of event scheduling, activity scanning, and process interaction. Serendipitously, supporting these transformations requires a representation that also enables several forms of useful diagnostic and informative analysis. The diagnostic capabilities of the condition specification are detailed in (Overstreet, Page, and Nance 1994, Nance, Overstreet, and Page 1999); an overview of its structure is described herein.

In a condition specification, a model consists of a set of objects. The state of each object is captured in a set of object attributes. Model execution consists of a sequence of changes to object attributes. While a complete condition specification has several components, the transition specification is of immediate

interest here. A transition specification describes what triggers attribute changes and how new values for them are assigned. The triggers are called conditions and the changes are called actions. A collection of model actions that must always occur atomically is called an action cluster (AC).

For example, consider the classical traveling repairman problem (described below). Examples of objects would be the repairman and facilities. One object attribute for the repairman might include whether he is busy or idle. An example of a transition in a transition specification might be:

> Condition: the repairman arrives at a facility in need of repair
> Action cluster: begin_repair: set repairman status to busy; schedule end_repair.

There are different types of conditions. Those that only depend on the value of simulation time are called time-based, or alarms. Those that depend on object attributes not including simulation time (e.g., based on conditions) are called state-based.

Each transition specification must have an initialization action cluster and a termination action cluster. The initialization action cluster occurs exactly once, at start-up. It may schedule one or more alarms for future times or it may change the values of object attributes so that some condition becomes true. The simulation proceeds accordingly with actions causing varying conditions to become true, either in the same instant as the action occurrence or at a future simulation time using alarms.

2.2 Action Clusters, Interaction Graphs

As stated above, an action cluster is a collection of model actions that must always occur atomically. Continuing the traveling repairman example, whenever begin_repair occurs, setting the repairman status to busy and scheduling end_repair occur as an indivisible unit.

Action clusters can be studied to create action cluster interaction graphs (ACIGs) (Nance, Overstreet, and Page 1999). The main purpose of this type of graph, derived from source code, is to show which events can cause which events. 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 can help us build mental models (Glenberg and Langston 1992) – in this context, a mental model of the encoded model.

In an action cluster interaction graph, nodes represent action clusters (events) and directed edges represent the ability of one action cluster to directly cause the occurrence of another action cluster – that is, an edge leads from AC 1 to AC 2 if the actions of AC 1 can cause the condition of AC 2 to become true either at the same instant as AC 1 or at a future instant (through scheduling an alarm). If an action cluster can schedule an action cluster, that is represented by a dashed line; if an action cluster could trigger an action cluster at the same simulation time, that is represented by a solid line.

3 TOOLS FOR ENHANCING UNDERSTANDING

Observing and analyzing the behaviors produced by a simulation are the usual techniques for improving understanding of a system being simulated. Different kinds of approaches yield different potential discoveries. Some analyses can tell the modeler about the model; others can uncover potential errors in the model (coding or otherwise).

Both static and dynamic analysis techniques offer different and complementary insights; the newly-created tools discussed below use both static and dynamic techniques. These techniques have a long history of use in the computer science community and software engineering community. Code optimization, automated generation of some types of documentation, checking that an implementation conforms to a design, and reverse engineering all use a combination of these techniques, as does this research.

3.1 Tool(s) Overview

A tool/suite of tools has been newly created to address these needs. There are seven functionalities:

- Creation of an extensive simulation log
- Addition of trip lines
- Identification of scheduled and triggered events
- Creation of event summaries
- Creation of the static action cluster interaction graph
- Creation of the tallied dynamic action cluster interaction graph
- Creation of a dynamic action cluster interaction graph flip book.

The original simulation output is never modified or supplemented; separate files are created by the tools. Each functionality is discussed below.

To run the tools, a single Python command is executed; alternatively, each functionality can be executed individually, also (each) with a single command. With the exception of trip lines (one line of code per request), no programming or any modification is needed. The tools are not yet general purpose such that they can be used with any given simulation written in any programming or simulation language; however, they allow exploration of the feasibility of analysis to supplement what many simulations produce.

3.2 Example: Traveling Repairman

A simulation of the classical traveling repairman model is used to demonstrate the created tools. In the model (Cox and Smith 1961), a repairman tends to a number of machines which fail over time and need repair. This model can be used to study how many machines or repairmen are needed, effects of machine modifications, and production rates.

3.3 Simulation Log

Using dynamic analysis, an extensive simulation log is generated that notes each action and the simulation time.

Many simulations are programmed to print final usage statistics, utilization, etc.; however, this log is generated without any user action or programming effort (such as including output print statements).

While the log can be examined by the modeler and is informative in its own right, it is used by most of the other tools to present summary information. For a typical simulation, the size of the log makes noticing issues of interest more difficult.

The created simulation log is parsed to create a screen- and printer-friendly version of the log as well.

In the example below, the original simulation output included the execution frequency of each action cluster and run statistics such as utilization and repairman travel time.

Original, complete, unmodified simulation output:

```
Run 0
Frequency count of AC executions
  AC procId   Frequency
    0              1
    1              1
    2           2001
    3           2000
    4           2000
    5           1351
```

```
    6           1351
    7           2000
Termination!  System time:    73612.47
Repairman utilization:           39.72
Repairman total work time:    16203.31
Repairman total travel time:  13035.50
Number of repairs:                2000
```

Part of the newly- and automatically-generated log:

```
Run 0
time:     description
0.000000: initialization
0.000000: initialization scheduled failure for time 0.004378
⋮
0.004378: failure
0.004378: (repairman.status == IDLE && SomeFailed()) triggered
          travel_to_facility
0.004378: travel_to_facility
0.004378: travel_to_facility scheduled begin_repair for time 3.504378
3.504378: begin_repair
3.504378: begin_repair scheduled end_repair for time 8.913395
8.913395: end_repair
8.913395: end_repair scheduled failure for time 251.966998
8.913395: (mrp.num_failed_facilities == 0 && repairman.status == IDLE
          && repairman.location != idle_loc) triggered travel_to_idle
⋮
73612.466229: (repairman.num_repairs >= mrp.max_repairs) triggered
              termination
73612.466229: termination
```

By examining this simulation log, a modeler might learn that the simulation starts by scheduling the first machine failures, or that the repairman traveled to the idle location only a few times, observations that are not readily apparent from the simulation output.

3.4 Trip Lines

A "trip line" concerns any boolean expression of model variables of which the modeler wants to be notified the first time it is passed – for example, if a queue length becomes greater than 10 or a wait time becomes greater than one hour. This could also be used to note other user-defined criteria.

The modeler can add a special line (or lines) anywhere in the simulation code indicating what variable(s) and/or condition(s) s/he would like to be noted. Two options are available: `trip_when(`*condition*`)` or `trip_when(`*condition*`,` *reset_condition*`)`. In the first case, if the condition becomes true, that is noted in the simulation log; this trip line can be tripped exactly once. In the second case, if the condition becomes true, that is noted in the simulation log; if the reset condition becomes true, that is also noted in the simulation log, the trip line is reset and can be tripped again.

Trip lines are an optional addition of a single, simple line of code per request that requires no knowledge of output, output formatting, or finding everywhere a change might occur, and allows a modeler to easily

check whether situations that may be of interest actually occur.

From the simulation code (programmer added the `trip_when`):

```
/* 10,000 hours to become an expert myth */
trip_when(repairman.work_time >= 10000.0);
```

From the automatically-generated simulation log:

```
⋮
44072.318287: begin_repair
44072.318287: begin_repair scheduled end_repair for time 44097.771914
44072.318287: ! trip line tripped: (repairman.work_time >= 10000.0)
⋮
```

3.5 Scheduled and Triggered Events

While many code coverage tools can aid programmers in detecting unexecuted components, a significant point of this research is to assist modelers and model users that may not be interested or comfortable in learning to use or exploit such tools. In addition, the results of such tools often include much that is not pertinent to the model itself, but rather its implementation – not likely to be of interest to the modeler and worse, might obfuscate information that is of interest. Combining this with interest in *model* analysis rather than *simulation* analysis yields a tool that creates a list of all scheduled, unscheduled, triggered, and untriggered events.

This list can be informative by possibly identifying unanticipated effects previously unrecognized by the modeler. They can also serve a diagnostic purpose if a list omits events the modeler knows should be included, or includes events the modeler knows should not be included.

From the tool:

```
during the simulation run:

scheduled events:
    arrive_idle
    begin_repair
    end_repair
    failure

unscheduled events:
    termination
    travel_to_facility
    travel_to_idle

triggered action events:
    termination
    travel_to_facility
    travel_to_idle

untriggered events:
```

```
    ~ none ~
```

In this case, for example, if travel_to_idle did not occur, it could indicate a busy repairman, a coding error, or a modeling error.

3.6 Event Summaries

In a simulation, different types of statistics can be of interest: some are general – for example, how often an event occurs; some are model-specific – for example, how often a particular machine is in use; and still others are implementation-specific – for example, how often a condition queue is empty.

Using dynamic analysis, total simulation time and a summary with respect to each event are tallied and presented for each simulation run. For each event in the run, its number of occurrences, events scheduled, number of times scheduled, events triggered, and number of times triggered are presented.

In a prior local simulation study, a modeler was studying trace data produced during simulation executions. It happened to be noticed 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 – a fundamental insight revealable through these created tools.

From the tool:

```
Run 0

Total simulation time: 73612.466229

Events:
    initialization
        occurrences: 1
        events scheduled:
            failure: 12 times
        events triggered:
             ~ none ~
    begin_repair
        occurrences: 2000
        events scheduled:
            end_repair: 2000 times
        events triggered:
             ~ none ~
    end_repair
        occurrences: 2000
        events scheduled:
            failure: 2000 times
        events triggered:
            termination: 1 time
            travel_to_facility: 648 times
            travel_to_idle: 1351 times
⋮
```

3.7 Static Action Cluster Interaction Graph

As discussed in Section 2.2, the main purpose of this type of graph is to show which events can cause which events.

The action cluster interaction graph is automatically generated by the tool (Figure 1, below). This also provides potentially useful model documentation.

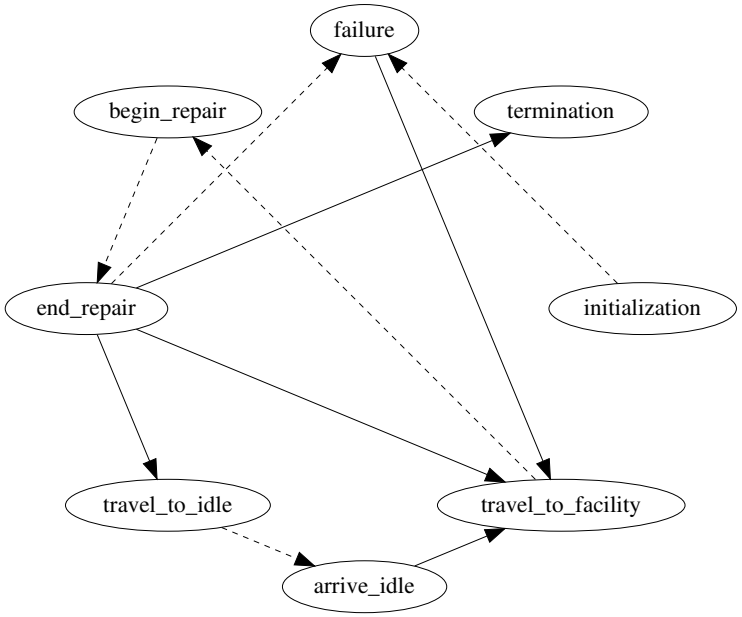

Figure 1: Generated graph: traveling repairman static action cluster interaction graph

3.8 Tallied Dynamic Action Cluster Interaction Graph

Using both static and dynamic analysis, the action cluster interaction graph is automatically generated, with edges labeled according to event frequency during a given run (Figure 2, below).

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 combination, if static analysis suggests that event A can cause event B, but dynamic analysis reveals this is not observed, this may be of interest.

Given an arbitrary condition specification, the automatic creation of a static ACIG with a minimum number of edges is unsolvable (Nance, Overstreet, and Page 1999); these superfluous edges are misleading as they suggest a false causal relationship between events. Edges labeled "0" in the tallied dynamic ACIG can guide modelers to consider if these edges are superfluous – perhaps through additional, non-automated analysis of the model – or if this interaction just did not occur during this particular run.

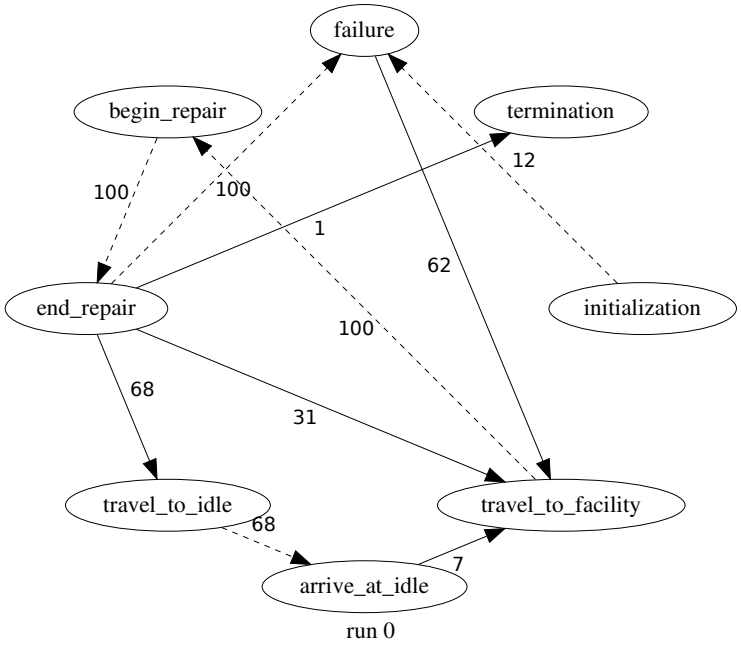

Figure 2: Generated graph: traveling repairman tallied dynamic action cluster interaction graph

3.9 Dynamic Action Cluster Interaction Graph Flip Book

A dynamic ACIG is created for every time step of the simulation run; these are then combined into a multi-page document that can be flipped through. This provides an initial visual representation of the entire simulation run and a basis for future animation.

Being able to study specific run interactions in a clear, visual way contributes additional possibilities for insight that are not as easily discernible from text-based output. The flip book provides the ability for a modeler to focus on particular periods of time or particular event sequences.

4 EVALUATION

The crux of this research was to create and present automatically-derived observations that could potentially enhance a modeler or model user's understanding, that would not necessitate that s/he have programming expertise or even a technical background. As modeling and simulation continues to be used increasingly often in research and as models continue to increase in complexity, these types of analyses and tools will continue to be an important contribution.

Another significant aspect of this is the focus on *model* aspects rather than *simulation* aspects. Source code tends to involve many issues unrelated to the model itself, such as data collection, animation, and tricks for efficient run-time behavior. Even when the modeler is an expert programmer, this other code often can obscure features of the model as implemented.

4.1 Simulation Log

Many simulations are programmed to generate execution traces, final usage statistics, perhaps animations, etc.; however, the first contribution of this log is its generation with no user action or programming effort. The amount and kinds of detail provided in the log are a helpful contribution in and of themselves –

possibly providing a modeler with additional insights into the behavior of the model they have created or are using – as well as an additional resource for the other created tools.

For example, consider the simulation output of the repairman model in the first part of Section 3.3: as with most simulations, it is designed to answer a few specific questions based on modeling objectives. However, studying the simulation log (second part of Section 3.3), one can see that immediately after the failure at time 0.004378, `(repairman.status == IDLE && SomeFailed())` triggered travel_to_facility, information that is not available otherwise.

Similarly, at the end of the simulation, why did the simulation terminate? Considering the simulation log, one can determine that the maximum number of repairs was reached: `(repairman.num_repairs >= mrp.max_repairs)` triggered termination.

Additionally, for someone with expertise, the simulation log is more readily searchable with regular expressions than most standard simulation output.

4.2 Trip Lines

Trip lines are an optional addition of a single, simple line of code per request that requires no knowledge of output, output formatting, or finding everywhere a change might occur, allowing a modeler to easily check whether situations that may be of interest actually occur.

The benefits of using a trip line over, say, general if statements include not having to find everywhere a variable or variables are changed and hence the condition might change; clarity and integration with the simulation log; and the aforementioned non-requirement of programming expertise. Trip lines can be simply configured to trip exactly once or tripped and reset, neither of which can be accomplished with only a (set of) print statement(s).

The availability of trip lines can increase understanding of the embedded model and more importantly, the system it represents.

4.3 Scheduled and Triggered Events

The scheduled and triggered event lists can be informative by possibly identifying unanticipated effects previously unrecognized by the modeler. They can also serve a diagnostic purpose if a list omits events the modeler knows should be included, or includes events the modeler knows should not be included. While existing software tools, such as `gcov`, can be used to provide some similar information, these lists omit much from `gcov`-like tools that is unlikely to be of interest to a modeler and instead focus on *model* behavior.

Similarly, consider Section 3.5, scheduled and triggered events with respect to the traveling repairman: one can note that termination can be either scheduled or triggered. Perhaps in the batch of runs under consideration, the simulation always terminates after a certain number of repairs (`(repairman.num_repairs >= mrp.max_repairs) triggered termination`, from the simulation log in Section 3.3) – that is, termination is always triggered. However, it could be insightful to know that the simulation also could be scheduled to end (perhaps the machines fail less often, causing the repairman to make fewer repairs) – something not necessarily discernible from any arbitrary batch of runs but now obvious.

4.4 Event Summaries

As discussed in Section 3.6, having concise, useful summary information about model components has already revealed model structure in some models that had not been previously recognized.

4.5 Static Action Cluster Interaction Graph

Being able to follow how model components can interact is a significant part of understanding the model itself. The static ACIG presents these possible interactions in a clear, visual way that is not easily discernible from usual simulation output. This additional information is unlikely to be detected by executing the simulations and contributes to the insights gained by modeling a complex system. These graphs have been discussed previously (e.g., (Nance, Overstreet, and Page 1999)) but were created manually; though concerning relatively simple models and multiply-reviewed, many of these manually created graphs were found later to have errors. This is another compelling argument for such automation.

Additionally, while not exemplified here, some analyses can be based on visual inspection of the static ACIG that are not easily noticed otherwise. For example, the only event that has no successors is termination. If visual inspection reveals that another event has no possible successors, this may warrant additional consideration: it may be included in anticipation of future development or a result of code reuse; or could indicate an error in either coding or specification rather than a characteristic of the system represented.

4.6 Tallied Dynamic Action Cluster Interaction Graph

Being able to follow how model components interact during a particular simulation run can also enhance model understanding. The tallied dynamic ACIG combines the insights of the static ACIG with those of each event summary during the simulation run, again, in a clear, visual way.

Not obvious from only the text-based event summaries or the static ACIG, though, is the possibility of superfluous edges and which edges may be such; these edges are misleading as they suggest a false causal relationship between events. Edges labeled "0" in the tallied dynamic ACIG can guide modelers to consider if these edges are extraneous or if this interaction just did not occur during this particular run, enhancing understanding of the model and the system it represents.

4.7 Dynamic Action Cluster Interaction Graph Flip Book

The dynamic ACIG flip book provides a visual representation of the entire simulation run. Again, being able to study specific run interactions in a clear, visual way contributes additional possibilities for insight that are not as easily discernible from text-based output. The flip book provides a first cut at animating this output and the ability for a modeler to focus on particular periods of time or particular event sequences.

For example, consider the combination of the flip book with trip lines: when a line is tripped, exploring the flip book can give a clear picture of the preceding events. Often, animations in and of themselves are not particularly useful if they are without navigation tools to enable exploration: dealing with the wealth of data available (graphically or otherwise) can often overwhelm and obscure useful information. Being able to choose a particular time or event – say, when a trip line tripped – and being able to consider specifically the surrounding simulation events can contribute to better understanding.

5 SUMMARY

The automated analysis of model specifications is an area that historically has received little attention in the simulation research community but which can offer significant benefits. This is particularly true for analysis intended to provide modelers and model users additional information about their models. A usual goal in simulation is enhanced understanding of a system; model analysis can provide insights not otherwise available. This work developed new approaches for the simulation community to complement current methods used to gain insights into models, their behaviors, and the systems they represent.

The contribution of this research is the demonstration of the feasibility of automatically-derived observations that could potentially enhance a modeler or model user's understanding, that does not necessitate that the modeler have programming expertise or even a technical background. While some of the information

presented here could be produced by existing software development tools, many modelers today do not have the technical background to use these tools or to make use of the reports such tools can produce. Another key point of this work is the focus on *model* aspects rather than *simulation* aspects. As modeling and simulation continues to be used increasingly often in research and as models continue to increase in complexity, these types of tools will become increasingly helpful.

Automatic tools have been newly-created and demonstrated to reveal new insights into models and the systems they represent. These tools include an extensive simulation log; trip lines; scheduled and triggered events; event summaries; static action cluster interaction graph; tallied dynamic action cluster interaction graph; and dynamic action cluster interaction graph flip book. This work demonstrates that using a particular simulation implementation, the automatic generation of new information is feasible.

REFERENCES

Cox, D. R., and W. L. Smith. 1961. *Queues*. London: Methuen & Co.

Glenberg, A. M., and W. E. Langston. 1992, April. "Comprehension of Illustrated Text: Pictures Help to Build Mental Models". *J. Mem. Lang.* 31 (2): 129–151.

Nance, R. E., and C. M. Overstreet. 1987, December. "Exploring the Forms of a Model Diagnosis in a Simulation Support Environment". In *Proceedings of the 1987 Winter Simulation Conference*, edited by A. Thesen, H. Grant, and W. D. Kelton, 590–596.

Nance, R. E., C. M. Overstreet, and E. H. Page. 1999, July. "Redundancy in Model Specifications for Discrete Event Simulation". *ACM Trans. Model. Comput. Simul.* 9 (3): 254–281.

Overstreet, C. M., and I. B. Levinstein. 2004, March. "Enhancing Understanding of Model Behavior Through Collaborative Interactions". In *Operational Research Society (UK) Simulation Study Group Two Day Workshop Proceedings*, edited by S. C. Brailsford, L. Oakshott, S. Robinson, and S. J. E. Taylor, 11–17.

Overstreet, C. M., E. H. Page, and R. E. Nance. 1994, December. "Model Diagnosis using the Condition Specification: From Conceptualization to Implementation". In *Proceedings of the 1994 Winter Simulation Conference*, edited by J. D. Tew, M. S. Manivannan, D. A. Sadowski, and A. F. Seila, 566–573.

Page, E. H., and R. E. Nance. 1999, June. "Incorporating Support for Model Execution Within the Condition Specification". *Trans. Soc. Comput. Simul. Int.* 16 (2): 47–62.

Paul, R. J., T. Eldabi, J. Kuljis, and S. J. E. Taylor. 2005, December. "Is Problem Solving, or Simulation Model Solving, Mission Critical?". In *Proceedings of the 2005 Winter Simulation Conference*, edited by M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, 547–554.

AUTHOR BIOGRAPHIES

KARA A. OLSON recently completed her doctorate in Computer Science at Old Dominion University. She has presented her work by invitation in Germany as well as in England, Canada, the Netherlands, and the United States. She holds a Master of Science, Computer Science, Bachelor of Computer Science, and Bachelor of Science, Mathematics from Old Dominion. She is a member of ACM and IEEE/CS. Her email address is kara@cs.odu.edu.

C. MICHAEL OVERSTREET is an Emeritus 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 over 80 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 e-mail address is cmo@cs.odu.edu.