Winter 2014

Enhancing Understanding of Discrete Event Simulation Models Through Analysis

Kara Ann Olson
Old Dominion University

ENHANCING UNDERSTANDING OF DISCRETE EVENT SIMULATION MODELS THROUGH ANALYSIS

by

Kara Ann Olson
B.S.C.S. May 1997, Old Dominion University
B.S. May 1997, Old Dominion University
M.S. May 2007, Old Dominion University

A Dissertation Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

COMPUTER SCIENCE

OLD DOMINION UNIVERSITY
December 2014

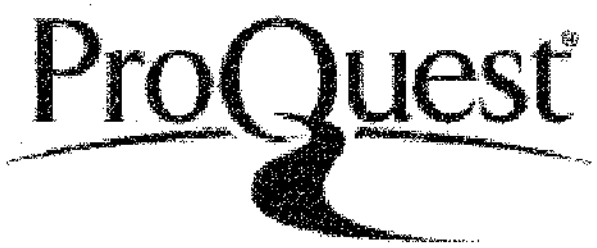
Approved by:

C. Michael Overstreet (Director)

Steven J. Zeil (Member)

Irwin B. Levinstein (Member)

Roland R. Mielke (Member)



ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

ABSTRACT

ENHANCING UNDERSTANDING OF DISCRETE EVENT SIMULATION MODELS THROUGH ANALYSIS

Kara Ann Olson
Old Dominion University, 2014
Director: Dr. C. Michael Overstreet

Simulation is used increasingly throughout research, development, and planning for many purposes. While model output is often the primary interest, insights gained through the simulation process can also be valuable. 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 existing traditional approaches, particularly as models continue to increase in complexity.

This research extends current work in model analysis and program understanding to assist modelers in obtaining more insight into their models and the systems they represent. A primary technique for model understanding is analysis of model output; this research has developed new, complementary techniques.

A 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. Some of the information presented here could be produced by existing software development tools; however, most modelers today do not have the technical background to use such tools or to make use of the reports they can produce.

Additionally, one of the significant details of this research is the focus on *model* aspects rather than *simulation* aspects: the tools developed here detail the model embedded in implementation code, not the code necessary for implementation. 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.

Results indicate these tools and 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. This work provides both model builders and model users with additional techniques that can give them improved understanding of their models.

To $H

TABLE OF CONTENTS

Page

LIST OF FIGURES ix

Chapter

1. PROBLEM DEFINITION, MOTIVATION 1

2. PRIOR RESEARCH 3
   - 2.1 COMPILER/OPTIMIZATION TECHNIQUES 3
   - 2.2 MODELING & SIMULATION 3
   - 2.3 PROGRAM VISUALIZATION 9
   - 2.4 FEATURE LOCATION 10
   - 2.5 PROGRAM UNDERSTANDING 11

3. SOLUTION MOTIVATION, FRAMEWORK 12
   - 3.1 THE CONDITION SPECIFICATION 12
   - 3.2 ACTION CLUSTERS, INTERACTION GRAPHS 14
   - 3.3 DIRECT EXECUTION OF ACTION CLUSTERS 14
   - 3.4 THREE MODELS 15

4. TOOLS FOR ENHANCING UNDERSTANDING 17
   - 4.1 LIMITS OF ANALYSIS 18
   - 4.2 TOOL(S) OVERVIEW 18
   - 4.3 SIMULATION LOG 19
   - 4.4 TRIP LINES 27
   - 4.5 SCHEDULED AND TRIGGERED EVENTS 33
   - 4.6 EVENT SUMMARIES 36
   - 4.7 STATIC ACTION CLUSTER INTERACTION GRAPH 42
   - 4.8 TALLIED DYNAMIC ACTION CLUSTER INTERACTION GRAPH 47
   - 4.9 DYNAMIC ACTION CLUSTER INTERACTION GRAPH FLIP BOOK 52

5. EVALUATION 78
   - 5.1 SIMULATION LOG 78
   - 5.2 TRIP LINES 80
   - 5.3 SCHEDULED AND TRIGGERED EVENTS 81
   - 5.4 EVENT SUMMARIES 81
   - 5.5 STATIC ACTION CLUSTER INTERACTION GRAPH 82
   - 5.6 TALLIED DYNAMIC ACTION CLUSTER INTERACTION GRAPH 82
   - 5.7 DYNAMIC ACTION CLUSTER INTERACTION GRAPH FLIP BOOK 83

6. FUTURE RESEARCH DIRECTIONS 84
6.1 CONDITION SPECIFICATION TO DIRECTION EXECUTION OF ACTION CLUSTERS 84
6.2 USE IN DETERMINING APPROPRIATENESS OF MODELS 84
6.3 IDENTIFICATION OF POSSIBLE RACE CONDITIONS 84
6.4 ADDITIONAL GRAPHICS 85

7. SUMMARY 86

REFERENCES 89

VITA 94

LIST OF FIGURES

Figure Page

1. A GPSS model segment [36] 5
2. An Arena model [41] 6
3. A Simio model [40] 7
4. A sample Petri net [42] 8
5. A sample event graph [37] 8
6. Component interactions of the tool 19
7. Generated graph: traveling repairman static action cluster interaction graph 44
8. Generated graph: harbor static action cluster interaction graph 46
9. Generated graph: single server queue static action cluster interaction graph 47
10. Generated graph: traveling repairman tallied dynamic action cluster interaction graph 50
11. Generated graph: harbor tallied dynamic action cluster interaction graph 51
12. Generated graph: single server queue tallied dynamic action cluster interaction graph 52
13. Generated graph: traveling repairman dynamic action cluster interaction graph flip book, page 1 of 369 54
14. Generated graph: traveling repairman dynamic action cluster interaction graph flip book, page 2 of 369 56
15. Generated graph: traveling repairman dynamic action cluster interaction graph flip book, page 3 of 369 57
16. Generated graph: traveling repairman dynamic action cluster interaction graph flip book, page 4 of 369 59
17. Generated graph: traveling repairman dynamic action cluster interaction graph flip book, page 5 of 369 60

18. Generated graph: traveling repairman dynamic action cluster interaction graph flip book, page 368 of 369 62

19. Generated graph: traveling repairman dynamic action cluster interaction graph flip book, page 369 of 369 64

20. Generated graph: harbor dynamic action cluster interaction graph flip book, page 1 of 212 65

21. Generated graph: harbor dynamic action cluster interaction graph flip book, page 2 of 212 66

22. Generated graph: harbor dynamic action cluster interaction graph flip book, page 3 of 212 67

23. Generated graph: harbor dynamic action cluster interaction graph flip book, page 4 of 212 68

24. Generated graph: harbor dynamic action cluster interaction graph flip book, page 5 of 212 69

25. Generated graph: harbor dynamic action cluster interaction graph flip book, page 6 of 212 70

26. Generated graph: harbor dynamic action cluster interaction graph flip book, page 7 of 212 71

27. Generated graph: harbor dynamic action cluster interaction graph flip book, page 211 of 212 72

28. Generated graph: harbor dynamic action cluster interaction graph flip book, page 212 of 212 73

29. Generated graph: single server queue dynamic action cluster interaction graph flip book, page 1 of 42 74

30. Generated graph: single server queue dynamic action cluster interaction graph flip book, page 2 of 42 74

31. Generated graph: single server queue dynamic action cluster interaction graph flip book, page 3 of 42 75

32. Generated graph: single server queue dynamic action cluster interaction graph flip book, page 4 of 42 75

33. Generated graph: single server queue dynamic action cluster interaction graph flip book, page 5 of 42 76

34. Generated graph: single server queue dynamic action cluster interaction graph flip book, page 6 of 42 76

35. Generated graph: single server queue dynamic action cluster interaction graph flip book, page 41 of 42 77

36. Generated graph: single server queue dynamic action cluster interaction graph flip book, page 42 of 42 77

CHAPTER 1

PROBLEM DEFINITION, MOTIVATION

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 [28], 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.

Researchers have long demarcated the conceptual model and the implemented model [5, 3]; indeed, the model as realized in source code is the only true specification of the model as executed. Paul and Kuljis accurately state [29]:

> 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.

Accordingly, the simulation code itself is the basis for the analyses developed here.

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 also gain insight when something that is expected to happen does not 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 [24, 23]. Anecdotal reports from modelers support the frequent difficulty of detecting important aspects of their models which when pointed out are quite useful.

This research extends current work in model analysis and program understanding to assist modelers in obtaining more insights into their models. A primary technique for model understanding is analysis of model output; this research has developed new, complementary techniques. Some of the 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 techniques that can give them improved understanding of their models.

CHAPTER 2

PRIOR RESEARCH

Several communities seem to have started in this direction, but none has quite brought these concepts all together in one place for use by the simulation community.

2.1 COMPILER/OPTIMIZATION TECHNIQUES

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 – 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, and relationships among different simulation components. Explicit identification of these interactions can help modelers identify causes and influences of system behaviors.

Control flow analysis can be used to help determine which variables control which behaviors. It is especially useful for parallelization, something being done more often due to the size and complexity of newer models (and which has its own body of research).

Program slicing [45] analyzes both the data flow and control flow of a given program to produce a reduced program that yields the same specified behaviors at a given point in time. Slices of model code can reveal causal chains of model behaviors. Many types of slicing have since been developed, including backward, forward, dynamic, static, and quasi-static, among others. A recent survey of these and additional slicing techniques can be found in [35].

2.2 MODELING & SIMULATION

Simulation has a long history of trying to visualize models, as "graphical models usually provide a better understanding of conceptual models with less effort than the other types of representation" [34]. Most graphical modeling languages and their corresponding analysis techniques are motivated by and have emphasis on making

the model easier to build and mistakes easier to identify. Our objectives are similar: again, to help modelers and model users better understand the models they are creating or using. While there is overlap in these approaches, the goals and techniques used can differ.

Some of the goals of this research were identified early on by the simulation community. GPSS – General Purpose Simulation System – was among the first simulation programming languages. While GPSS has an Assembly-like syntax, it models a system as a block diagram, not unlike an extended, specialized flow chart [Figure 1]. It was intended as a graphical representation for simulation. Each action in GPSS has an iconic graphic and its inventor, Gordon, intended that modelers would construct their models on paper using these graphics, "making the simulation directly accessible to system analysts rather than through programmers" [13]. Gordon aptly noted some of the same arguments for a graphical tool: "The relative ease of learning GPSS made it attractive ... in particular, to people without a technical background." "[I]f it were properly organized and documented, engineers and analysts would be able to use the program themselves, even if they were not trained in programming." "The block diagram language enhanced ... that the user was ... describing a system." "Block diagrams were also an asset in improving understanding between the various people who needed to know about the system" [13]. Several modern rapid prototyping simulation systems (such as Arena [Figure 2] and Simio [Figure 3]) take this approach as well, where programming is done using a graphical interface. While some models can be represented naturally using these methods, others require significant contortion and creativity on the part of the model programmer.

Petri nets are another graphical means to describe a discrete event simulation. Petri nets use a mathematical language to describe a simulation in terms of places, transitions and directed arcs [46] [Figure 4]. Much of the work around Petri nets concerns process analysis [46]; an excellent survey of Petri net analysis can be found in [21]. These techniques, though, require that a Petri net model be constructed: many simulations do not lend themselves to this representational form and Petri nets are not easily understood by many modelers. Timed Petri nets are an extension created in order to accommodate "real-world systems" [31]. A timed Petri net consists of a Petri net and a function that assigns a real, non-negative time to each transition in the net. No efforts have been made or analysis discussed to help modelers enhance understanding their models other than the presentation of the graphical

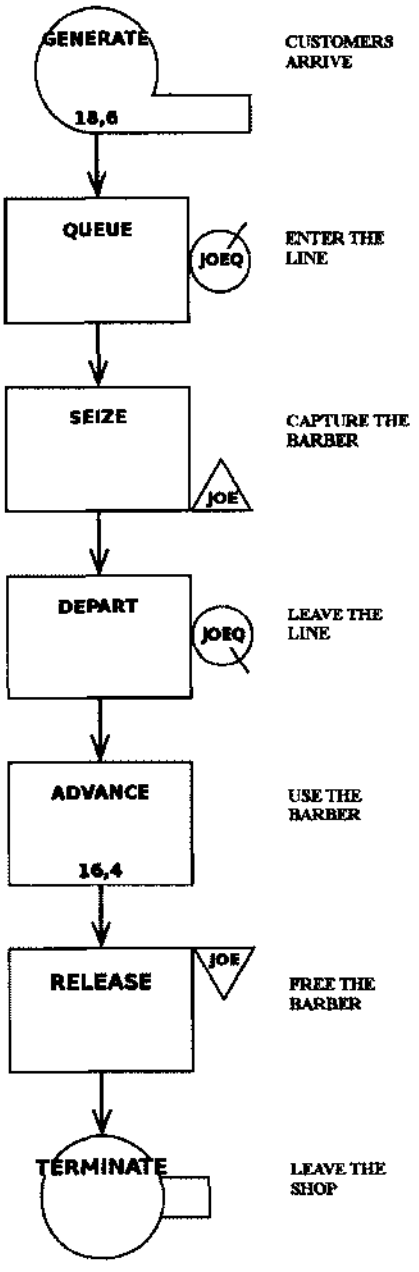


Figure 1. *A GPSS model segment [36].*

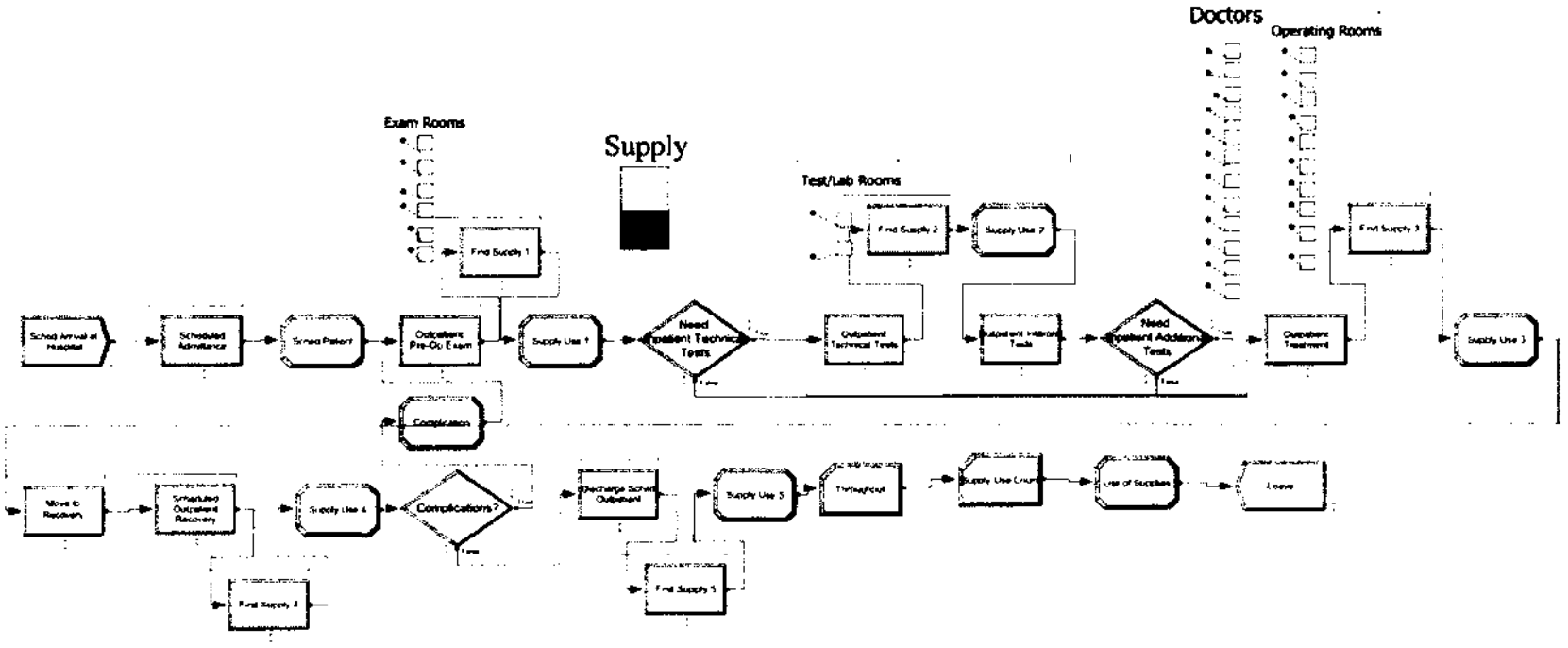


Figure 2. *An Arena model [41].*

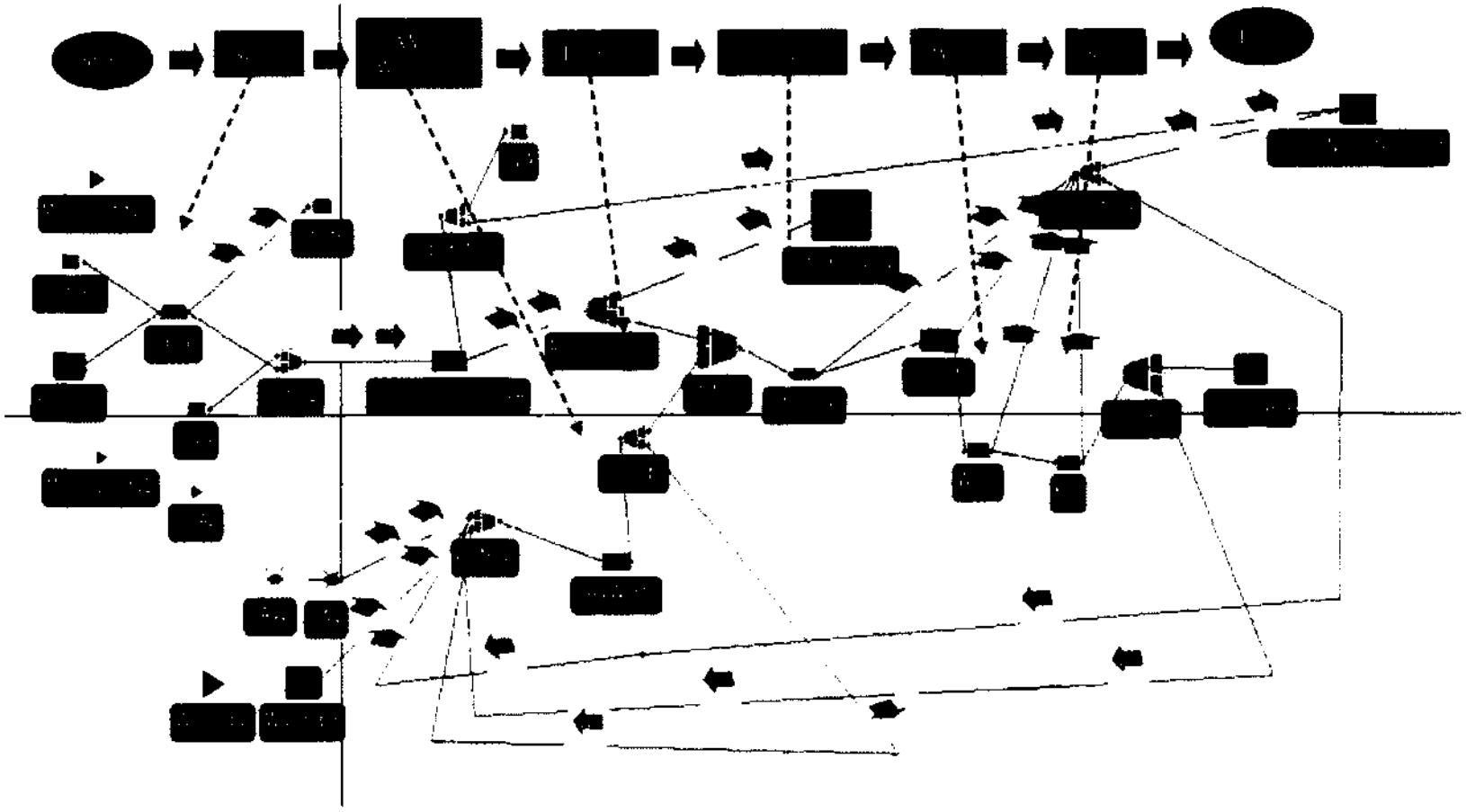


Figure 3. *A Simio model [40].*

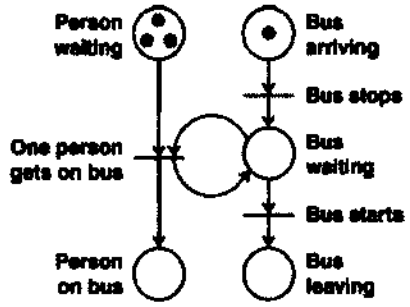


Figure 4. *A sample Petri net [42].*

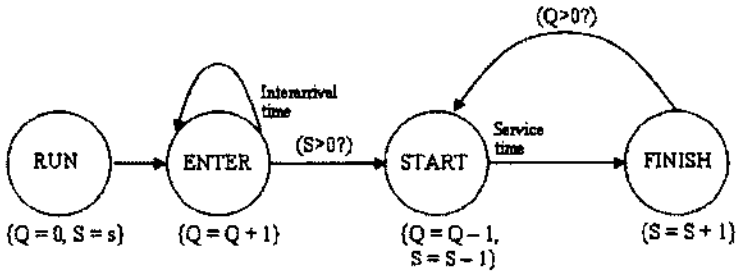


Figure 5. *A sample event graph [37].*

representation itself.

Schruben created the event graph modeling formalism since "[e]stablished graphical techniques for visualizing event-oriented structures [were] lacking" [38]. Event graphs define a simulation through specifications of the system state, event logic, and relationships between events [14] [Figure 5]. Event graphs can be used as a basis for model analysis, though as our objectives differ from Schruben's, the information in our graphs differs. Schruben also created simulation graphs, an extension of event graphs. A simulation graph is a mathematical structure that defines a simulation through a vertex set, sets of scheduling and canceling edges, and an incidence function. Schruben states that "[a]ny simulation, indeed any computer program, can be modeled using a Simulation Graph" [39]. While this may be the case, some representations are much more amenable to the kinds of analysis of interest.

Kranzlmüller also formulated a graph formalism called an event graph. This event graph is a "directed global communication graph ... partially ordered in time, [showing] the interprocess dependencies between processes" [20]. A point crucial to

our research, Kranzlmüller notes that as the number of processes increases, the visualization becomes "significantly more complicated and decreases the understanding of the user for the displayed information" [20]. Rather than using a graphical language to code a model – the result of which still becomes large and obfuscating – the created tools present selective, graphical representations of potentially useful aspects of these models.

Zeigler's Discrete Event System Specification (DEVS) formalism [48] has been the basis of significant analysis work. DEVS was heavily influenced by general systems theory and model formulations using mathematical notations. A simulation is represented through a structure consisting of sets of input and output values; a set of states; internal and external transition functions; an output function; and a set of time values [49]. Behavior of the simulation can be reasoned about mathematically, similar in notion to how one might argue about linear algebra. However, the focus of much of this work concerns model verification – for example, [15] – as well as connectivity and reachability; the intent has not been and consequently does not lend itself to support a modeler in gaining understanding about a model.

2.3 PROGRAM VISUALIZATION

Some work has been done specifically on the "visuality" of programs. 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" [2]) has similarly stated goals, in part: "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" [2]. The approach of program visualization to doing so, though, "focuses on output, on the display of programs, their code, documentation, and behavior" [2].

One such example, an information mural [18] 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 information is difficult to read and understand. A graphical view ... could better help a software developer understand what occurs during a program's execution."

Program visualization focuses primarily on code; our interest is in depicting interactions of model behaviors. While program visualization techniques are indeed beneficial, additional insights can potentially be revealed through additional techniques, especially as models continue to increase in complexity.

2.4 FEATURE LOCATION

Feature location [11] is part of the software maintenance community and deals with finding "features" or "functionalities" [47] in the code – code that causes a given behavior – usually with the goal of change, extension, or removal. Feature location work is classified by its community as a software engineering reverse engineering task. The problem "is a hard problem in software engineering because it is an inherently human activity" [32].

Software reconnaissance [47] compares traces of test runs exhibiting a feature and test runs not exhibiting the feature to determine which pieces of code are related to the feature. Wilde and Scully note that their techniques cannot be used with features that are always present in the program – that is, in the case that the program cannot be run in such a way that they are not exhibited. They reasonably conclude that their method "complements other sources of information in providing places to start looking at code" – our goal as well. Program slicing, mentioned above, can assist with both feature location and software reconnaissance.

Chen and Rajlich [8] present a case study of locating features with "computer-assisted search of [a] software dependence graph." They note, however, "extensive knowledge is required, including domain knowledge, programming knowledge, knowledge of algorithms and data structures, knowledge of the software components and their interactions, etc." If a user had that knowledge, s/he wouldn't need such tools – a sentiment stated too by Koschke and Quante [19]: "The scenario for feature location is that we do not know the system in all details – otherwise feature location would not be an issue in the first place."

Bohnet and Döllner [6] use a combination of static and dynamic analysis techniques as well as call run times and a "$2\frac{1}{2}$-dimensional" "graph visualization" technique. Their tool relies initially on the user to extract the system architecture and identify feature-executing scenarios. Similar to the aforementioned feature location approaches, this approach may be considered reasonable for someone with a computer science or programming background, but not for the many non-computer scientist

researchers using modeling and simulation in their work.

2.5 PROGRAM UNDERSTANDING

Programming understanding concerns understanding a given computer program and the relationships among its components [7]. This seems like a good match to our goals: while we are not trying to understand programs themselves per se, we are trying to understand the model as expressed in a given piece of code, as the source code is the true specification of the model as executed.

IBM's Research Division began working on tools to assist specifically with program understanding as early as 1986 [9]. The program comprehension community considers program comprehension "a vital software engineering and maintenance activity ... necessary to facilitate reuse, inspection, maintenance, reverse engineering, reengineering, migration, and extension of existing software systems" [16]. While our reasons are different, our focuses are not so.

Cognitive theories have long been established in the field of program comprehension; tool needs have been documented and researched [43]. However, no one seems to have yet applied this knowledge to the field of modeling and simulation. This leads to the crux of the research.

CHAPTER 3

SOLUTION MOTIVATION, 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 model – 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.

A 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 most modelers today do not have the technical background to use these tools or to make use of the reports such tools can produce.

These tools – detailed extensively, below – 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.

Prior research initiated by Derrick (which formed the basis of my Master's work) involved using Extensible Markup Language (XML) to create a service-oriented architecture to enable automated diagnostic techniques, and was described by Roeder and Schruben, the creator of Simulation Graphs (Section 2.2, above), as "interesting new work" [33]. Condition Specifications – described in the next section – and Simulation Graphs are among the few specification formalisms that have demonstrated promise and amenability to automated diagnostic techniques. Of the two, the condition specification (CS) was the more natural choice for the research due to the accessibility of the CS: again, the goal of this research is to explore analysis approaches to help non-programmers better understand their models.

3.1 THE CONDITION SPECIFICATION

As mentioned above, 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 [22, 27]. 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 [25, 23]; 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. Table 1 illustrates, in conceptual form, a transition specification for a CS.

Condition	**Actions**
Condition 1	Action cluster 1
Condition 2	Action cluster 2
⋮	⋮
Condition n	Action cluster n

TABLE 1. *Structure of transition specification.*

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. At (exactly) the beginning of a simulation, the special boolean condition initialization is true. Consequently, the initialization action cluster occurs only 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 value of simulation time using alarms.

3.2 ACTION CLUSTERS, INTERACTION GRAPHS

An action cluster (AC) 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). 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 [12] – 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.

3.3 DIRECT EXECUTION OF ACTION CLUSTERS

A discrete event simulation can be written in a way that embodies the condition specification, called direct execution of action clusters. This style has been described in [27] and continues to be used successfully.

One such algorithm (similar to those suggested in [27]) involves the creation of a routine for each action cluster in the condition specification. Model execution starts by executing the initialization action cluster routine and its state-based successors, if any. Thereafter, execution consists of a simple loop:

1. Update the simulation time to the next scheduled event.

2. Execute the events that have been scheduled for the current simulation time. If one of these events is the termination action cluster, it executes and the simulation terminates.

3. Scan, in turn, the condition of each action cluster. If a condition is true, execute the appropriate routine. Repeat this step until no action clusters are triggered. If one of these events is the termination action cluster, it executes and the simulation terminates.

4. Return to step 1.

To improve run-time efficiency, some have presented techniques in step 3 so that only a minimum number of conditions are scanned (e.g., [26]). The created tools assume this direct execution of action clusters style.

3.4 THREE MODELS

Three C DEAC simulations of classical models are used to demonstrate the created tools.

3.4.1 TRAVELING REPAIRMAN

In the traveling repairman model from Cox and Smith [10], 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.

3.4.2 HARBOR

In the harbor model from Schriber [36], ships arrive at a harbor and wait for both a berth and a tugboat to become available. A ship is then escorted by the tugboat to a berth, unloaded, and escorted back to sea. This model can be used to study tugboat utilization and ship in-harbor time.

3.4.3 SINGLE SERVER QUEUE

A single server serves customers one at a time from the front of the queue (first-come, first-served). When the service is complete the customer leaves the queue. This model is often used to estimate long-term average queue length.

CHAPTER 4

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).

Static analysis involves analyzing an object (such as code or a list of specifications) without executing it.

Dynamic analysis involves collecting data during execution of the object of interest (usually 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.

Static analysis can often reveal characteristics of a model not readily apparent from observing only its run-time behavior. Dynamic analysis can miss causal relationships because they did not occur during a particular run or set of runs, as no finite number of runs can necessarily discover all things that are possible in a simulation. (Indeed, there is research within the simulation community that focuses exclusively on rare event simulation.) However, from static analysis, one can reveal the possibility of infrequent situations.

Static code analysis has limitations. From static analysis, one may discover that event A can appear to cause event B, but dynamic analysis often can reveal specifically which events caused which events, 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 combination is not observed, this may be of interest to a modeler or user of the simulation.

These analyses 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.

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.

Both static and dynamic analysis techniques can assist in the goals of this research: "Ultimately, it will be a combination of tools and techniques that help an analyst [understand] programs" [17]. As both types of analysis offer different and complementary insights, the created tools use both static and dynamic techniques.

4.1 LIMITS OF ANALYSIS

Many questions one might like to answer are unsolvable, such as whether or not a particular simulation always terminates, the classical halting problem. Likewise, static analysis cannot determine whether a particular event causes another. Similar observations can be made about the use of dynamic analysis. For example, in the testing community, it is known that in general no amount of testing can show a piece of code is without error; however, testing is still helpful and insightful. We feel that the techniques presented here still can assist modelers to better understand their models.

4.2 TOOL(S) OVERVIEW

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

- Simulation log
- Trip lines
- Scheduled and triggered events
- Event summaries
- Static action cluster interaction graph
- Tallied dynamic action cluster interaction graph
- Dynamic action cluster interaction graph flip book.

These are implemented in six components.

Some components use the results of other components. Specifically, neither creation of the simulation log (which also implements trip lines – these occur as a single component) nor creation of the static ACIG uses any other component. Creation of scheduled and triggered events; event summaries; and the dynamic ACIG flip book each also create the simulation log for their use. Creation of the tallied dynamic ACIG also creates the simulation log and static ACIG for its use.

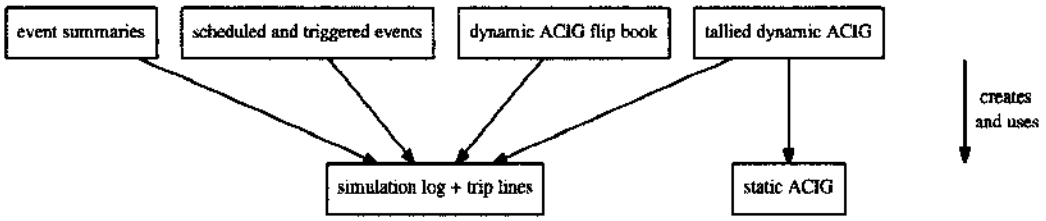


Figure 6. *Component interactions of the tool.*

No original supplied files are ever modified; working copies are made. Similarly, simulation output (`stdout`) is never modified or supplemented; separate files are created by the tools.

Each functionality is discussed in detail below.

4.3 SIMULATION LOG

Using dynamic analysis, a 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).

In the tool, parameters for the tool (such as file locations) and for the simulation (such as number of runs) are read in from a file. The original simulation files are moved and copies are made; the originals are kept untouched and unmodified. The (copies of the) simulation code files are run through `astyle`, an open-source automatic formatter for C (and other programming language) files, so that the format styling of the files is known.

For simplicity, the tool makes four passes on the simulation code files. The first pass injects the file pointer declaration, file open, run number, simulation log header, and file close statements. First, the tool scans for the `main` procedure and injects the file pointer declaration immediately before it. Next, the tool looks for `main`'s error checking of its arguments and immediately thereafter, injects the file open and accompanying error-handling statements. Continuing, the tool looks for the `initialize` call and injects simulation log print statements noting the run number and adding the log header immediately before the call. Finally, the tool looks for the end of `main` and injects the file close statement immediately before it.

The second pass looks for AC procedure calls and scheduled alarms. If an AC procedure is found, immediately after the procedure opening, a simulation log print statement is injected noting the simulation time and the AC being executed. Whenever an `addAlarmList` statement is detected, a simulation log print statement is injected noting the simulation time, the AC that is scheduling, and the AC scheduled.

The third pass looks for the `stateAcList` and its switch statement. Each case of this switch statement corresponds to a particular AC that can be triggered; the first statement of a case is an if statement laying out the conditions for the AC to be triggered. For each case in the switch statement, immediately after the triggering if statement, a simulation log print statement is injected that notes the simulation time, the condition that became true, and the AC triggered.

The fourth pass implements trip lines, discussed below.

The enhanced files are run through `astyle` again to align the added lines for ease of readability by the modeler, if so inclined. The simulation is compiled using the simulation makefile and run the specified number of times, generating the simulation log.

While one line corresponding to one time/action is best for searching data sets with tools such as `grep`, this may not be the best format for reading on a screen or for printing; an additional printer-friendly version is also generated.

The created simulation log is parsed to create a screen- and printer-friendly version of the log. Whenever a line exceeds 80 characters (not anticipated to change but easily modified in one place in the code), the line is broken at the last space before or at the $80^{\text{th}}+1$ character. The remainder of the line is indented the appropriate number of spaces, and the process is repeated as necessary until the remainder of the

line (including indent) has 80 or fewer characters.

Examples are provided below.

4.3.1 EXAMPLE: TRAVELING REPAIRMAN

Simulation output (stdout):

```
Run 1
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
Run 2
Frequency count of AC executions
  AC procId    Frequency
     0            1
     1            1
     2         2000
     3         2000
     4         2000
     5         1363
     6         1363
     7         2000
```

```
⋮
```

Some sample lines of injected code:

```
/* simulation log */
FILE *fp1406944336;
⋮
/* initialize simulation log */
if ((fp1406944336 = fopen("./patrepsimulationlog.txt", "w")) ==
    NULL) {

    fprintf(stderr, "Could not open patrepsimulationlog.txt\n");
    exit(EXIT_FAILURE);
}
⋮

fprintf(fp1406944336, "Run %d\n", arun);
fprintf(fp1406944336, "time:      description\n");
⋮
/* note AC in simulation log */
fprintf(fp1406944336, "%f: initialization\n", clock);
⋮
/* note added alarm in simulation log */
fprintf(fp1406944336, "%f: initialization scheduled failure for time
        %f\n", clock, bact->alarmTime);
⋮
/* note condition and triggered AC in simulation log */
fprintf(fp1406944336, "%f: (repairman.num_repairs >= mrp.max_repairs)
        triggered termination\n", clock);
⋮
/* close simulation log */
fclose(fp1406944336);
```

Part of the generated printer-friendly log:

```
Run 0
time:      description
0.000000: initialization
0.000000: initialization scheduled failure for time 375.411933
0.000000: initialization scheduled failure for time 175.502268
0.000000: initialization scheduled failure for time 641.226639
0.000000: initialization scheduled failure for time 217.206273
0.000000: initialization scheduled failure for time 72.688235
0.000000: initialization scheduled failure for time 178.309819
0.000000: initialization scheduled failure for time 0.004378
0.000000: initialization scheduled failure for time 80.682566
0.000000: initialization scheduled failure for time 151.559682
0.000000: initialization scheduled failure for time 299.116456
0.000000: initialization scheduled failure for time 1374.016025
0.000000: initialization scheduled failure for time 2139.786424
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
8.913395: travel_to_idle
8.913395: travel_to_idle scheduled arrive_at_idle for time 12.413395
12.413395: arrive_at_idle
72.688235: failure
72.688235: (repairman.status == IDLE && SomeFailed()) triggered
           travel_to_facility
72.688235: travel_to_facility
```

```
72.688235: travel_to_facility scheduled begin_repair for time
           75.188235
75.188235: begin_repair
75.188235: begin_repair scheduled end_repair for time 75.715220
75.715220: end_repair
75.715220: end_repair scheduled failure for time 499.740800
75.715220: (mrp.num_failed_facilities == 0 && repairman.status == IDLE
           && repairman.location != idle_loc) triggered travel_to_idle
75.715220: travel_to_idle
75.715220: travel_to_idle scheduled arrive_at_idle for time 78.215220
⋮
73612.466229: end_repair
73612.466229: end_repair scheduled failure for time 74336.614565
73612.466229: (repairman.num_repairs >= mrp.max_repairs) triggered
              termination
73612.466229: termination

Run 1
time:      description
0.000000: initialization
⋮
```

By examining the 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.

4.3.2 EXAMPLE: HARBOR

Simulation output (`stdout`):

```
Run 1
Frequency count of AC executions
  AC procId    Frequency
     0           1
```

```
     1          1
     2       1000
     3       1000
     4       1000
     5       1000
     6       1000
     7       1000
     8         14
     9         14
    10         19
    11         19

Termination!  System time:   99536.71
  Tug utilization:     18.25
  Max number ships waiting at arrival area:   2
⋮
```

From the generated printer-friendly simulation log:

```
Run 0
time:      description
0.000000: initialization
0.000000: initialization scheduled arrival for time 86.301594
86.301594: arrival
86.301594: arrival scheduled arrival for time 126.646943
86.301594: ((num_arr_tugs + tug_to_ocean_ct < num_arr_q) &&
           num_pier_tugs > 0 && num_free_berths > 0) triggered
           move_tug_to_ocean
86.301594: move_tug_to_ocean
86.301594: move_tug_to_ocean scheduled tug_arrive_at_ocean for time
           111.301594
111.301594: tug_arrive_at_ocean
111.301594: (num_arr_q > 0 && num_arr_tugs > 0 && num_free_berths >
            0) triggered enter
```

```
111.301594: enter
111.301594: enter scheduled unload for time 156.301594
⋮
99458.706118: unload
99458.706118: unload scheduled end_unload for time 99491.710185
99491.710185: end_unload
99491.710185: (num_depart_q > 0 && num_pier_tugs > 0 &&
              (num_free_berths == 0 || (num_arr_tugs + tug_to_ocean_ct
              >= num_arr_q))) triggered deberth
99491.710185: deberth
99491.710185: deberth scheduled end_deberth for time 99536.710185
99536.710185: end_deberth
99536.710185: (exit_count >= maxBerths) triggered termination
99536.710185: termination
⋮
```

4.3.3 EXAMPLE: SINGLE SERVER QUEUE

Simulation output (stdout):

```
Run 1
Frequency count of AC executions
  AC procId    Frequency
     0            1
     1            1
     2           22
     3           21
     4           20

Termination!  System time:   567.25
Number served:                  20
Maximum number waiting:          7
⋮
```

From the generated printer-friendly simulation log:

```
Run 0
time:      description
0.000000:  initialization
0.000000:  initialization scheduled arrival for time 0.000000
0.000000:  arrival
0.000000:  arrival scheduled arrival for time 22.006906
0.000000:  (parts.num_waiting > 0 && server.status == IDLE) triggered
           begin_service
0.000000:  begin_service
0.000000:  begin_service scheduled end_service for time 9.610396
9.610396:  end_service
22.006906: arrival
22.006906: arrival scheduled arrival for time 32.294970
22.006906: (parts.num_waiting > 0 && server.status == IDLE) triggered
            begin_service
22.006906: begin_service
22.006906: begin_service scheduled end_service for time 38.772976
⋮
567.251057: begin_service
567.251057: begin_service scheduled end_service for time 579.549980
567.251057: (server.num_served >= mrp.stop_num) triggered termination
567.251057: termination
⋮
```

4.4 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.

In SIMSCRIPT, modelers could provide different routines, one to be invoked whenever a variable was referenced and another whenever it was modified. This was often used to separate statistical analysis code from model code but could serve any purpose of interest to a creative programmer (such as validity or range checking of variables) as well as remove the burden of finding every place in the code where a

variable was referenced or changed.

Similarly, the modeler can add a special line (or lines) to 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. Conditions and reset conditions should pertain to the same variable(s).

Trip lines are an optional addition of one, 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.

Continuing the implementation discussion of Section 4.3 above, the fourth pass makes two passes. The first looks for `trip_when` statements. If the tool finds a `trip_when`, the number of trips is incremented; the `trip_when` variable is noted; and a corresponding if statement and statement block is created. The statement block includes a simulation log print statement noting the simulation time and which trip was tripped, as well as a trip flag (so that the trip line can only trip once). If the `trip_when` has a reset condition, an if statement is also included for the reset condition, again with a statement block that includes a simulation log print statement noting the simulation time and which trip was reset, and a statement resetting the trip flag.

The second pass injects the trip flag initializers at the beginning of the main simulation code file immediately after any comments and `#includes`. Then, wherever one of the `trip_when` variables is changed, the above lines of code are injected immediately thereafter. The `trip_when` lines (not standard C) are removed.

Examples are provided below.

4.4.1 EXAMPLE: TRAVELING REPAIRMAN

From the simulation code:

```
⋮
repairman.work_time = 0.0;
```

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

Injected code:

```
⋮
/* booleans for trip lines */
int trip0 = 0;
⋮

/* user request: trip_when(repairman.work_time >= 10000.0) */
/* first time trip line is tripped, note it in the simulation log */
if ((repairman.work_time >= 10000.0) && (trip0 == 0)) {
    fprintf(fp1408484027, "%f: ! trip line tripped:
            (repairman.work_time >= 10000.0)\n", clock);
    trip0 = 1;
}
⋮
```

From the 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)
⋮
```

4.4.2 EXAMPLE: HARBOR

From the simulation code:

```
⋮
num_berths = 5;
num_free_berths = num_berths;
```

```
trip_when(num_free_berths == 1, num_free_berths >= 3);
⋮
exit_count = 0;
trip_when(exit_count == 10);
⋮
```

Injected code:

```
⋮
/* booleans for trip lines */
int trip0 = 0;
int trip1 = 0;
⋮

/* user request: trip_when(num_free_berths == 1, num_free_berths >=
   3) */
/* first time trip line is tripped, note it in the simulation log */
if ((num_free_berths == 1) && (trip0 == 0)) {
    fprintf(fp1408484050, "%f: ! trip line tripped: (num_free_berths
            == 1)\n", clock);
    trip0 = 1;
}
/* if trip line is reset, note it in the simulation log */
if ((num_free_berths >= 3) && (trip0 == 1)) {
    fprintf(fp1408484050, "%f: ! trip line reset: (num_free_berths
            >= 3)\n", clock);
    trip0 = 0;
}
⋮
/* user request: trip_when(exit_count == 10) */
/* first time trip line is tripped, note it in the simulation log */
if ((exit_count == 10) && (trip1 == 0)) {
    fprintf(fp1408484050, "%f: ! trip line tripped: (exit_count ==
            10)\n", clock);
```

```
    trip1 = 1;
}
⋮
```

From the generated simulation log:

```
⋮
406.689551: tug_arrive_at_ocean
406.689551: (num_arr_q > 0 && num_arr_tugs > 0 && num_free_berths >
            0) triggered enter
406.689551: enter
406.689551: enter scheduled unload for time 451.689551
406.689551: ! trip line tripped: (num_free_berths == 1)
⋮
478.321643: end_unload
478.321643: (num_depart_q > 0 && num_pier_tugs > 0 && (num_free_berths
            == 0 || (num_arr_tugs + tug_to_ocean_ct >= num_arr_q)))
            triggered deberth
478.321643: deberth
478.321643: deberth scheduled end_deberth for time 523.321643
478.321643: ! trip line reset: (num_free_berths >= 3)
⋮
608.983167: end_unload
608.983167: (num_depart_q > 0 && num_pier_tugs > 0 && (num_free_berths
            == 0 || (num_arr_tugs + tug_to_ocean_ct >= num_arr_q)))
            triggered deberth
608.983167: deberth
608.983167: deberth scheduled end_deberth for time 653.983167
653.983167: end_deberth
653.983167: ! trip line tripped: (exit_count == 10)
⋮
```

4.4.3 EXAMPLE: SINGLE SERVER QUEUE

From the simulation code:

```
⋮
parts.num_waiting = 0;
trip_when(parts.num_waiting > 4, parts.num_waiting == 0);
⋮
server.num_served = 0;
trip_when(server.num_served == 6);
trip_when(server.num_served == 12);
⋮
```

From the generated simulation log:

```
⋮
69.884118: begin_service
69.884118: begin_service scheduled end_service for time 133.241050
82.616900: arrival
82.616900: arrival scheduled arrival for time 86.877934
86.877934: arrival
86.877934: arrival scheduled arrival for time 97.330579
97.330579: arrival
97.330579: arrival scheduled arrival for time 97.330835
97.330835: arrival
97.330835: arrival scheduled arrival for time 102.060503
102.060503: arrival
102.060503: ! trip line tripped: (parts.num_waiting > 4)
102.060503: arrival scheduled arrival for time 110.945036
⋮
148.019685: end_service
148.019685: (parts.num_waiting > 0 && server.status == IDLE)
            triggered begin_service
148.019685: begin_service
148.019685: begin_service scheduled end_service for time 156.415883
156.415883: end_service
156.415883: ! trip line tripped: (server.num_served == 6)
```

```
156.415883: (parts.num_waiting > 0 && server.status == IDLE)
            triggered begin_service
156.415883: begin_service
156.415883: begin_service scheduled end_service for time 169.503654
⋮
226.292220: end_service
226.292220: (parts.num_waiting > 0 && server.status == IDLE)
            triggered begin_service
226.292220: begin_service
226.292220: ! trip line reset: (parts.num_waiting == 0)
226.292220: begin_service scheduled end_service for time 266.078875
⋮
266.078875: end_service
266.078875: end_service
266.078875: ! trip line tripped: (server.num_served == 12)
⋮
```

4.5 SCHEDULED AND TRIGGERED EVENTS

While there are plenty of code coverage tools that 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.

In the DEAC implementation (as with any implementation), only certain parts of the code correspond to the conceptual model; because of the DEAC structure, these sections of the code can be known and noted. In the simulate routine, there are two

main sections that correspond to the conceptual model: the execution of scheduled ACs (phase B), and the repeated scanning of conditions (and possible execution) of triggered ACs (phase D).

The tool first creates the simulation log. It then creates a modified makefile to support `gcov`, makes the simulation, runs the simulation, and runs `gcov`. Next, it considers the `.gcov` file and the main simulation file. The phase B section of the code is considered; for each case in the switch statement, the tool checks if the case statement was executed. If it was, the AC is noted as scheduled; if not, it is noted as not scheduled. Similarly, the phase D section of the code is considered; for each case in the switch statement, the tool checks if the case statement was executed. If it was, the AC is noted as triggered; if not, it is noted as not triggered. These notes are then sorted into scheduled events, unscheduled events, triggered events, and untriggered events.

Examples are provided below.

4.5.1 EXAMPLE: TRAVELING REPAIRMAN

```
during the simulation run:

scheduled events:
    arrive_at_idle
    begin_repair
    end_repair
    failure

unscheduled events:
    termination
    travel_to_facility
    travel_to_idle

triggered events:
    termination
    travel_to_facility
    travel_to_idle
```

```
untriggered events:
    ~ none ~
```

4.5.2 EXAMPLE: HARBOR

```
during the simulation run:

scheduled events:
    arrival
    end_deberth
    end_unload
    tug_arrive_at_ocean
    tug_arrive_at_pier
    unload

unscheduled events:
    ~ none ~

triggered events:
    deberth
    enter
    move_tug_to_ocean
    move_tug_to_pier
    termination

untriggered events:
    ~ none ~
```

4.5.3 EXAMPLE: SINGLE SERVER QUEUE

```
during the simulation run:

scheduled events:
    arrival
    end_service
```

```
unscheduled events:
    ~ none ~

triggered events:
    begin_service
    termination

untriggered events:
    ~ none ~
```

4.6 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 one past 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.

The tool first creates the simulation log. For each simulation run, a tally is created, counting each event, each event scheduled and by which event it was scheduled, and each event triggered and by which event it was triggered. The tally is then sorted, with initialization first, termination last, and the remaining events in between.

Examples are provided below.

4.6.1 EXAMPLE: TRAVELING REPAIRMAN

```
Run 0
```

```
Total simulation time: 73612.466229

Events:
    initialization
        occurrences: 1
        events scheduled:
            failure: 12 times
        events triggered:
            ~ none ~
    arrive_at_idle
        occurrences: 1351
        events scheduled:
            ~ none ~
        events triggered:
            travel_to_facility: 117 times
    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
    failure
        occurrences: 2001
        events scheduled:
            ~ none ~
```

```
        events triggered:
            travel_to_facility: 1235 times
    travel_to_facility
        occurrences: 2000
        events scheduled:
            begin_repair: 2000 times
        events triggered:
            ~ none ~
    travel_to_idle
        occurrences: 1351
        events scheduled:
            arrive_at_idle: 1351 times
        events triggered:
            ~ none ~
    termination
        occurrences: 1
        events scheduled:
            ~ none ~
        events triggered:
            ~ none ~

~~~~~~~~~
Run 1

Total simulation time: 74386.284534

Events:
    initialization
        occurrences: 1
        events scheduled:
            failure: 12 times
        events triggered:
            ~ none ~
⋮
```

4.6.2 EXAMPLE: HARBOR

```
Run 0

Total simulation time: 99536.710185

Events:

     initialization
        occurrences: 1
        events scheduled:
            arrival: 1 time
        events triggered:
            ~ none ~
    arrival
        occurrences: 1000
        events scheduled:
            arrival: 1000 times
        events triggered:
            enter: 978 times
            move_tug_to_ocean: 15 times
    deberth
        occurrences: 1000
        events scheduled:
            end_deberth: 1000 times
        events triggered:
            move_tug_to_ocean: 2 times
    end_deberth
        occurrences: 1000
        events scheduled:
            ~ none ~
        events triggered:
            enter: 10 times
            move_tug_to_pier: 1 time
            termination: 1 time
```

```
end_unload
    occurrences: 1000
    events scheduled:
        ~ none ~
    events triggered:
        deberth: 979 times
        move_tug_to_pier: 13 times
enter
    occurrences: 1000
    events scheduled:
        unload: 1000 times
    events triggered:
        ~ none ~
move_tug_to_ocean
    occurrences: 19
    events scheduled:
        tug_arrive_at_ocean: 19 times
    events triggered:
        ~ none ~
move_tug_to_pier
    occurrences: 14
    events scheduled:
        tug_arrive_at_pier: 14 times
    events triggered:
        ~ none ~
tug_arrive_at_ocean
    occurrences: 19
    events scheduled:
        ~ none ~
    events triggered:
        enter: 12 times
tug_arrive_at_pier
    occurrences: 14
    events scheduled:
```

```
            ~ none ~
        events triggered:
            deberth: 9 times
    unload
        occurrences: 1000
        events scheduled:
            end_unload: 1000 times
        events triggered:
            deberth: 12 times
            move_tug_to_ocean: 2 times
    termination
        occurrences: 1
        events scheduled:
            ~ none ~
        events triggered:
            ~ none ~
⋮
```

4.6.3 EXAMPLE: SINGLE SERVER QUEUE

```
Run 0

Total simulation time: 567.251057

Events:
    initialization
        occurrences: 1
        events scheduled:
            arrival: 1 time
        events triggered:
            ~ none ~
    arrival
        occurrences: 22
        events scheduled:
            arrival: 22 times
```

```
        events triggered:
            begin_service: 6 times
    begin_service
        occurrences: 21
        events scheduled:
            end_service: 21 times
        events triggered:
            termination: 1 time
    end_service
        occurrences: 20
        events scheduled:
            ~ none ~
        events triggered:
            begin_service: 15 times
    termination
        occurrences: 1
        events scheduled:
            ~ none ~
        events triggered:
            ~ none ~
⋮
```

4.7 STATIC ACTION CLUSTER INTERACTION GRAPH

Using static analysis, the action cluster interaction graph is automatically generated.

In a precursor to this automation effort, part of my research focused on using a now-commercial tool (CodeSurfer [1]) to generate the ACIG. Although the goal was to explore what kind of tools might be able to reproduce information garnered by hand, I discovered unrealized errors in graphs in a reviewed, published paper, [23]. Research on how visualization can assist understanding coupled with how easily one can miss interactions – in [23], among only 12 action clusters – again demonstrates the usefulness of automating these analyses.

In the DEAC implementation, a file contains the possible successors of each AC in the model. Puthoff [30] presents a way to identify such successors (though this

list may not be minimal since creation of such a list is unsolvable). The tool first creates a list of all ACs by scanning the main simulation file. Next, the tool parses the successors file to determine which, if any, successors each AC could have. Using this information, the tool creates a DOT graph description language file. Positions for each AC are calculated and added to the DOT file so that the ACIG is always circular. Recall a solid line in the ACIG means a given action cluster could trigger an action cluster. A line in the DOT file is created for each solid edge for the graph: a solid line is created between an AC and each of its possible successors. Recall also that a dashed line in the ACIG means that an action cluster can schedule an action cluster. A line in the DOT file is created for each dashed line for the graph: the main simulation file is scanned; a dashed line is created between an AC and each AC it can schedule. Finally, the DOT file is processed by **neato** to create a portable document file containing the static action cluster interaction graph.

Examples are provided below.

4.7.1 EXAMPLE: TRAVELING REPAIRMAN

Generated DOT file:

```
digraph patrep {
  initialization [pos="4.0,0.0!"];
  termination [pos="2.82842712475,2.82842712475!"];
  failure [pos="2.44929359829e-16,4.0!"];
  begin_repair [pos="-2.82842712475,2.82842712475!"];
  end_repair [pos="-4.0,4.89858719659e-16!"];
  travel_to_idle [pos="-2.82842712475,-2.82842712475!"];
  arrive_at_idle [pos="-7.34788079488e-16,-4.0!"];
  travel_to_facility [pos="2.82842712475,-2.82842712475!"];

  failure -> travel_to_facility;
  end_repair -> termination;
  end_repair -> travel_to_facility;
  end_repair -> travel_to_idle;
  arrive_at_idle -> travel_to_facility;
  initialization -> failure [style = dashed];
```

```
  travel_to_facility -> begin_repair [style = dashed];
  begin_repair -> end_repair [style = dashed];
  end_repair -> failure [style = dashed];
  travel_to_idle -> arrive_at_idle [style = dashed];
}
```

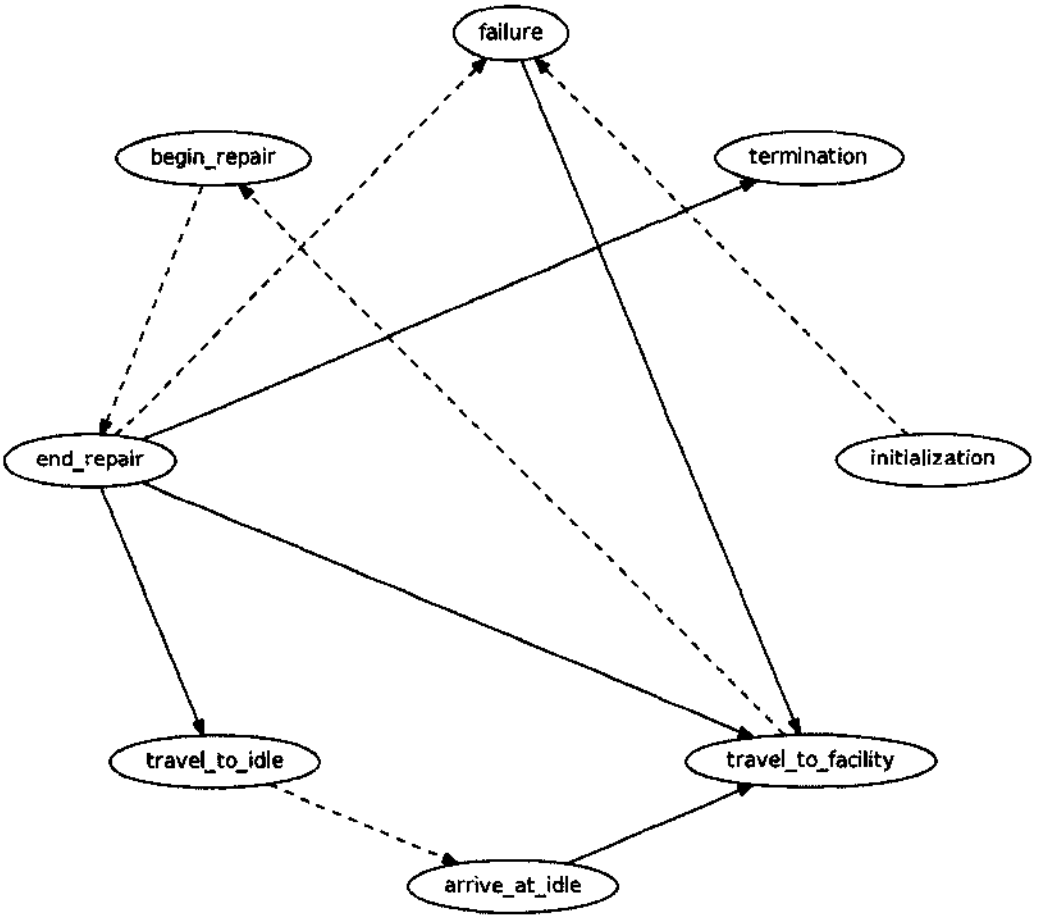


Figure 7. *Generated graph: traveling repairman static action cluster interaction graph.*

4.7.2 EXAMPLE: HARBOR

Generated DOT file:

```
digraph harbor {
  initialization [pos="6.0,0.0!"];
  termination [pos="5.19615242271,3.0!"];
  arrival [pos="3.0,5.19615242271!"];
  enter [pos="3.67394039744e-16,6.0!"];
  unload [pos="-3.0,5.19615242271!"];
  end_unload [pos="-5.19615242271,3.0!"];
  deberth [pos="-6.0,7.34788079488e-16!"];
  end_deberth [pos="-5.19615242271,-3.0!"];
  move_tug_to_pier [pos="-3.0,-5.19615242271!"];
  tug_arrive_at_pier [pos="-1.10218211923e-15,-6.0!"];
  move_tug_to_ocean [pos="3.0,-5.19615242271!"];
  tug_arrive_at_ocean [pos="5.19615242271,-3.0!"];

  arrival -> enter;
  arrival -> move_tug_to_ocean;
  enter -> deberth;
  enter -> move_tug_to_pier;
  enter -> move_tug_to_ocean;
  unload -> deberth;
  unload -> move_tug_to_ocean;
  end_unload -> deberth;
  end_unload -> move_tug_to_pier;
  deberth -> enter;
  deberth -> move_tug_to_pier;
  deberth -> move_tug_to_ocean;
  end_deberth -> termination;
  end_deberth -> enter;
  end_deberth -> move_tug_to_pier;
  tug_arrive_at_pier -> deberth;
  tug_arrive_at_pier -> move_tug_to_ocean;
```

```
  tug_arrive_at_ocean -> enter;
  tug_arrive_at_ocean -> move_tug_to_pier;
  initialization -> arrival [style = dashed];
  arrival -> arrival [style = dashed];
  enter -> unload [style = dashed];
  unload -> end_unload [style = dashed];
  deberth -> end_deberth [style = dashed];
  move_tug_to_pier -> tug_arrive_at_pier [style = dashed];
  move_tug_to_ocean -> tug_arrive_at_ocean [style = dashed];
}
```

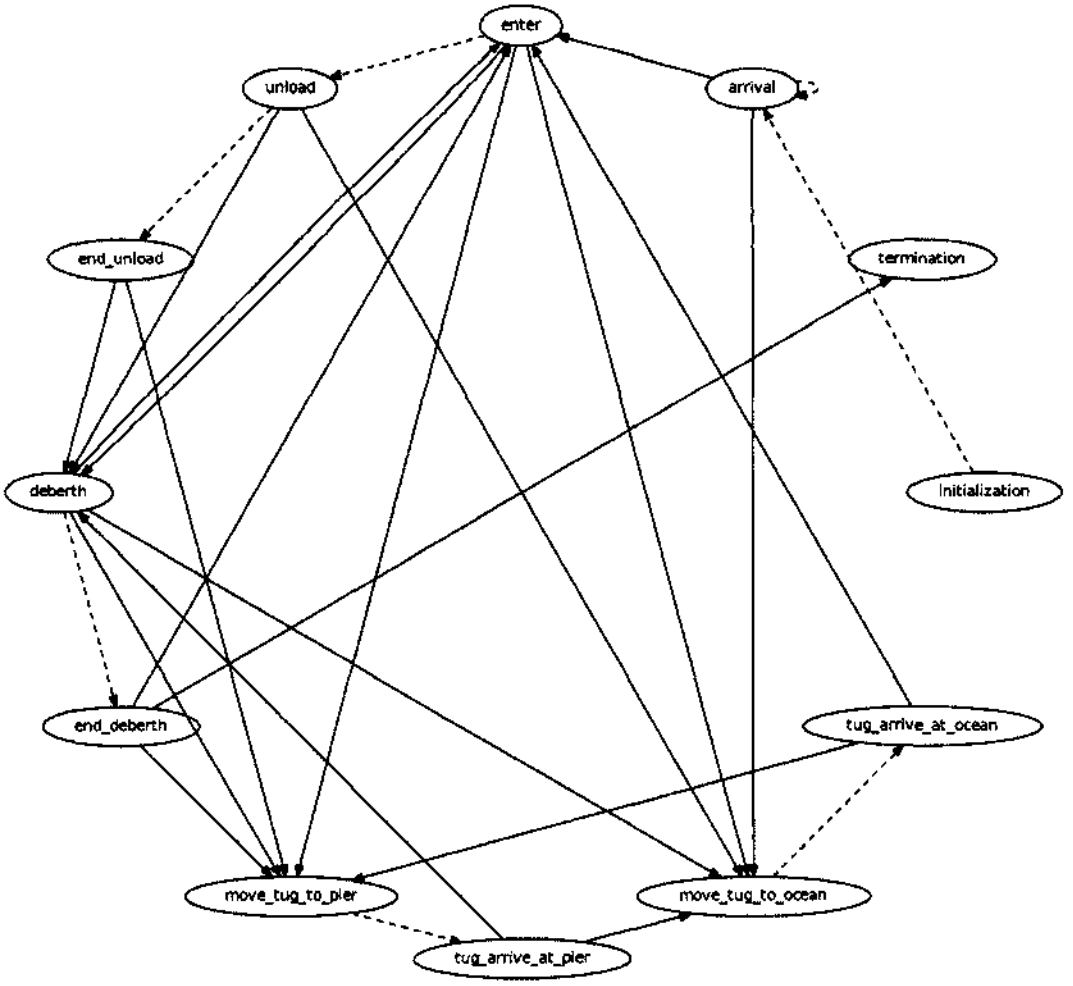


Figure 8. *Generated graph: harbor static action cluster interaction graph.*

4.7.3 EXAMPLE: SINGLE SERVER QUEUE

Generated DOT file:

```
digraph mm1 {
  initialization [pos="2.5,0.0!"];
  termination [pos="0.772542485937,2.37764129074!"];
  arrival [pos="-2.02254248594,1.46946313073!"];
  begin_service [pos="-2.02254248594,-1.46946313073!"];
  end_service [pos="0.772542485937,-2.37764129074!"];

  arrival -> begin_service;
  end_service -> begin_service;
  end_service -> termination;
  initialization -> arrival [style = dashed];
  arrival -> arrival [style = dashed];
  begin_service -> end_service [style = dashed];
}
```

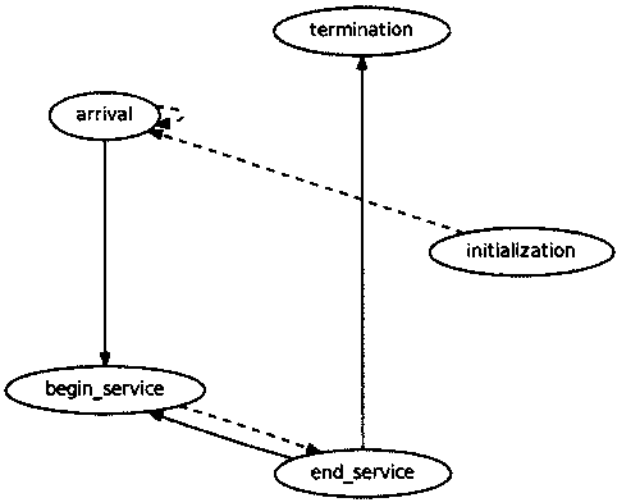


Figure 9. *Generated graph: single server queue static action cluster interaction graph.*

4.8 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.

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 to a modeler or user of the simulation.

The tool first creates the static action cluster interaction graph, forming the basis of the DOT file, and the simulation log. For each simulation run, the graph is labeled with the run; a tally is created, counting each time an AC schedules an AC, and each time an AC triggers an AC. These are then added as labels for each line in the DOT file. If an edge is not in the tally, it is labeled "0." Finally, the DOT file is processed by `neato` to create a portable document file containing the dynamic action cluster interaction graph.

Given an arbitrary condition specification, the automatic creation of a static ACIG with no redundant or unnecessary edges is unsolvable [23]; 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.

Examples are provided below.

4.8.1 EXAMPLE: TRAVELING REPAIRMAN

Generated DOT file:

```
digraph patrep {
  label = "run 0";
  initialization [pos="4.0,0.0!"];
  termination [pos="2.82842712475,2.82842712475!"];
  failure [pos="2.44929359829e-16,4.0!"];
```

```
  begin_repair [pos="-2.82842712475,2.82842712475!"];
  end_repair [pos="-4.0,4.89858719659e-16!"];
  travel_to_idle [pos="-2.82842712475,-2.82842712475!"];
  arrive_at_idle [pos="-7.34788079488e-16,-4.0!"];
  travel_to_facility [pos="2.82842712475,-2.82842712475!"];

  arrive_at_idle -> travel_to_facility [label = " 7 "];
  begin_repair -> end_repair [style = dashed] [label = " 100 "];
  end_repair -> failure [style = dashed] [label = " 100 "];
  end_repair -> termination [label = " 1 "];
  end_repair -> travel_to_facility [label = " 31 "];
  end_repair -> travel_to_idle [label = " 68 "];
  failure -> travel_to_facility [label = " 62 "];
  initialization -> failure [style = dashed] [label = " 12 "];
  travel_to_facility -> begin_repair [style = dashed] [label =
                        " 100 "];
  travel_to_idle -> arrive_at_idle [style = dashed] [label = " 68 "];
}
```

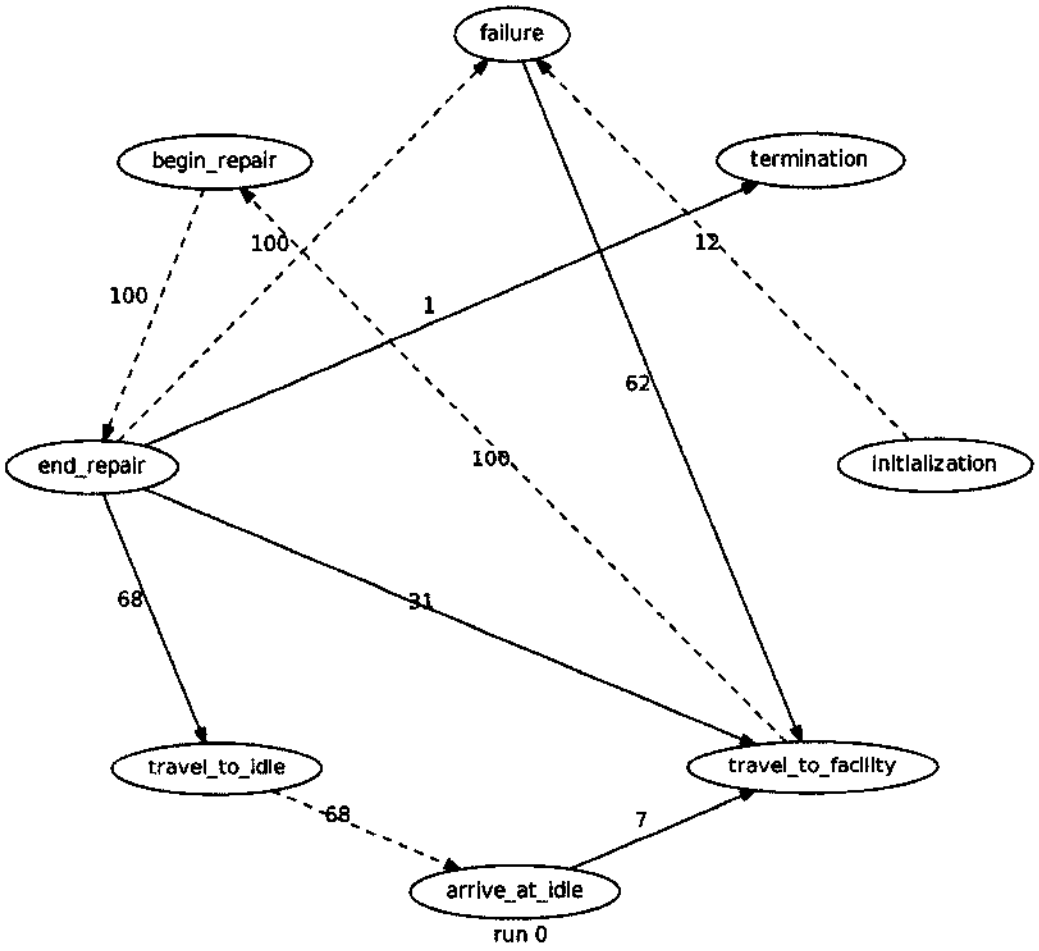


Figure 10. *Generated graph: traveling repairman tallied dynamic action cluster interaction graph.*

4.8.2 EXAMPLE: HARBOR

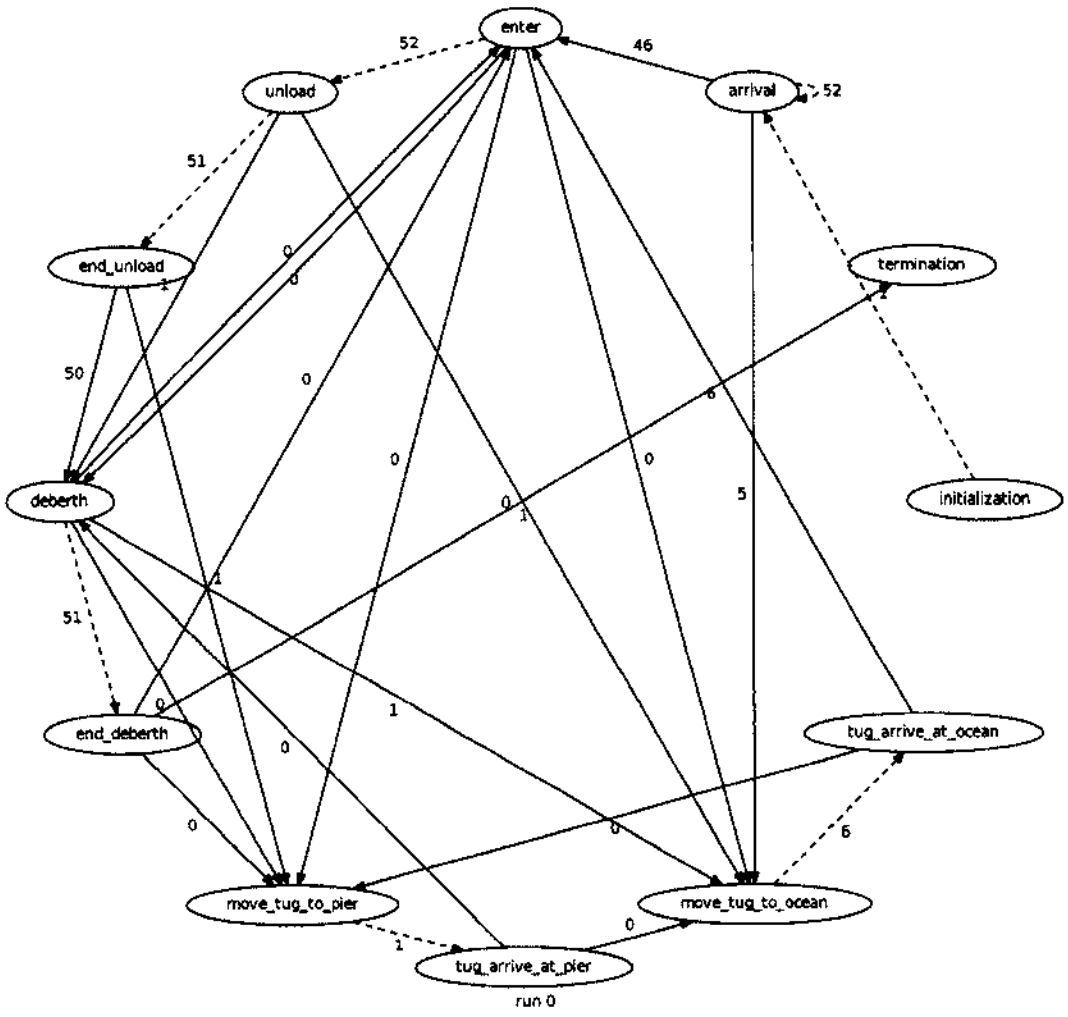


Figure 11. *Generated graph: harbor tallied dynamic action cluster interaction graph.*

4.8.3 EXAMPLE: SINGLE SERVER QUEUE

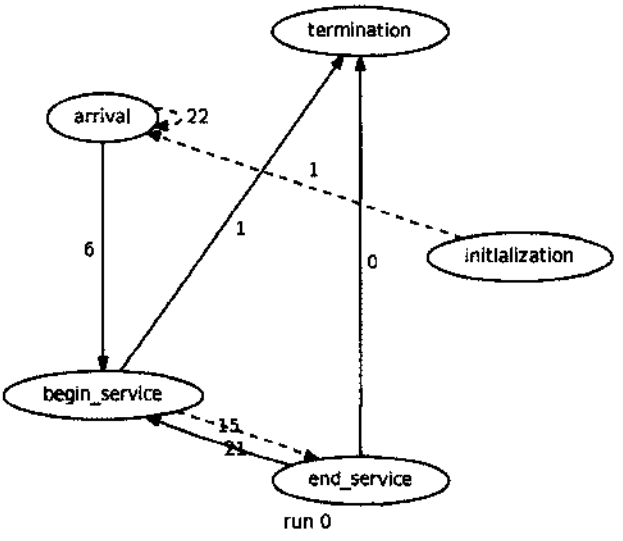


Figure 12. *Generated graph: single server queue tallied dynamic action cluster interaction graph.*

4.9 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 a visual representation of the entire simulation run.

The tool first creates the simulation log. Next, the tool creates a list of all ACs by scanning the main simulation file. Positions for each AC are calculated so that the ACIG is always circular and each AC is in the same location for each page in the flip book. The tool parses the simulation log. For each simulation run and simulation time step, one ACIG is created. The graph is labeled with the run and simulation time. The first thing to occur during any simulation time step is the execution of an AC; this AC is given a bold circle. A line in the DOT file is created for each solid edge for the graph: a solid line is created between an AC and any ACs it triggered. A line in the DOT file is created for each dashed line for the graph: a dashed line is created between an AC and any ACs it scheduled, and is labeled with the time

for when it is scheduled. After the current time step is complete, the DOT file is processed by `neato` to create a portable document file. Each graph is then appended to the end of the flip book using `pdfunite`.

Partial examples are provided below.

4.9.1 EXAMPLE: TRAVELING REPAIRMAN

Generated DOT file:

```
digraph patrep {
  label = "run 0, time 0.000000";
  initialization [pos="4.0,0.0!"];
  termination [pos="2.82842712475,2.82842712475!"];
  failure [pos="2.44929359829e-16,4.0!"];
  begin_repair [pos="-2.82842712475,2.82842712475!"];
  end_repair [pos="-4.0,4.89858719659e-16!"];
  travel_to_idle [pos="-2.82842712475,-2.82842712475!"];
  arrive_at_idle [pos="-7.34788079488e-16,-4.0!"];
  travel_to_facility [pos="2.82842712475,-2.82842712475!"];

  initialization [style=bold];
  initialization -> failure [style=dashed, label=
                    " for time 375.411933 "];
  initialization -> failure [style=dashed, label=
                    " for time 175.502268 "];
  initialization -> failure [style=dashed, label=
                    " for time 641.226639 "];
  initialization -> failure [style=dashed, label=
                    " for time 217.206273 "];
  initialization -> failure [style=dashed, label=
                    " for time 72.688235 "];
  initialization -> failure [style=dashed, label=
                    " for time 178.309819 "];
  initialization -> failure [style=dashed, label=
                    " for time 0.004378 "];
```

```
  initialization -> failure [style=dashed, label=
                    " for time 80.682566 "];
  initialization -> failure [style=dashed, label=
                    " for time 151.559682 "];
  initialization -> failure [style=dashed, label=
                    " for time 299.116456 "];
  initialization -> failure [style=dashed, label=
                    " for time 1374.016025 "];
  initialization -> failure [style=dashed, label=
                    " for time 2139.786424 "];
}
```

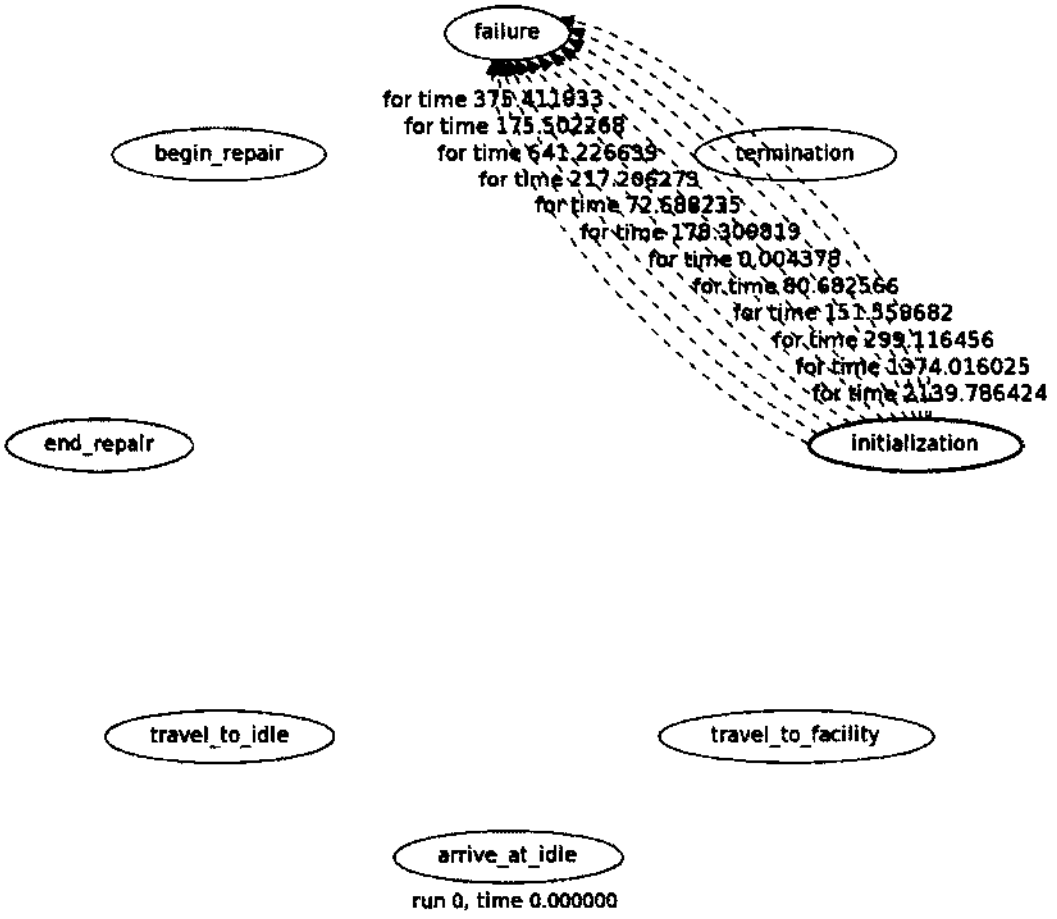


Figure 13. *Generated graph: traveling repairman dynamic action cluster interaction graph flip book, page 1 of 369.*

Generated DOT file:

```
digraph patrep {
  label = "run 0, time 0.000000";
  initialization [pos="4.0,0.0!"];
  termination [pos="2.82842712475,2.82842712475!"];
  failure [pos="2.44929359829e-16,4.0!"];
  begin_repair [pos="-2.82842712475,2.82842712475!"];
  end_repair [pos="-4.0,4.89858719659e-16!"];
  travel_to_idle [pos="-2.82842712475,-2.82842712475!"];
  arrive_at_idle [pos="-7.34788079488e-16,-4.0!"];
  travel_to_facility [pos="2.82842712475,-2.82842712475!"];

  failure [style=bold];
  failure -> travel_to_facility;
  travel_to_facility -> begin_repair [style=dashed, label=
                        " for time 3.504378 "];
}
```

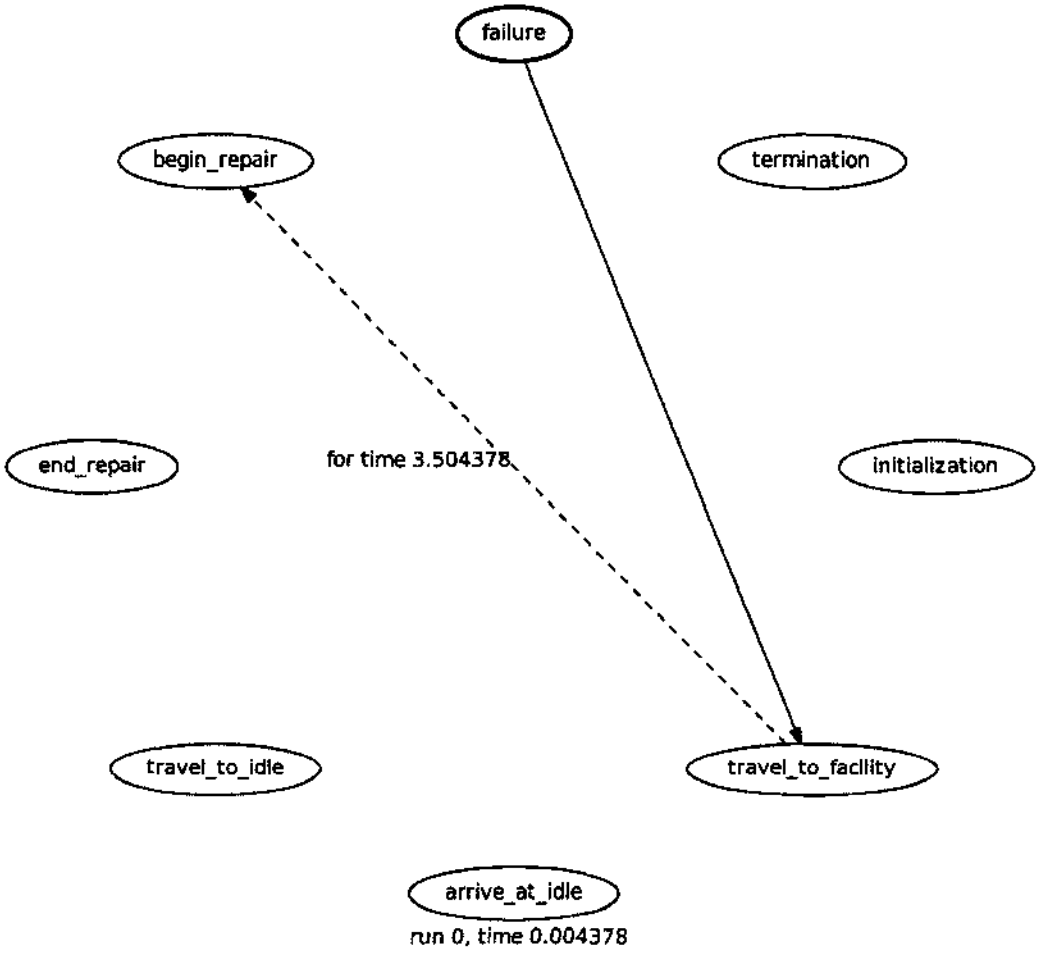


Figure 14. *Generated graph: traveling repairman dynamic action cluster interaction graph flip book, page 2 of 369.*

Generated DOT file:

```
digraph patrep {
  label = "run 0, time 0.000000";
  initialization [pos="4.0,0.0!"];
  termination [pos="2.82842712475,2.82842712475!"];
  failure [pos="2.44929359829e-16,4.0!"];
  begin_repair [pos="-2.82842712475,2.82842712475!"];
  end_repair [pos="-4.0,4.89858719659e-16!"];
  travel_to_idle [pos="-2.82842712475,-2.82842712475!"];
```

```
  arrive_at_idle [pos="-7.34788079488e-16,-4.0!"];
  travel_to_facility [pos="2.82842712475,-2.82842712475!"];

  begin_repair [style=bold];
  begin_repair -> end_repair [style=dashed, label=
                  " for time 8.913395 "];
}
```

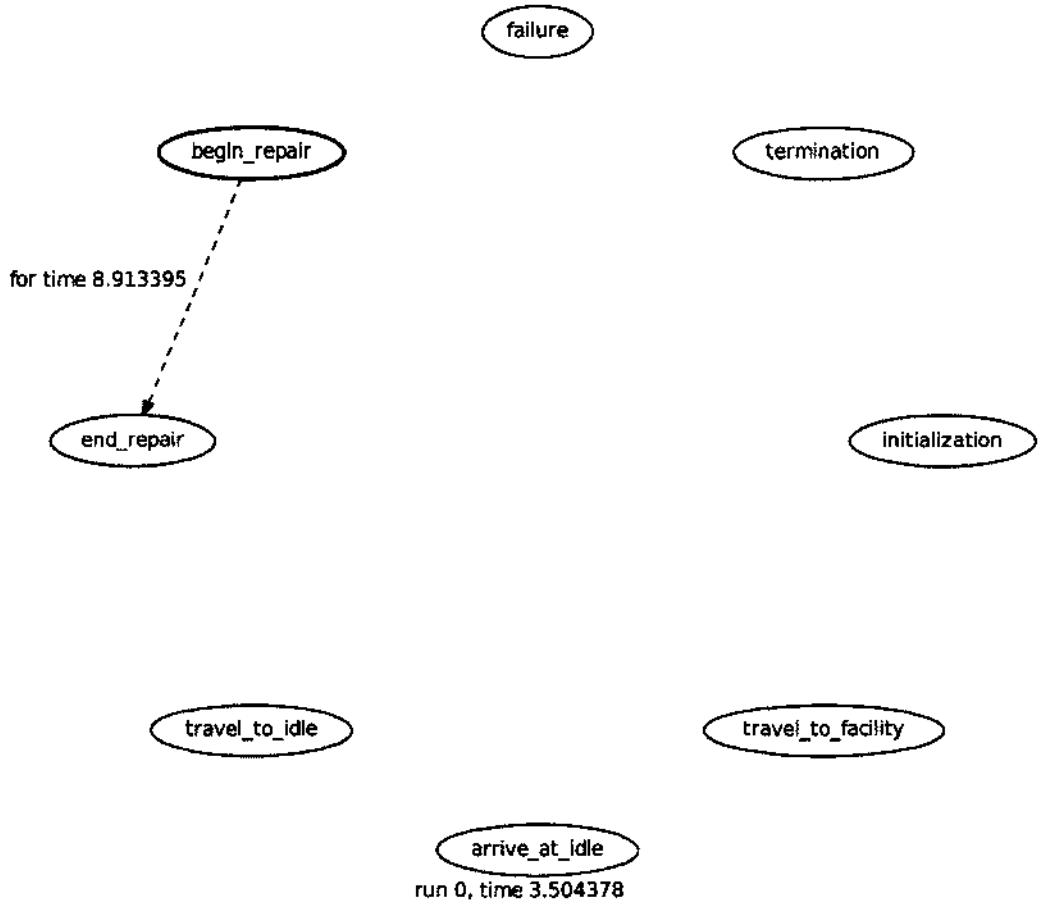


Figure 15. *Generated graph: traveling repairman dynamic action cluster interaction graph flip book, page 3 of 369.*

Generated DOT file:

```
digraph patrep {
  label = "run 0, time 0.000000";
  initialization [pos="4.0,0.0!"];
  termination [pos="2.82842712475,2.82842712475!"];
  failure [pos="2.44929359829e-16,4.0!"];
  begin_repair [pos="-2.82842712475,2.82842712475!"];
  end_repair [pos="-4.0,4.89858719659e-16!"];
  travel_to_idle [pos="-2.82842712475,-2.82842712475!"];
  arrive_at_idle [pos="-7.34788079488e-16,-4.0!"];
  travel_to_facility [pos="2.82842712475,-2.82842712475!"];

  end_repair [style=bold];
  end_repair -> failure [style=dashed, label=
              " for time 251.966998 "];
  end_repair -> travel_to_idle;
  travel_to_idle -> arrive_at_idle [style=dashed, label=
                  " for time 12.413395 "];
}
```

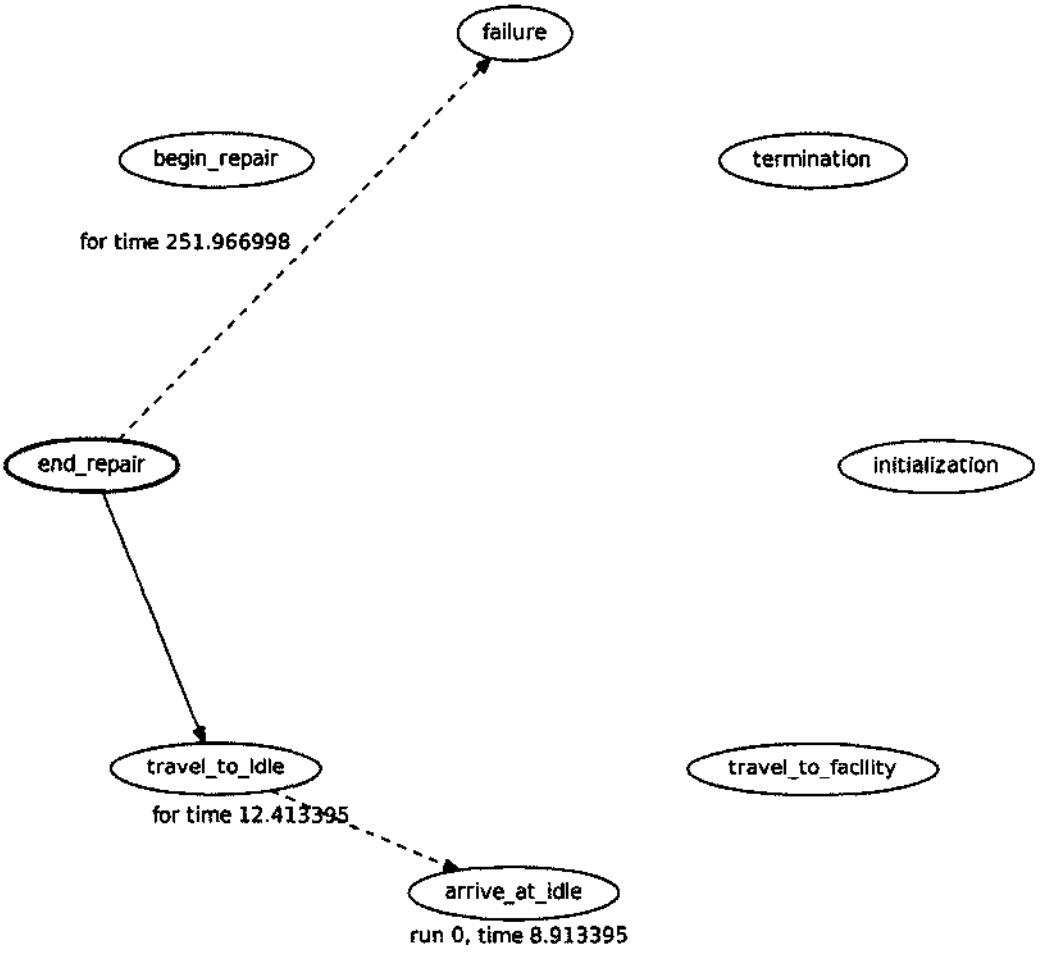


Figure 16. *Generated graph: traveling repairman dynamic action cluster interaction graph flip book, page 4 of 369.*

Generated DOT file:

```
digraph patrep {
  label = "run 0, time 0.000000";
  initialization [pos="4.0,0.0!"];
  termination [pos="2.82842712475,2.82842712475!"];
  failure [pos="2.44929359829e-16,4.0!"];
  begin_repair [pos="-2.82842712475,2.82842712475!"];
  end_repair [pos="-4.0,4.89858719659e-16!"];
  travel_to_idle [pos="-2.82842712475,-2.82842712475!"];
```

```
  arrive_at_idle [pos="-7.34788079488e-16,-4.0!"];
  travel_to_facility [pos="2.82842712475,-2.82842712475!"];

  arrive_at_idle [style=bold];
}
```

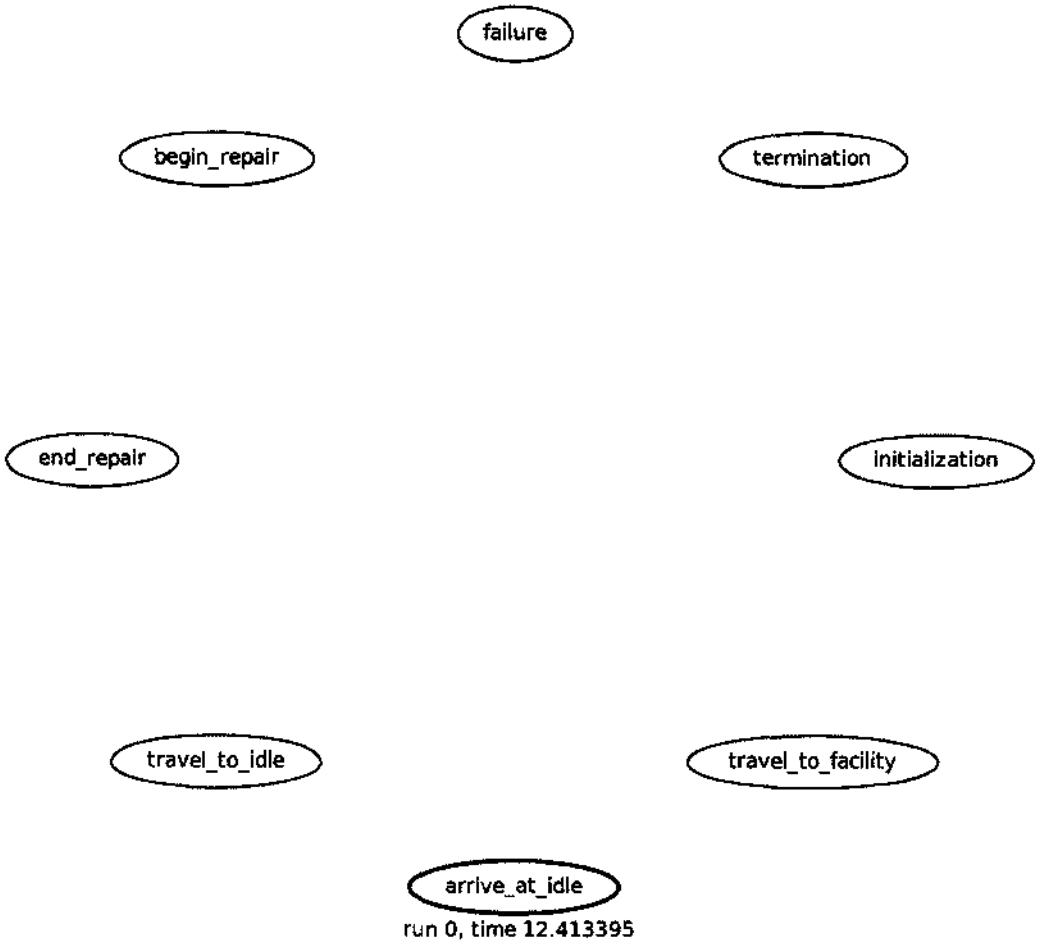


Figure 17. *Generated graph: traveling repairman dynamic action cluster interaction graph flip book, page 5 of 369.*

Generated DOT file:

```
digraph patrep {
  label = "run 0, time 0.000000";
  initialization [pos="4.0,0.0!"];
  termination [pos="2.82842712475,2.82842712475!"];
  failure [pos="2.44929359829e-16,4.0!"];
  begin_repair [pos="-2.82842712475,2.82842712475!"];
  end_repair [pos="-4.0,4.89858719659e-16!"];
  travel_to_idle [pos="-2.82842712475,-2.82842712475!"];
  arrive_at_idle [pos="-7.34788079488e-16,-4.0!"];
  travel_to_facility [pos="2.82842712475,-2.82842712475!"];

  begin_repair [style=bold];
  begin_repair -> end_repair [style=dashed, label=
                " for time 3698.025941 "];
}
```

Figure 18. *Generated graph: traveling repairman dynamic action cluster interaction graph flip book, page 368 of 369.*

Generated DOT file:

```
digraph patrep {
  label = "run 0, time 0.000000";
  initialization [pos="4.0,0.0!"];
  termination [pos="2.82842712475,2.82842712475!"];
  failure [pos="2.44929359829e-16,4.0!"];
  begin_repair [pos="-2.82842712475,2.82842712475!"];
  end_repair [pos="-4.0,4.89858719659e-16!"];
  travel_to_idle [pos="-2.82842712475,-2.82842712475!"];
  arrive_at_idle [pos="-7.34788079488e-16,-4.0!"];
  travel_to_facility [pos="2.82842712475,-2.82842712475!"];

  end_repair [style=bold];
  end_repair -> failure [style=dashed, label=
               " for time 3945.487127 "];
  end_repair -> termination;
}
```

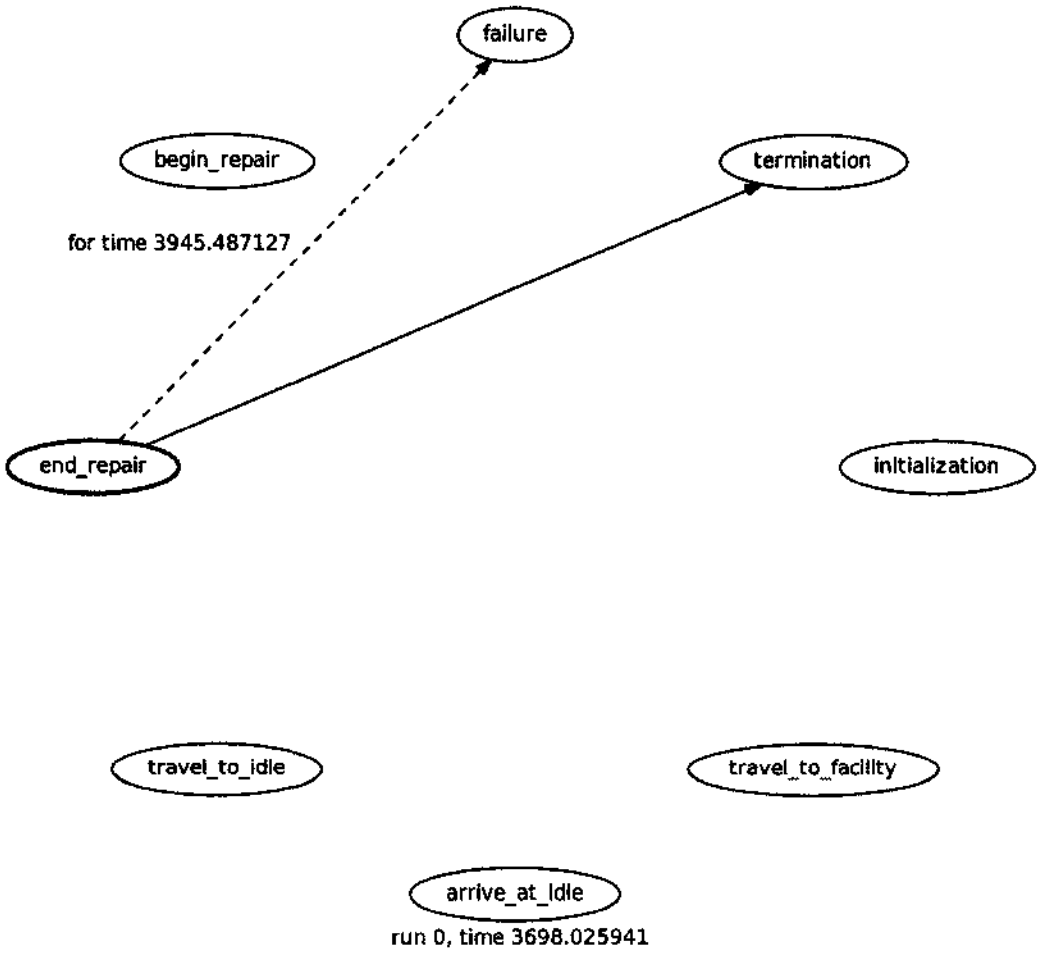


Figure 19. *Generated graph: traveling repairman dynamic action cluster interaction graph flip book, page 369 of 369.*

4.9.2 EXAMPLE: HARBOR

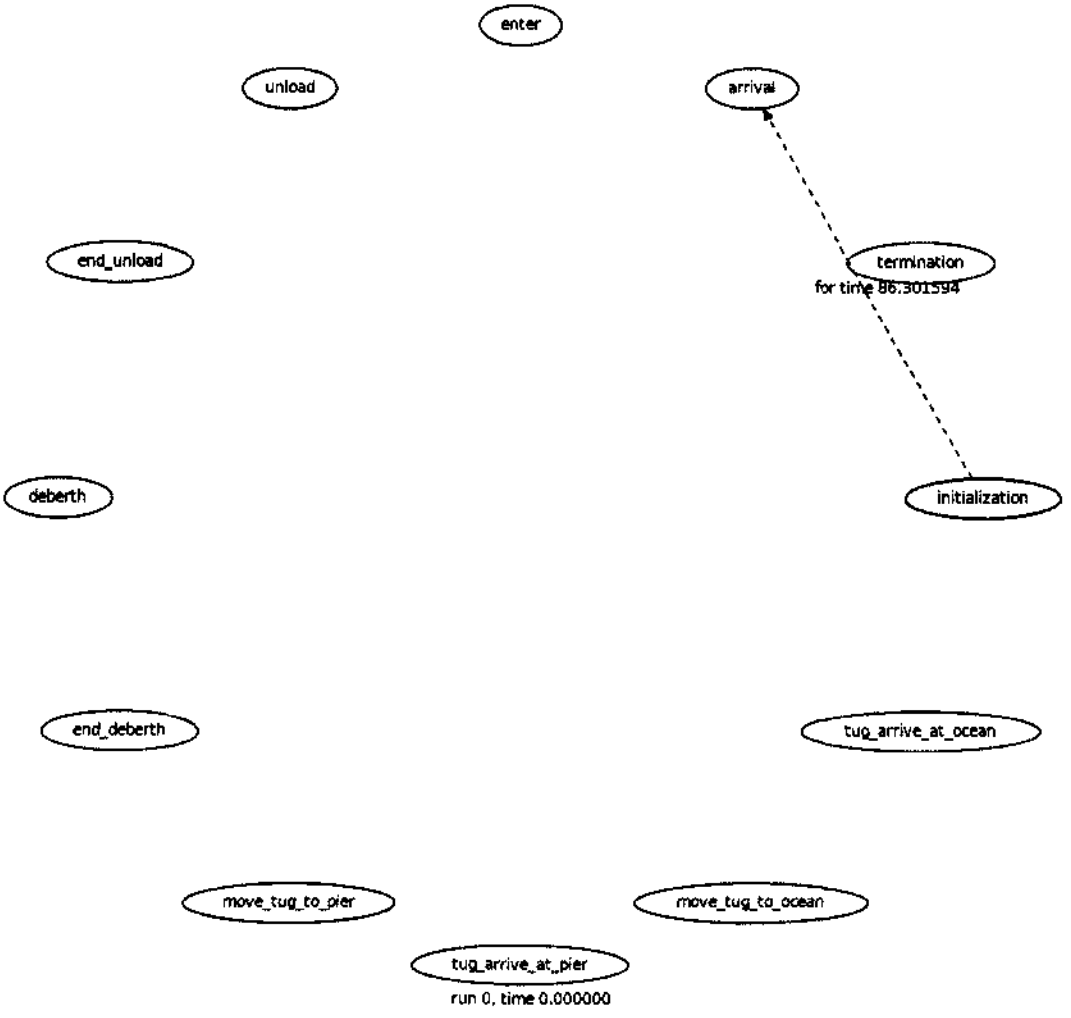


Figure 20. *Generated graph: harbor dynamic action cluster interaction graph flip book, page 1 of 212.*

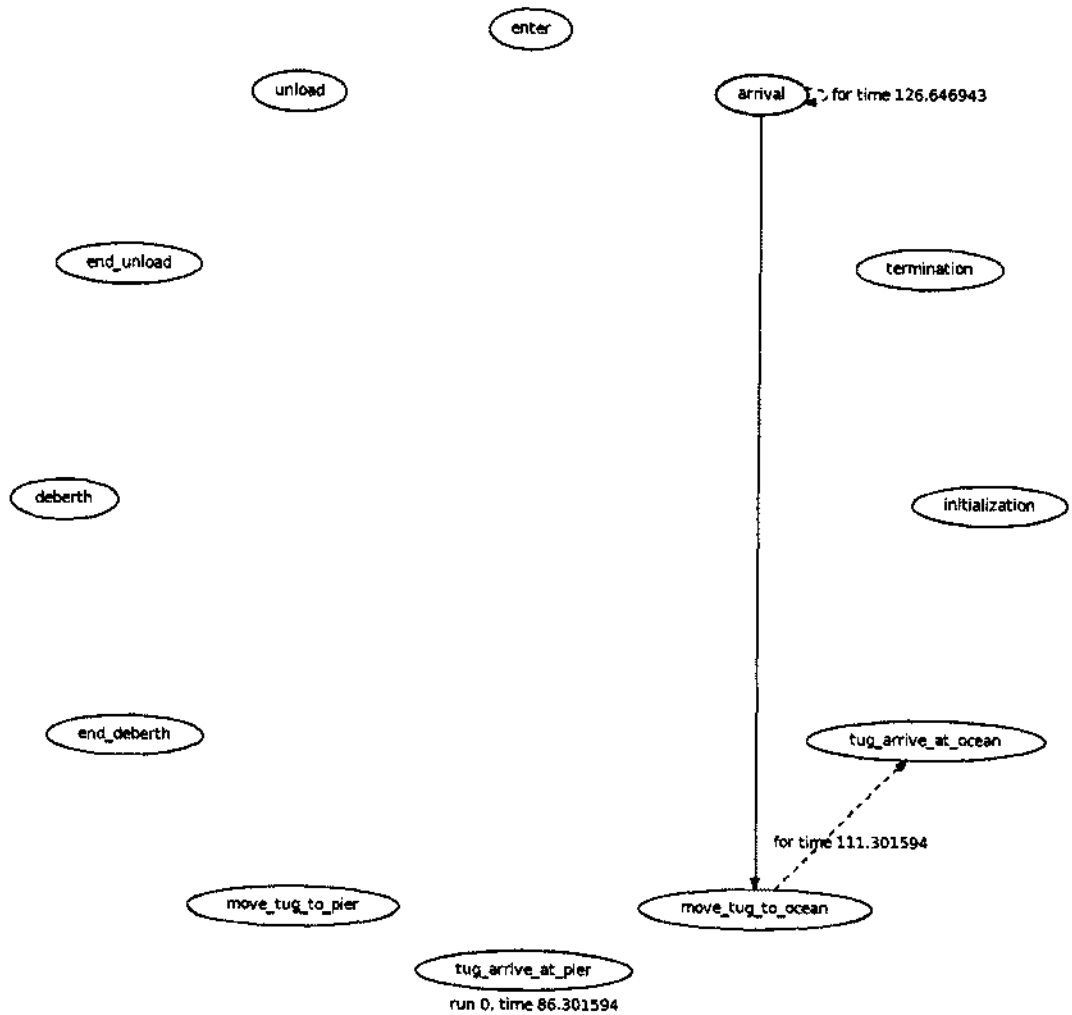


Figure 21. *Generated graph: harbor dynamic action cluster interaction graph flip book, page 2 of 212.*

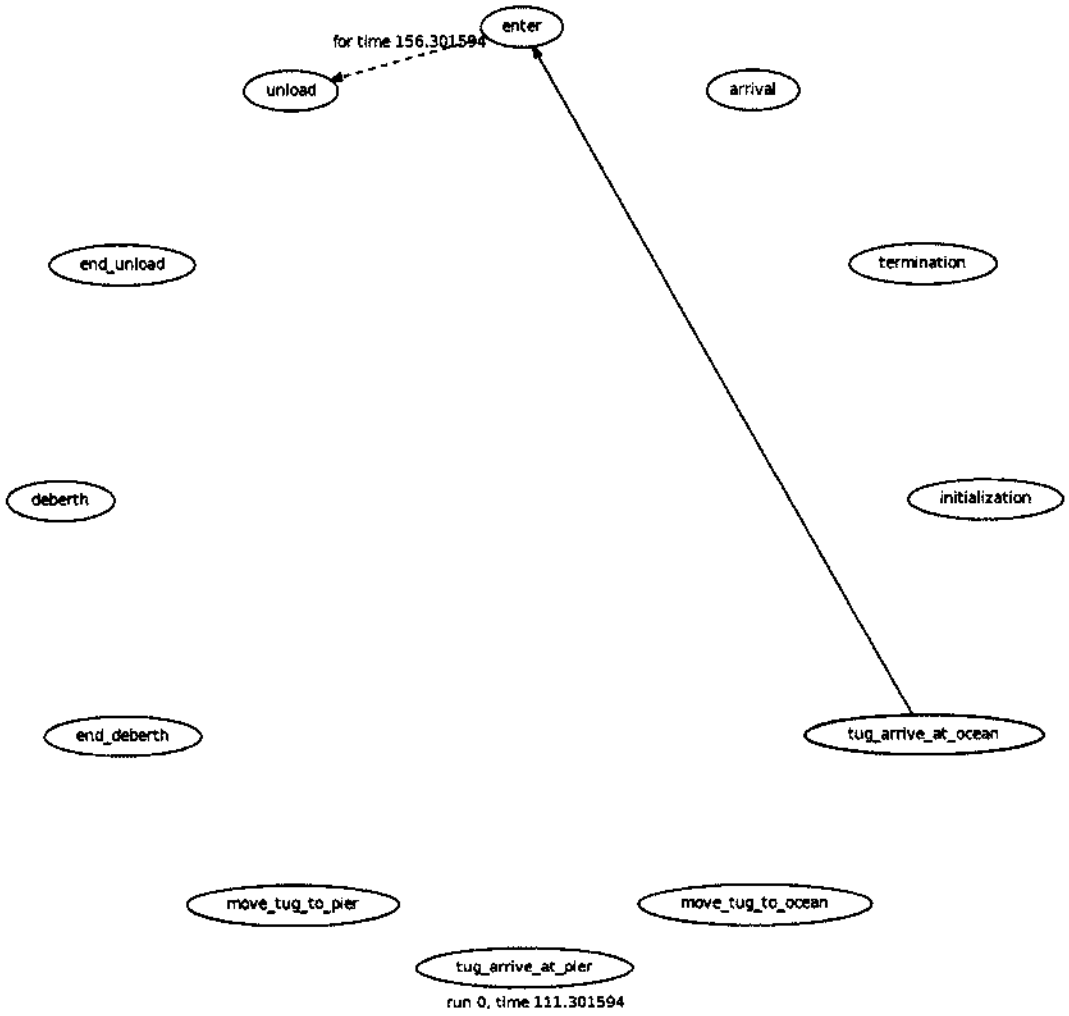


Figure 22. *Generated graph: harbor dynamic action cluster interaction graph flip book, page 3 of 212.*

Figure 23. *Generated graph: harbor dynamic action cluster interaction graph flip book, page 4 of 212.*

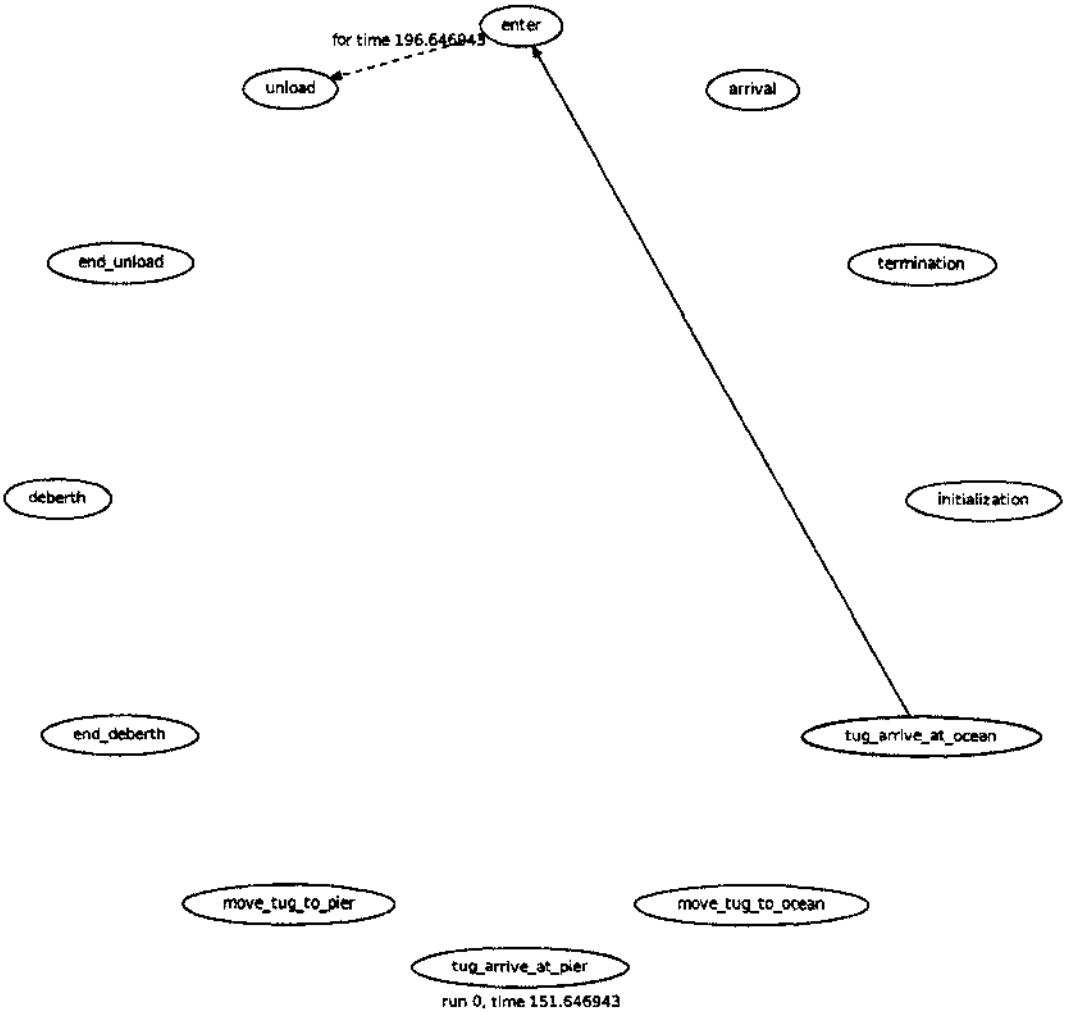


Figure 24. *Generated graph: harbor dynamic action cluster interaction graph flip book, page 5 of 212.*

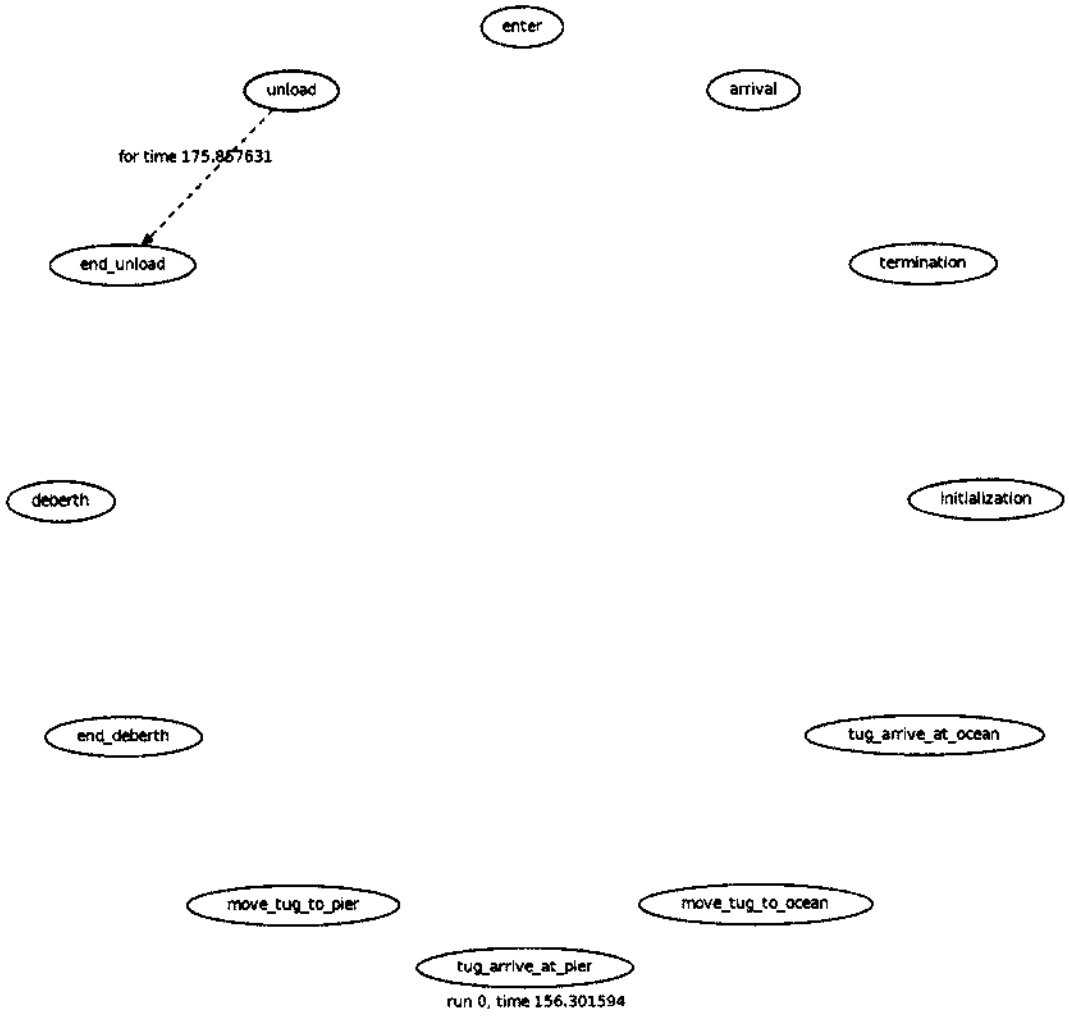


Figure 25. *Generated graph: harbor dynamic action cluster interaction graph flip book, page 6 of 212.*

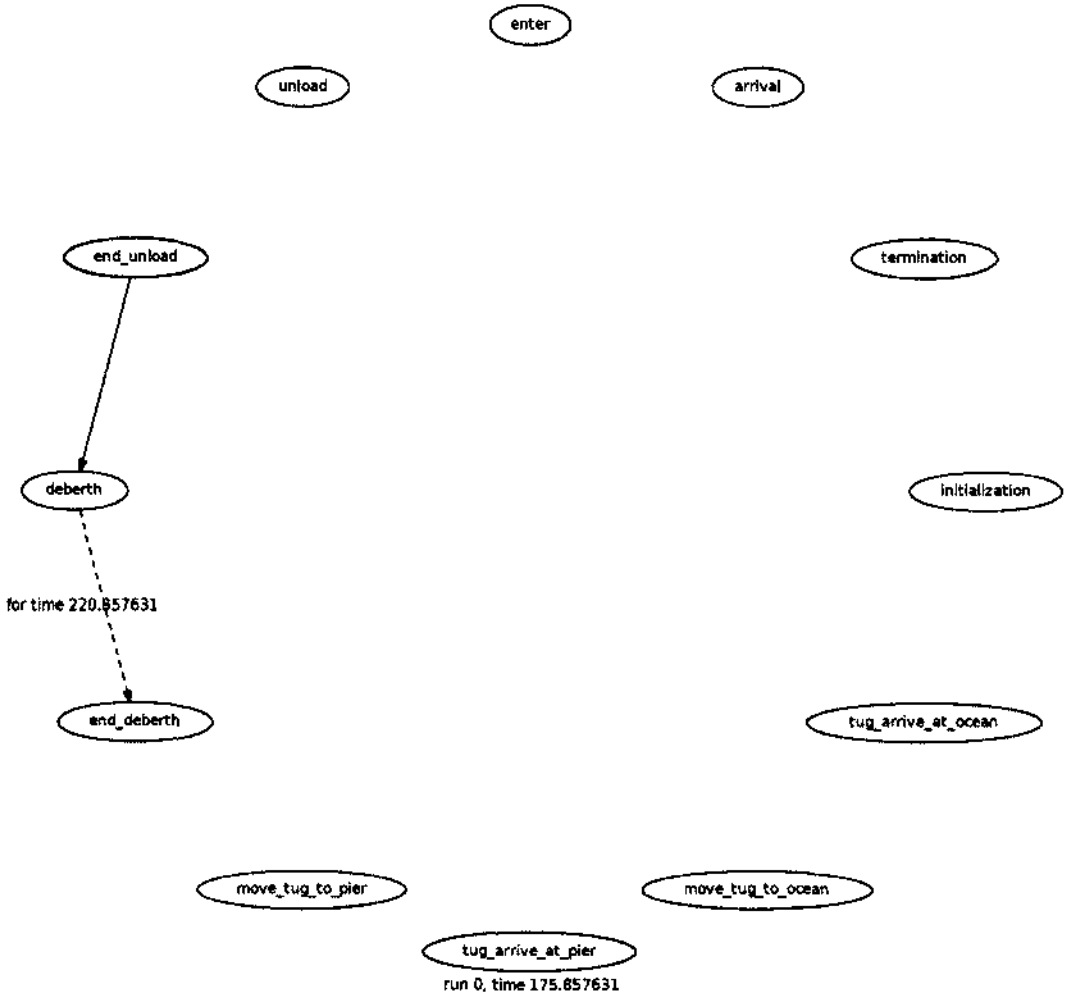


Figure 26. *Generated graph: harbor dynamic action cluster interaction graph flip book, page 7 of 212.*

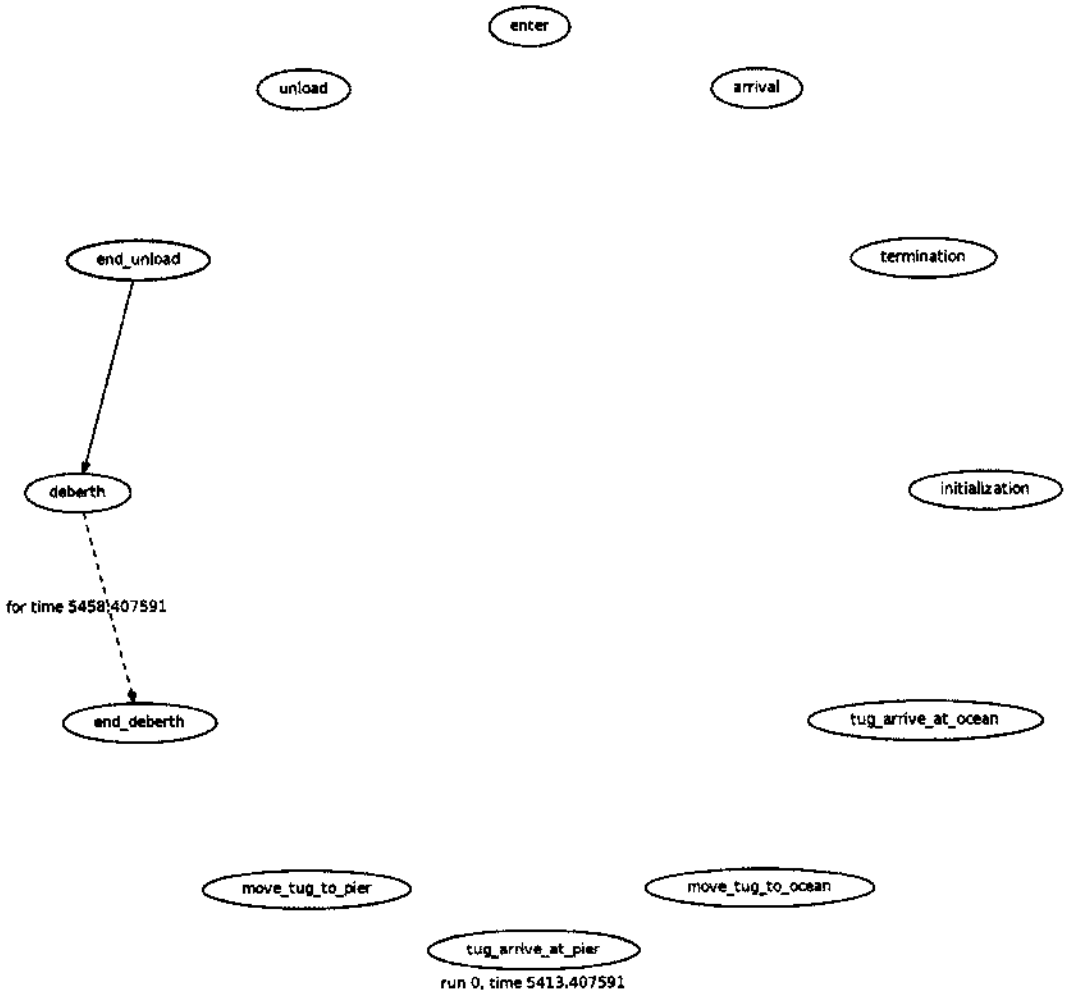


Figure 27. *Generated graph: harbor dynamic action cluster interaction graph flip book, page 211 of 212.*

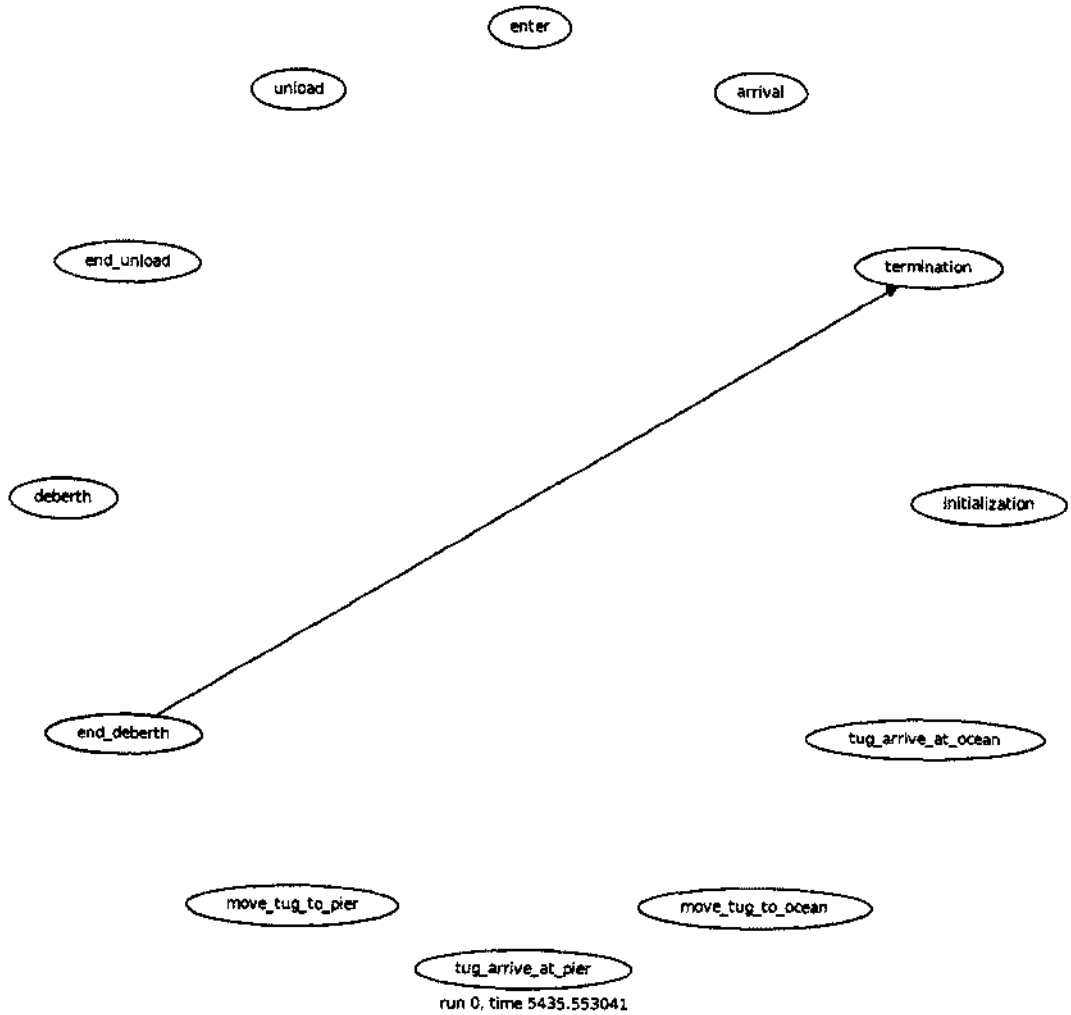


Figure 28. *Generated graph: harbor dynamic action cluster interaction graph flip book, page 212 of 212.*

4.9.3 EXAMPLE: SINGLE SERVER QUEUE

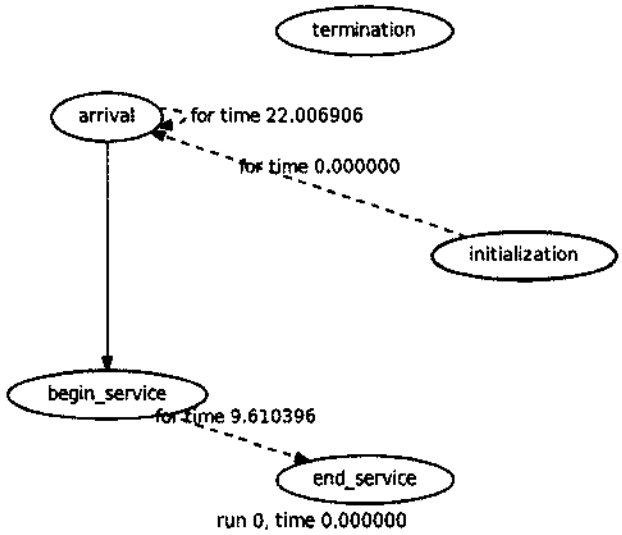


Figure 29. *Generated graph: single server queue dynamic action cluster interaction graph flip book, page 1 of 42.*

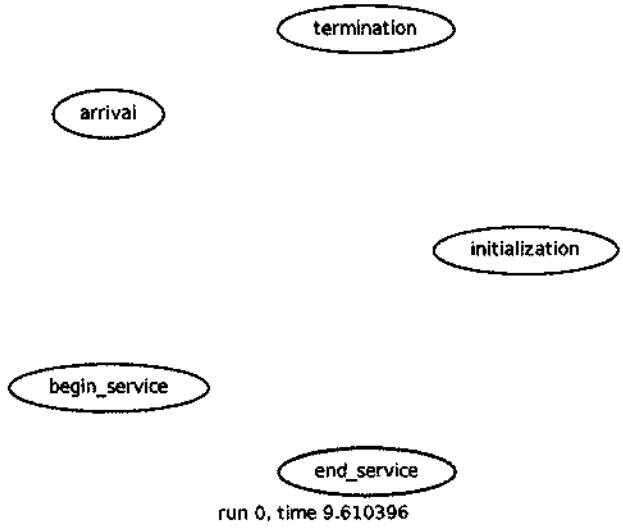


Figure 30. *Generated graph: single server queue dynamic action cluster interaction graph flip book, page 2 of 42.*

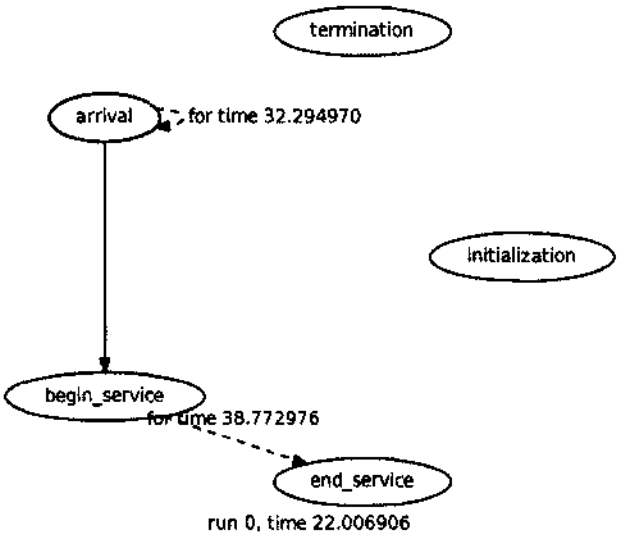


Figure 31. *Generated graph: single server queue dynamic action cluster interaction graph flip book, page 3 of 42.*

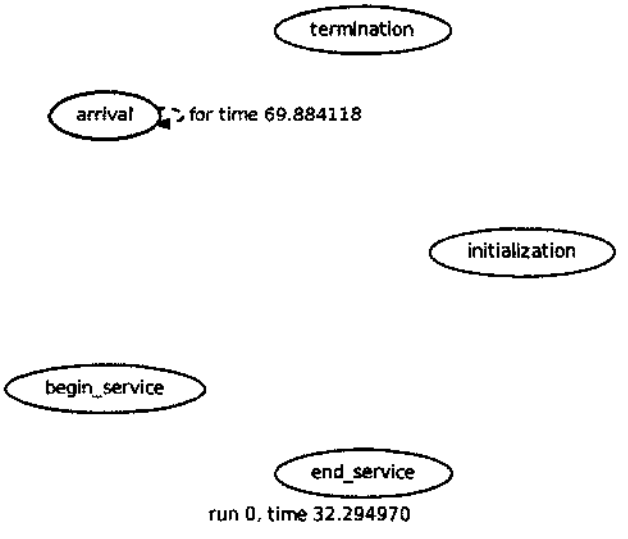


Figure 32. *Generated graph: single server queue dynamic action cluster interaction graph flip book, page 4 of 42.*

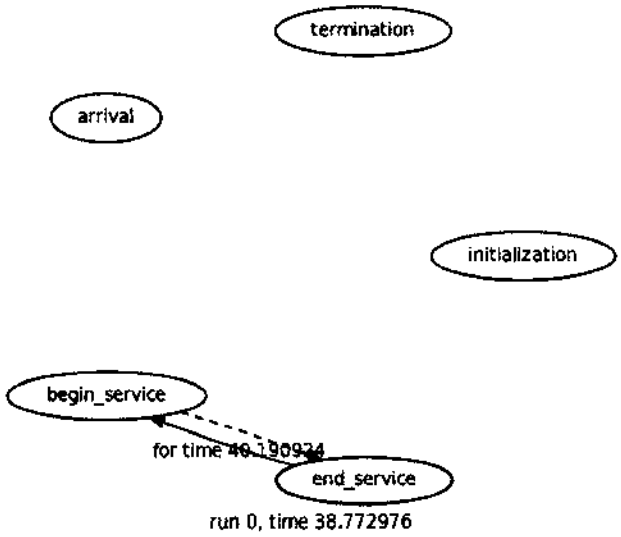


Figure 33. *Generated graph: single server queue dynamic action cluster interaction graph flip book, page 5 of 42.*

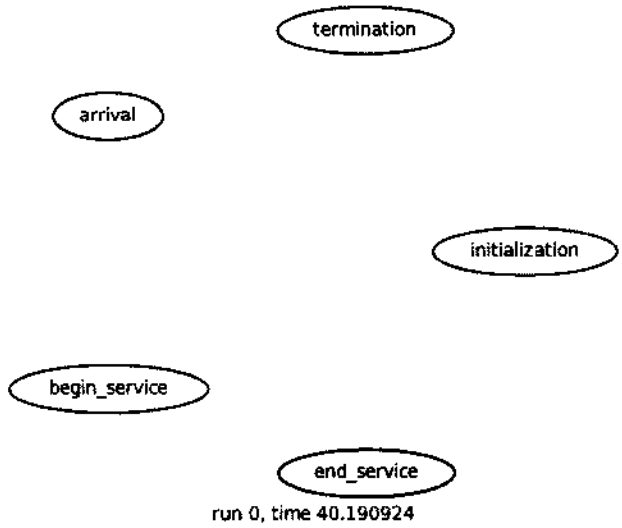


Figure 34. *Generated graph: single server queue dynamic action cluster interaction graph flip book, page 6 of 42.*

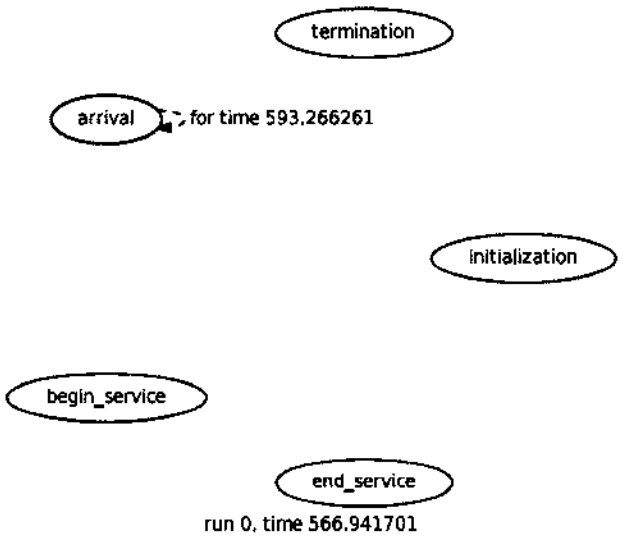


Figure 35. *Generated graph: single server queue dynamic action cluster interaction graph flip book, page 41 of 42.*

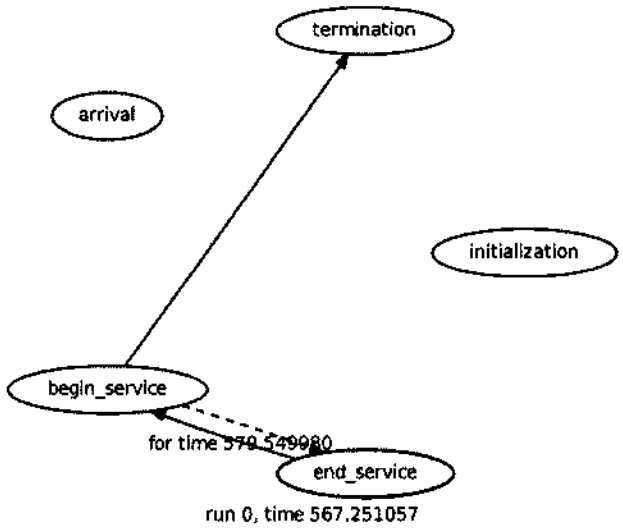


Figure 36. *Generated graph: single server queue dynamic action cluster interaction graph flip book, page 42 of 42.*

CHAPTER 5

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.

Some of the most recent written reviews of this research, from the 2012 Winter Simulation Conference, included, "[i]nteresting work in terms of both the problem assessed and the method proposed," and, "[t]he work promises great practical value."

One of the significant aspects 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.

5.1 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 (`stdout`) of the repairman model in the first part of Section 4.3.1: as with most simulations, it is designed to answer a few specific questions based on modeling objectives. Even if one recompiles to enable this particular simulation's generously verbose debugging/trace output – chosen by the simulation coder (unlikely to be the current model user) for his or her interests, and not always a pre-coded available option anyway – the results still do not help in understanding some key aspects of the model embedded in the simulation code.

Simulation output with trace output enabled:

```
Run 1
initialization AC.                          clock:    0.0
Initial facility failure times
 fac    time
   1   375.4
   2   175.5
   3   641.2
   4   217.2
   5    72.7
   6   178.3
   7     0.0
   8    80.7
   9   151.6
  10   299.1
  11  1374.0
  12  2139.8
failure AC, facility:  7,                   clock:    0.0
travel to facility AC,                      clock:    0.0
  traveling to facility number:  7
  begin repair scheduled for        3.5
begin repair AC, facility:  7,              clock:    3.5
  end repair scheduled for        8.9
end repair AC, facility:  7,                clock:    8.9
  failure scheduled for      252.0
travel to idle AC,                          clock:    8.9
  arrive idle scheduled for        12.4
arrive idle AC,                             clock:   12.4
failure AC, facility:  5,                   clock:   72.7
travel to facility AC,                      clock:   72.7
  traveling to facility number:  5
  begin repair scheduled for       75.2
⋮
```

Events and their scheduling are indeed noted and the output fidelity could be increased easily enough by a programmer (e.g., the first failure actually occurs at time 0.004378, which is not the same as time 0.0 – initialization – indicated above). However, considering the simulation log, 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:

```
⋮
failure AC, facility:  1,                clock: 73611.8
end repair AC, facility:  11,            clock: 73612.5
  failure scheduled for  74336.6
termination AC.                          clock: 73612.5
```

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.

5.2 TRIP LINES

Trip lines are an optional addition of one, 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, a general print statement include 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).

Additionally, consider the output of the harbor simulation of Section 4.3.2: tug utilization and the maximum number of ships waiting at the arrival area. The tug utilization percentage might seem low – the tugs are mostly idle – and the number of

ships waiting at a given time seems reasonable: perhaps one might keep fewer tugs to decrease expenses.

However, adding trip lines (such as those in Section 4.4.2) can reveal that in this simulation, there is a dearth of berths: rather than try to decrease expenses, a more informed choice might be to add an additional berth, thus increasing tug utilization, decreasing ship wait time, and increasing overall profits.

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

5.3 SCHEDULED AND TRIGGERED EVENTS

These 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 of the information, this output omits much from `gcov`-like tools that is unlikely to be of interest to a modeler and instead focuses on *model* behavior.

Similarly, consider Section 4.5.1, 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 4.3.1) – 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.

5.4 EVENT SUMMARIES

Having concise, useful summary information about model components has already revealed model structure in some models that had not been previously recognized.

In a past local simulation study, a modeler was studying trace data produced during simulation executions. It was 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 more importantly, aspects of the

system it represented – that had not been previously recognized: a fundamental insight now easily discernible through these created tools.

5.5 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 text-based output. This additional information about model properties 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 but were created manually; more than one of the manually created graphs, though presented in reviewed, published research, contained errors.

Additionally, while not exemplified in this document, 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.

5.6 TALLIED DYNAMIC ACTION CLUSTER INTERACTION GRAPH

Being able to follow how model components *do* 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 the only the text-based event summaries or the static ACIG, though, is the possibility of superfluous edges and which edges may be such. Given an arbitrary condition specification, the automatic creation of a static ACIG with no redundant or unnecessary edges is unsolvable [23]; 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 extraneous or if this interaction just did not occur during this particular run, enhancing understanding of the model and the system it represents.

5.7 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.

CHAPTER 6

FUTURE RESEARCH DIRECTIONS

A prime problem with model descriptions, whether in textual or graphical notations, is that even for simple models, descriptions are often difficult to fully comprehend. Even in relatively simple cases, the wealth of data available can easily obscure (other) useful information. The overarching goal of this research is to create new possibilities for modelers and model users to understand more about their models and consequently the systems they represent. Any additional ways to filter the information produced by the simulation or obtained about the model that furthers this goal would be useful and encouraged directions.

6.1 CONDITION SPECIFICATION TO DIRECTION EXECUTION OF ACTION CLUSTERS

As mentioned, the tools here assume a direct execution of action clusters style (Section 3.3). A compiler to translate a given condition specification into this C DEAC implementation would be a straightforward and welcome addition to this research, as this would ensure that the simulation code analyzed is of the expected structure for these analysis tools.

6.2 USE IN DETERMINING APPROPRIATENESS OF MODELS

Over the course of many research discussions at conferences, there is a strong interest in using tools such as those created here to determine model or simulation re-use or integration appropriateness. It has been demonstrated that automated model diagnosis supports model verification and validation in the early stages of the model development process, thereby leading to savings in project development time and costs and yielding improvements to overall process quality [4, 30].

6.3 IDENTIFICATION OF POSSIBLE RACE CONDITIONS

A race condition in this context refers to the possibility of different model behaviors occurring if event orders are an accident of implementation technique rather than

determined by the model specification. In a common implementation of a discrete event simulation, an events list is checked before the clock is advanced. Generally, if implemented properly, the order in which events that are scheduled to occur at the same time are posted to the list should not matter to the simulation results; however, analysis can flag this possibility – a potential surprise to modeler.

Note that this is not necessarily a modeling or implementation error but could reflect a property of the system being simulated. While exact identification of race conditions is certainly unsolvable, a reasonable set of possible race conditions could be helpful.

6.4 ADDITIONAL GRAPHICS

Many visuals lend themselves as future extensions to this work for the researcher with a background or interest in graphical programming.

Rather than a flip book, one could create an animated version of the dynamic action cluster interaction graphs, or a version that progresses through the graphs using scroll bars rather than having a multi-page document.

Weinberg identified the importance of program locality [44], 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" [44]. 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 might also change.

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 aforementioned program slices, could aid in allowing model characteristics to be more easily understood.

Similarly, "zoomable" action cluster interaction graphs could be insightful. Large simulations can have hundreds or even thousands of action clusters. Zoomable versions of the action cluster interaction graphs presented here – where one could see only a desired portion of the ACIG – could allow the interactive exploration of a model so that only relevant information is presented. Exploring this graph interactively – perhaps as the simulation progresses, perhaps as the curiosity of the moment changes – also could be insightful.

CHAPTER 7

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.

Different analysis techniques can yield different potential discoveries. 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 (usually code).

Static analysis can often reveal characteristics of a model not readily apparent from observing merely its run-time behavior. No finite number of runs can necessarily discover what is possible; however, from static analysis, one can reveal the possibility of infrequent situations.

From static analysis, one may discover that event A can appear to cause event B, but dynamic analysis often can reveal specifically which events caused which events, 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 that this does not happen, this may be of interest.

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 in examining data produced by simulations during execution. This work has provided both model builders and model users with additional techniques that can give them improved understanding of their models not otherwise available.

The contribution of this research is the creation and presentation 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. A 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. While some of the information presented here could be produced by existing software development tools, most modelers today do not have the technical background to use these tools or to make use of the reports such tools can produce. Continuing, one of the key aspects here 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 continue to increase in contribution.

Automatic tools have been created and demonstrated to reveal new insights into models and the systems they represent.

A simulation log is generated, without any user action or programming effort, that notes each action and the simulation time; an additional printer-friendly version is also created.

"Trip lines" concern any boolean expression of model variables of which the modeler wants to be notified the first time it is passed. Two options are available: a trip line that can be tripped exactly once or one that can be tripped and reset under specified conditions.

A list is generated of all scheduled, unscheduled, triggered, and untriggered events.

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.

The static action cluster interaction graph is generated, showing which events can cause which events.

The dynamic action cluster interaction graph is generated, with edges labeled according to event frequency during a given run.

And, a dynamic action cluster interaction graph is created for every time step of the simulation run and combined in to a flip book, providing a visual representation of the entire simulation run.

The work described here provides modelers with new views of their models; significantly, these views can be generated without additional work or knowledge on the modelers' part. In the simulation community, little work had been done on exploring automatic generation of different views and representations of existing models, especially to the extent presented here. These new techniques can provide additional insights into models and the systems they represent.

REFERENCES

[1] P. ANDERSON, T. W. REPS, T. TEITELBAUM, AND M. ZARNIS, *Tool support for fine-grained software inspection*, IEEE Software, 20 (2003), pp. 42–50.

[2] R. BAECKER, *Enhancing program readability and comprehensibility with tools for program visualization*, in Proceedings of the 10th International Conference on Software Engineering, 1988, pp. 356–366.

[3] O. BALCI, *Verification, validation, and certification of modeling and simulation applications*, in Proceedings of the 2003 Winter Simulation Conference, S. E. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, eds., December 2003, pp. 150–158.

[4] O. BALCI AND R. E. NANCE, *Simulation model development environments: A research prototype*, J. Opl Res. Soc., 38 (1987), pp. 753–763.

[5] K. W. BAUER, B. KOCHAR, AND J. J. TALAVAGE, *Simulation model decomposition by factor analysis*, in Proceedings of the 1985 Winter Simulation Conference, D. T. Gantz, G. C. Blais, and S. L. Solomon, eds., December 1985, pp. 185–188.

[6] J. BOHNET AND J. DÖLLNER, *Visual exploration of function call graphs for feature location in complex software systems*, in SoftVis '06 Proceedings of the 2006 ACM Symposium on Software Visualization, 2006, pp. 95–104.

[7] R. BROOKS, *Towards a theory of the comprehension of computer programs*, Int. J. Man-Mach. Stud., 18 (1983), pp. 543–554.

[8] K. CHEN AND V. RAJLICH, *Case study of feature location using dependence graph*, in Proceedings of ICSM 2000, 2000, pp. 241–249.

[9] T. A. CORBI, *Program understanding: Challenge for the 1990s*, IBM Syst. J., 28 (1989), pp. 294–306.

[10] D. R. COX AND W. L. SMITH, *Queues*, Methuen & Co., London, 1961.

[11] T. EISENBARTH, R. KOSCHKE, AND D. SIMON, *Locating features in source code*, IEEE Trans. Softw. Eng., 29 (2003), pp. 210–224.

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

[13] G. Gordon, *The development of the General Purpose Simulation System (GPSS)*, ACM SIGPLAN Notices, 13 (1978), pp. 183–198.

[14] K. J. Healy, *The use of event graphs in simulation modeling instruction*, in Proceedings of the 1993 Winter Simulation Conference, G. W. Evans, M. Mollaghasemi, E. C. Russell, and W. E. Biles, eds., December 1993, pp. 1131–1134.

[15] M. H. Hwang and B. P. Zeigler, *A modular verification framework based on finite & deterministic DEVS*, in Proceedings of the 2006 DEVS Integrative M&S Symposium, 2006, pp. 57–65.

[16] IEEE International Conference on Program Comprehension. `http://www.program-comprehension.org`.

[17] D. F. Jerding and S. Rugaber, *Using visualization for architectural localization and extraction*, in Proceedings of the Fourth Working Conference on Reverse Engineering, 1997, pp. 56–65.

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

[19] R. Koschke and J. Quante, *On dynamic feature location*, in ASE '05 Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, 2005, pp. 86–95.

[20] D. Kranzlmüller, S. Grabner, and J. Volkert, *Event graph visualization for debugging large applications*, in Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT96), J. Francioni and D. Reed, eds., 1996, pp. 108–117.

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

[22] R. E. Nance and C. M. Overstreet, *Exploring the forms of a model diagnosis in a simulation support environment*, in Proceedings of the 1987 Winter

Simulation Conference, A. Thesen, H. Grant, and W. D. Kelton, eds., December 1987, pp. 590–596.

[23] 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.

[24] C. M. OVERSTREET AND I. B. LEVINSTEIN, *Enhancing understanding of model behavior through collaborative interactions*, in Operational Research Society (UK) Simulation Study Group Two Day Workshop Proceedings, S. C. Brailsford, L. Oakshott, S. Robinson, and S. J. E. Taylor, eds., March 2004, pp. 11–17.

[25] C. M. OVERSTREET, E. H. PAGE, AND R. E. NANCE, *Model diagnosis using the condition specification: From conceptualization to implementation*, in Proceedings of the 1994 Winter Simulation Conference, J. D. Tew, M. S. Manivannan, D. A. Sadowski, and A. F. Seila, eds., December 1994, pp. 566–573.

[26] E. H. PAGE, *The condition specification: Revisiting its role within a hierarchy of simulation model specifications*, SIGSIM Simul. Dig., 22 (1993), pp. 11–33.

[27] E. H. PAGE AND R. E. NANCE, *Incorporating support for model execution within the condition specification*, Trans. Soc. Comput. Simul. Int., 16 (1999), pp. 47–62.

[28] 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., December 2005, pp. 547–554.

[29] R. J. PAUL AND J. KULJIS, *Problem solving, model solving, or what?*, in Proceedings of the 2010 Winter Simulation Conference, B. Johansson, S. Jain, J. R. Montoya-Torres, J. C. Hugan, and E. Yücesan, eds., December 2010, pp. 353–358.

[30] F. A. PUTHOFF, *The model analyzer: prototyping the diagnosis of discrete-event simulation model specifications*, Master's thesis, Virginia Polytechnic Institute and State University, September 1991.

[31] C. RAMCHANDANI, *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets*, PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, September 1973.

[32] M. P. ROBILLARD, *Automatic generation of suggestions for program investigation*, ACM SIGSOFT, 30 (2005), pp. 11–20.

[33] T. M. ROEDER AND L. W. SCHRUBEN, *Information models for queueing system simulation*, ACM Trans. Model. Comput. Simul., 20 (2010), pp. 8:1–8:34.

[34] R. G. SARGENT, *The use of graphical models in model validation*, in Proceedings of the 1986 Winter Simulation Conference, J. R. Wilson, J. O. Henriksen, and S. D. Roberts, eds., December 1986, pp. 237–241.

[35] N. SASIREKHA, A. E. ROBERT, AND M. HEMALATHA, *Program slicing techniques and its applications*, Int. J. Softw. Eng. Appl., 2 (2011), pp. 50–64.

[36] T. J. SCHRIBER, *Simulation Using GPSS*, John Wiley & Sons, New York, NY, 1974.

[37] L. W. SCHRUBEN, *Industrial Engineering & Operations Research, University of California, Berkeley.* `http://ieor.berkeley.edu/Research/Projects/simulationGroup.htm`.

[38] ——, *Simulation modeling with event graphs*, Commun. ACM, 26 (1983), pp. 957–963.

[39] L. W. SCHRUBEN AND E. YÜCESAN, *Simulation graphs*, in Proceedings of the 1988 Winter Simulation Conference, M. A. Abrams, P. L. Haigh, and J. C. Comfort, eds., December 1988, pp. 504–508.

[40] SIMIO, *Newsroom: Simio chosen for Naval Postgraduate School project.* `http://www.simio.com/case-studies/NPS`.

[41] P. B. SOUTHARD, C. CHANDRA, AND S. KUMAR, *RFID in healthcare: A Six Sigma DMAIC and simulation case study*, Int. J. Health Care Qual. Assur., 25 (2012), pp. 291–321.

[42] J. F. SOWA, *Processes and causality.* `http://www.jfsowa.com/ontology/causal.htm`.

[43] M.-A. STOREY, *Theories, methods and tools in program comprehension: Past, present and future*, in Proceedings of the 13th International Workshop on Program Comprehension (IWPC'05), 2005, pp. 181–191.

[44] G. M. WEINBERG, *The Psychology of Computer Programming*, Computer Science Series, Van Nostrand Reinhold Company, New York, NY, 1971.

[45] M. WEISER, *Program slicing*, IEEE Trans. Softw. Eng., SE-10 (1984), pp. 352–357.

[46] WIKIPEDIA. *Petri net*, `http://en.wikipedia.org/wiki/Petri_nets`.

[47] N. WILDE AND M. C. SCULLY, *Software reconnaissance: Mapping program features to code*, J. Softw. Maint.: Res. Pract., 7 (1995), pp. 49–62.

[48] B. P. ZEIGLER, *Theory of Modeling and Simulation*, John Wiley & Sons, New York, NY, 1976.

[49] B. P. ZEIGLER, H. PRAEHOFER, AND T. G. KIM, *Theory of Modeling and Simulation*, Academic Press, San Diego, CA, second ed., 2000.

VITA

Kara Ann Olson
Department of Computer Science
Old Dominion University
Norfolk, VA 23529

Old Dominion University, Norfolk, VA

- Master of Science, Computer Science, May 2007
- Bachelor of Science in Computer Science with Honors Distinction, Minor in English, Magna Cum Laude, May 1997
- Bachelor of Science, Mathematics, Magna Cum Laude, May 1997

Selected Related Presentations

- "Enhancing Understanding of Discrete Event Simulation Models Through Analysis," 2014 Winter Simulation Conference, Savannah, GA, December 7–10, 2014; with C. Michael Overstreet
- "A Forthcoming Useful Tool: Enhancing Understanding of Models Through Analysis," ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, Montreal, Canada, May 19–22, 2013; and 2012 Winter Simulation Conference, Berlin, Germany, December 9–12, 2012; with C. Michael Overstreet
- "Enhancing Understanding of Models Through Analysis," 1st International Conference on Simulation and Modeling Methodologies, Technologies and Applications, Noordwijkerhout, The Netherlands, July 29–31, 2011; with C. Michael Overstreet; *with publication*
- "Enhancing System Understanding Through Model Analysis," Culture and Computer Science VII – Serious Games, Berlin, Germany, May 14–15, 2009; with C. Michael Overstreet and E. Joseph Derrick; *with publication*
- "Enhancing Model Understanding Using CS-XML," Operational Research Society 4th Simulation Workshop, Worcestershire, England, April 1–2, 2008; with C. Michael Overstreet and E. Joseph Derrick; *with publication*
- "Code Analysis and CS–XML," 2007 Winter Simulation Conference, Washington, DC, December 9–12, 2007; with C. Michael Overstreet and E. Joseph Derrick; *with publication*