Report Generation for Simulation Traces with Traviando

Peter Kemper
College of William and Mary
Department of Computer Science
Williamsburg, VA 23187, USA
kemper@cs.wm.edu

Abstract

Any model-based evaluation of the dependability of a system requires validation and verification to justify that its results are meaningful. Modern modeling frameworks enable us to create and evaluate models of great complexity. However, we believe that much more can be done to support a modeler in ensuring that the dynamic behavior of an executable simulation model is consistent with the modeler's understanding. In this paper, we describe a new command line version of Traviando that reads an execution trace of a discrete event simulation and generates a set of HTML formatted web pages to document properties that it recognizes from its input. Those properties include characteristics of state variables as well as changes to state variables that are performed by events. The point is to highlight the content of a simulation run in a format that is immediately accessible and understandable.

1. Introduction

Stochastic discrete event dynamic system (DEDS) models have been in use for many years to conduct a dependability or performance assessment of a system under study. Although other solution techniques for particular classes of models exist, discrete event simulation is the most commonly applied technique due to its few constraints and broad applicability. A simulation executes a given model and derives values for dependability and performance measures based on a statistical evaluation of the observed dynamic behavior. This a very low level technique compared to how models are specified in modeling formalisms supported in modeling tools and most tools hide such details from a user and come up with results at a level that corresponds to the abstraction level used in modeling. This is appropriate if a given model is correct. It is attributed to George Box, that all models are wrong, but some are useful. So it is important to justify confidence in the usefulness of the considered model and this justification is with what is commonly termed validation and verification (V&V) of a model [9]. Validation relates to the question ”Did we build the right system?” while verification addresses the transformation of a conceptual model into an implemented, executable model with the question ”Did we build the system right?”. A valid and verified model is expected to provide reasonably accurate values for measures of interest that are consistent with what would be observed for the system of interest. V&V is a classic topic in simulation and the Winter Simulation Conference devotes a tutorial paper to it in each year, often by Balci [3] and Sargent [9]. In particular for model verification, Sargent [9] names structured walk-throughs, correctness proofs and examining the structure properties of a simulation program for static ”testing” techniques. For dynamic testing he names traces, investigations on input-output relations, internal consistency checks, and reprogramming critical components to determine if same results are obtained. Such steps are embedded in an overall procedure of model V&V.

In this paper, we focus on tool support for a verification that uses dynamic testing and traces, that is rather experimental, and that cannot guarantee absence of errors with its results. However, on the positive side, it provides feedback on truly observed model behavior as documented in a simulation trace. The tool bridges the gap between the details of a simulation run and a high-level model specification in a complementary manner to measurement results reported by a simulator itself and pseudo-realistic animations of the simulated system which modern simulation frameworks often provide. We report on a recent extension of Traviando [8] by a new feature that generates a set of HTML formatted web pages which report observations and list warnings. With the generation of this list of warnings, we attempt to accumulate ”lessons learned” experiences such that common errors and pitfalls are detected. The approach is stimulated by the way software development is supported by static code analysis, where a software tool generates a report for a given program code that lists locations in the code base that require further attention based on a set of rules that incorporate knowledge on common pitfalls in programming; see FindBugs as a particular example [7]. In dependability modeling, there is no one (or few) common main stream model notation as it is the case in programming, but there is a common ground for the execution of stochastic DEDS models as a state transition system. Möbius [4] is an example of a modeling framework based on the concept of a state transition system, which allows a modeler to compose models specified in different formalisms but all share a common notion of what is a state and a state transformation by an event. So, we put forward a simulation execution checker based on observations made from a trace of states and transitions of a stochastic DEDS model.

In the area of business process modeling, there exists related work that goes a step further and attempts to identify a business process model based on a log file of a workflow. To do so, the workflow log contains information on cases and their corresponding tasks that are logged with a time stamp. Van der Aalst and coworkers have developed a rich body of knowledge for similar concerns in business process modeling and workflows, e.g. the identification of models from observed behavior, detection of anomalies, equivalence of models based on observes to name a few. E.g. in [10], Weijters and Maruster discuss for which kinds of workflow models it is possible to rediscover a corresponding, equivalent model. This work is very promising for any application where events correspond to entities, e.g., as for simulation models that follow a process interaction approach. For the work presented here, the trace data gives information on states and events, where in particular the state information allows us to deduce other properties, e.g., the identification of cyclic behavior. So we focus on a slightly different type of information and consequently focus on different properties, in particular properties of states and regularities in traversing the state space of a model. With respect to model identification, we only conduct an invariant analysis similar to classical Petri net invariant analysis for a matching subset of state variables and actions.

The rest of the paper is structured as follows: Section 2 outlines the tool architecture. Section 3 describes the content of the generated web pages. Section 4 shows how to identify an error in a model with the information provided in the generated report. We conclude in Section 5.

2. Tool Architecture

The command line version of Traviando [2] has a minimalistic user interface that outputs a report as a set of HTML-formatted files for an XML-formatted trace given as input. In this section, we outline input, output and internal structure of Traviando.

Input. An input trace file consists of a prefix and a sequence part. The prefix part contains information on all state variables and actions that perform changes to state variables. State variables can be partitioned into subsets and each subset corresponds to a so-called process. An action performs changes to value settings of state variables. So actions can be partitioned into local actions that change only values of variables of a single process and so-called interactions that change values of variables of more than one process. The sequence part of the trace consists of an initial state that assigns values to all state variables and a sequence of events, each described by an action identifier, a time stamp (optional) and value assignments to state variables which yield the successor state. We assume that variables that are not explicitly assigned a new value remain unchanged (a common inertia rule). The file format is XML, for a more detailed description and a document type definition (DTD), see [2]. This input format has been successfully used for simulation traces generated with Möbius [4], NS2 [1], ProC/B toolset [6] and the APNN toolbox [5].

Output. Traviando scans the trace and generates a set of web pages with a main webpage that provides an outline with sections on variables, actions, invariants and warnings. The list of warnings also contains links to static webpages which provide a generic and detailed description of the rationale, symptoms and solutions for any particular warning. We elaborate on the generated content in Section 3.

Architecture. Traviando's command line version is implemented in Java 1.5 and shares code with the full, interactive version of Traviando, an interactive trace analyzer and visualizer. The software parses a trace file (with the help of a SAX parser) and creates an internal representation in the size of the trace and conducts a series of analysis steps. Any graphics are in PNG format and generated with the jFreeChart library. The code is developed and tested in a MacOS and a Linux environment. The command-line interface of Traviando minimizes the learning curve to apply it to a simulation trace. Since knowledge on web browsing is common, user appreciation of Traviando relies mainly on the quality of the generated content, its presentation on web pages, and the scalability of internal algorithms to apply to lengthy simulation runs. Figure 1 outlines the tool's architecture. Traviando makes use of a set of libraries including a parser support (SAX), graphics (jfreechart), and statistics (commons-math).

Difference Between Traviando's Command-line and GUI version. The new command-line interface draws on a newly implemented classification package that provides functionality to characterize variables and actions and to detect anomalies. New functionality includes a basic type recognition of variables, a check for variables being unchanged or actions not being performed towards the end of a trace which may suggest a deadlock situation, a check for

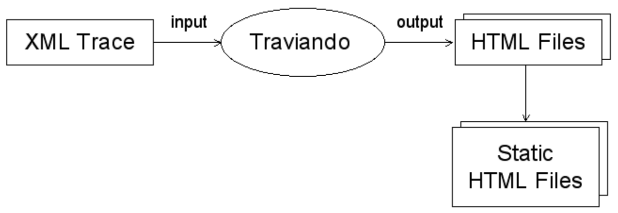

Figure 1. Commandline version of Traviando

actions performing state transformations to overflow a numerical range for integer type variables, the identification of state transformations performed by actions and a detection of invariants for actions and variables, the generation of a list of warnings to help a user recognize particular aspects of the observed behavior, to name a few. This new functionality has also been made available in the interactive GUI of Traviando but in a different format. In providing two different ways to access the information we extract from a trace, we want to explore which works better in practice. There is a clear difference on what the generated HTML report can provide compared to the interactive GUI. The report is strong at providing a firsthand feedback with figures, statistics, and warnings across the overall trace, but not to visualize particular states or locations in the trace. The latter is a strong point of the interactive GUI which provides a model-checker and a trace browser among other features to locate areas of interest in a trace which are then visualized graphically in the style of a message sequence chart.

3. Generated Content

In this section, we describe the structure and content of the generated web report.

Main Page. The main page provides some aggregated information and links to other pages. It reports the total number of variables and actions as well as the number of events in the trace. It contains a table that lists the number of local variables, local actions and interactions for each process. The numerical entries carry a link to web pages that provide further information, e.g., the number of local actions of a particular process links to a page with further information for all local actions of that process. The page also provides generated text passages to explain the content of tables and figures. The main page further shows a bar chart with the total number of changes being made by the events in the trace. This helps in recognizing variables that are changed very often or very rarely, as well as groups of variables that are changed in a similar manner. A second chart shows how often each action occurs, which allows a user to recognize extremes and patterns as for the previous figure on the number of value assignments. The third and the fourth figure show characteristics of how the simulation run visits states in the state space of the model. The third figure associates an integer index $1, 2, \ldots$ to the sequence of states in an increasing order, such that there is a 1-1 mapping between states and indices. Fig. 2 shows an example; it shows the state index as a function of the position in the trace. This plot indicates if the simulation returns to previously visited states in a regular manner (which simulation models often do). The fourth and last figure shows how the length of any prefix of the simulation run evolves if cycles are removed. This is called ”progress” in [8] and illustrates in a different manner if the simulation run returns to previously visited states. Plots of the progress measure show certain patterns for certain problematic situations, see [8] for further details.

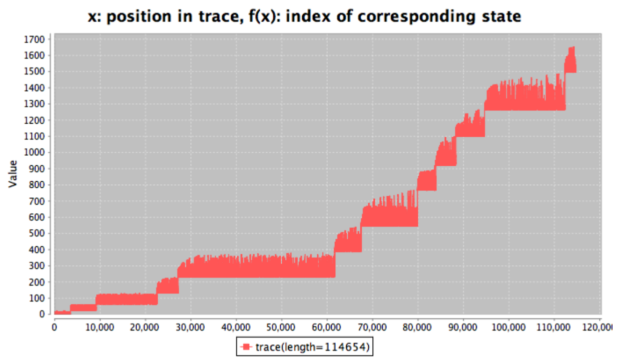

Figure 2. States visited by a trace of an example server model

Finally the page lists links to six further sections (pages), that give details 1) for all variables, 2) for all actions, 3) for action invariants observed in the trace, 4) for action invariants calculated from an invariant analysis, 5) for variable invariants in a similar manner, and 6) a list of warnings. There is also an individual page for each variable and action that includes characteristics and warnings. We will summarize the contents of those pages in the following paragraphs.

Variables. A state is described by the value settings of state variables. For state variables, certain characteristics are straightforward to obtain from a simulation trace, yet can provide useful feedback. For state variables that are numbers (integer, floating point), we can consider the range of values and the sequence of value assignments. We identified three types of variables:

- constants: variables that are initialized and never modified. Such a variable may hold a parameter value of a parameterized model in a series of experiments.
- state: variables that change in value, often on a discrete, integer domain. Such a variable may encode the

state of a component of a system.

- counters: variables that are monotonously non-decreasing (non-increasing). Such a variable may encode a way to obtain measurement data from a simulation run.

Of course, variables can turn out to be in one of the categories due to missing or faulty state transformations. So, we consider it useful to provide the range of values, statistical values of mean, mode and variance as well as information on which action performs what kind of state transformation. If variables do change in value but not in a monotonous manner, it is interesting to obtain more detailed information on the distribution of values as well as to see if there is a trend in the sequence of values; for the latter, a detailed diagram and the aggregated information of a linear regression are provided. For regular behavior, we assume that the slope of the regression is close to zero and that the frequency with which the variable gets modified shows little variation. For each variable, we report on which actions perform changes to its values and in which manner. We also check, if a variable is rarely changed or frequently changed and if after an initial period of changes those die out and the variable remains constant for the rest of the trace. The latter can indicate a deadlocking situation, if actions that have made changes to that variable die out as well.

Actions. For each action, we want to provide feedback about which states enable this action and what state transformations are observed. This feedback helps a modeler to recognize faulty action specifications. The first issue is to recognize actions that are declared in the prefix but never occur in the trace, which may indicate dead model components or indicate that the trace is too short to represent the dynamic behavior of a model completely. For those actions that are indeed present in the trace, we are interested in a characterization of states where they can occur (enabling conditions) and what changes an action causes to which state variable (state transformation function). We try to fit the transformation that an action a performs for a numerical variable with current value v and resulting value v' as a linear function $v' = b \cdot v + c$. Linear transformations are often seen for encodings of automata, in particular for Petri nets. If a linear function does not fit, we consider the function as dependent on the current state s, such that $v' = v + \delta(v, a, s)$, which works for all deterministic state transformations. In those cases, we report how often a increases, decreases, or does not change the value of v as well as the range of values seen for $\delta(v, a, *)$. One possible reason for a state dependent function may actually be a range overflow of an integer-valued variable. We check this by looking at states preceding action a and detect the one with the highest value seen for v. Given that value, we see if we can exceed a common integer range threshold (e.g., 2^8 for unsigned short, 2^{32} for unsigned integer) by adding a possible value seen for $\delta(v, a, s)$. In a similar manner, we check for underflow for signed and unsigned short and integer variables. By checking all predecessor and successor states of events that perform a particular action, we can also detect if the action shows a non-deterministic behavior (same action yields different successor states for equal predecessor states). As for variables, we check if occurrences of an action are seen only for some initial phase of the trace, which can indicate a deadlock situation.

Immediate actions. A discrete event simulator schedules events according to its time stamps. In practice, it is possible that multiple events get scheduled for a single point of time. Examples for this scenario are discrete time steps, actions that have particular deterministic, discrete delays, or actions that model immediate reactions to state changes triggered by some other actions. It is often difficult for a modeler to evaluate how a simulator schedules multiple events at a single point of time. Documentation may not be detailed and specific enough to cover all cases or may not be up to date and in sync with the simulator code.

In any case, a sequence of events with the same time stamps can be easily recognized in a given trace. We identify those sequences and report the triggering actions, provide a set of immediate successor actions and the length of such subsequences. However, for an investigation of full details, the interactive use of Traviando with its event browser and its trace visualization with message sequence charts is more suitable than what can be provided in a generated HTML report.

Invariants. So far, we have considered properties of individual variables or actions. However, models often carry invariants like a constant number of customers in a closed queueing network or that particular sets of actions may cause the simulation to return to a state, e.g., a failure and repair of a subsystem. We aim to detect some of these invariants of a model as well. We consider invariants of two kinds, namely (1) action invariants that describe occurrence counts for sequences of actions that describe a cyclic subsequence in a trace, i.e., a sequence of events that leads back to its starting state, and (2) state (variable) invariants, i.e., weighted sums of numerical state variables that remain constant throughout a trace. The presence or absence of an invariant provides useful feedback to a modeler. For example, a particular component like a resource goes through different levels of utilization and operational modes, which often constitutes a state invariant if there are state variables that account for how many components are in which operational mode and an action invariant accounting for all actions that take place between two states where the resource is idle and operational. If those expected invariants are not seen, a modeler may want to check all involved actions and their state transformations to identify the reason for an absence of the expected regular behavior.

Invariants can be obtained in two ways, either by observing and checking invariants throughout the trace or, for a subclass of models, by identifying an underlying vector addition system that allows us to apply traditional invariant analysis known for Petri nets. For actions, both ways are currently supported; for variables, only the latter is currently available. From exercising a set of traces from a variety of example models, we have seen that sets of invariants tend to be large, such that we decided to restrict the set of observed invariants that are reported on a web page to those vectors that are linearly independent.

For a modeler, it is of interest to obtain a list of invariants as well as an enumeration of all actions (all variables) that are not covered by any invariant (whose computation is immediate). This information is provided on the same page with some further textual explanations and the list of invariants such that the *search* feature of a web browser make it straightforward to identify invariants that cover a particular action (or variable).

Warnings. While the generated content we discussed so far gives guidance to a model on the dynamic behavior of a discrete event model as observed in a simulation run, we also generate a list of warnings. Conceptually, the warnings are of two kinds. The main category is for warnings that describe anomalies detected for certain actions, variables and combinations thereof. The second category consists of warnings that indicate difficulties the analysis techniques can face, e.g., the fitting of a state transformation function to a linear function is based on a single occurrence of an action, which is not enough data, or if a trace may be too short to observe all actions, such that a lot of actions are not present and detected as potentially dead code.

Each warning carries a link to a corresponding external and static webpage that documents the underlying rule that raises the warning, its rationale, symptoms, diagnosis and therapy as a concrete advice on how to fix the problem.

4. Example

In this section, we briefly discuss how to detect an error in the specification of an action in a simple queueing model with two customer classes, where the server is subject to failures and repair. The model is described with Möbius and we consider a generated trace of 113,654 events. The model is taken from [8], where it is described how the progress measure of the trace points us to the error. The new version of Traviando generates a report of 73 webpages for this model of 6 processes with a total of 11 variables and 12 actions. The purpose of the report is to give guidance to a modeler if something is wrong and what that can be. It provides multiple clues that help to recognize the faulty state transformation including the results of [8]. Fig. 2 shows the sequence of states reached in the trace, which indicates that certain events seem to prohibit the simulation to return to previous states which is not as expected. The figure for the progress measure gives a similar message. If we check

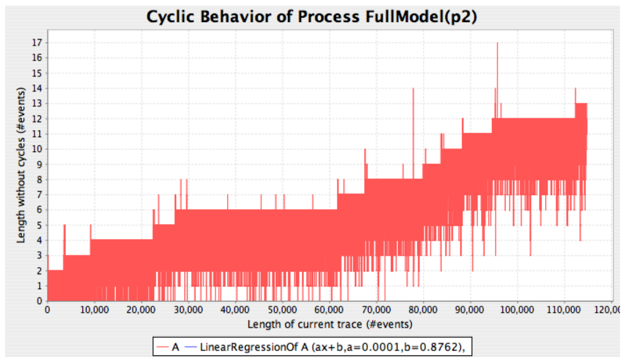

Figure 3. Progress measured for a single process of the server model

the page for all variables, we see the same stepwise function for the progress of variables in process *FullModel* as in Fig. 3. The name of the process should not be understood as being the complete server model; this process corresponds to the top-level Join node in the Möbius model where four variables are shared that are seen as variables of this process in Traviando. The webpage reports for each variable the first and last events that make assignments to it and, if its type is numerical, the range of values observed. For variables corresponding to one customer class, we can easily detect that there are more customers than specified since the range of values is reported as $\{0, 1, ..., 12\}$ instead of $\{0, 1\}$. Following a link for one of those variables, namely *C1WaitsForServer* gets us to the variable's detailed page which informs us how the value of that variable increases throughout the trace, which actions perform which state transformation to the variable (as given in Table 1) and finally a warning that this variable is not an element of any variable invariant computed for this trace. By checking the list of actions that increase the value of this variable, we see that an action *C2RestartIfFailed* occurs 11 times and increments the variable. This tells us that this action which is supposed to deal with customers of class 2 is faulty. Although at this point, it is clear where the problem comes from, we can also follow the link to the detailed page of that action and check what else this action does. We see that the state transformations performed with other variables are all as expected, only the one to *C1WaitsForServer* is incorrect.

Action	Occurrences	Transformation
C1RestartIfFailed(a7)	18	v'=v+1
C2RestartIfFailed(a8)	11	v'=v+1
C1StartService(a9)	19340	v'=v-1
C1StopThinking(a17)	19323	v'=v+1

Table 1. Table of actions that change variable *C1WaitsForServer* as given in the report

Alternatively, we can check results of the invariant analysis: the page on observed action invariants lists four invariants and informs us that action *C2RestartIfFailed* is the only that is not present in any invariant. Again, this points us check this particular action. Action invariants computed from a Petri-net type invariant analysis give a similar result. The variable invariants computed with invariant analysis tell us that three variables are not covered by any invariant, namely *C1WaitsForServer*, *C1WaitsForUser*, and *C1Thinking* which are all state variables modeling the customer class that does not have a constant population of customers. The computed three variable invariants, on the other hand, confirm that the server is either in the state *failed* or *available*, a constant number of customers of class 2 distribute over four variables and the server is either idle or serving a customer of class 1 or of class 2, which is all as expected. The list of warnings contains 4 warnings, one for each of the three variables and one for the action that are all not covered by any invariant, which guides us to the error in the model.

For further details and the full report for the server model, see the Traviando's example page [2].

5. Conclusion

We presented an approach to identify model characteristics from a simulation trace that contains information on states as a set of value settings for state variables and events with associated information on time stamps and actions. The approach is implemented as a command-line extension to Traviando and generates a detailed report in HTML format. Among other information, the report lists warnings for unusual behavior and those warnings come with links to webpages that contain further documentation on the rules that are associated with a warning. A webpage for a particular rule gives guidance on symptoms, possible causes and suggests solutions for known pitfalls in simulation modeling. Ongoing work is dedicated to extending the ruleset and increasing the size of the sample set. At this point, the existing software is available on request for research and teaching purposes. We plan to make a future version freely available together with a rule set of modeling rules.

Acknowledgements. We would like to thank the reviewers and Daniel Varro for their constructive comments and Ruth Lamprecht for her editorial assistance, which all helped to improve this paper.

References

[1] NS2: The network simulator. http://www.isi.edu/nsnam/ns/.

[2] Traviando: www.cs.wm.edu/∼kemper/traviando.html, www.cs.wm.edu/∼kemper/traviando/examples.html.

[3] O. Balci. Quality assessment, verification, and validation of modeling and simulation applications. In *Proc. of the 2004 Winter Simulation Conference*, pages 122–129. IEEE, 2004.

[4] D. D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. G. Webster. The Möbius framework and its implementation. *IEEE Trans. Software Eng.*, 28(10):956–969, 2002.

[5] F. Bause et al. A toolbox for functional and quantitative analysis of DEDS. In *Computer Performance Evaluation / TOOLS*, Springer LNCS 1469, pages 356–359, 1998.

[6] F. Bause et al. The ProC/B toolset for the modelling and analysis of process chains. In T. Field et al, editor, *Computer Performance Evaluation / TOOLS*, Springer LNCS 2324, pages 51–70, 2002.

[7] D. Hovemeyer and W. Pugh. Finding more null pointer bugs, but not too many. In *Proc. 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2007.

[8] P. Kemper and C. Tepper. Automated trace analysis of discrete event system models. *IEEE Transactions on Software Engineering*, in print, 2009.

[9] R. G. Sargent. Verification and validation of simulation models. In *Winter Simulation Conference*, pages 157–169. ACM, 2008.

[10] A. J. M. M. Weijters and L. Maruster. Workflow mining: Discovering process models from event logs. *IEEE Transactions on Knowledge and Data Engineering*, 16, 2004.