Proceedings of the 2007 Winter Simulation Conference
S. G. Henderson, B. Biller, M.-H. Hsieh, J. Shortle, J. D. Tew, and R. R. Barton, eds.

A TRACE-BASED VISUAL INSPECTION TECHNIQUE TO DETECT ERRORS IN SIMULATION MODELS

Peter Kemper

Department of Computer Science
College of William and Mary
Williamsburg, VA 23187, U.S.A.

ABSTRACT

Generation of traces from a simulation model and their analysis is a powerful and common mean to debug simulation models. In this paper, we define a measure of progress for simulation traces and describe how it can be used to detect certain errors. We devise a visual inspection technique based on that measure and discuss several examples to illustrate how one can distinguish normal behavior from irregular, potentially erroneous behavior documented in a trace of a simulation run. The overall approach is implemented and integrated in Traviando, a trace analyzer for debugging stochastic simulation models.

1 INTRODUCTION

Discrete event simulation of stochastic models is very useful in the evaluation of real-world systems, be it to address performance, dependability, performability, survivability or related issues. Encoding an abstract, conceptual model as input for some simulation engine is a task that may introduce errors of many kind, so debugging simulation models becomes an issue very much the same way as in software development in general. In the modeling and simulation area, this is part of the verification and validation steps performed in a simulation study. Errors may be rooted in a misconception of the original system, a misrepresentation of a system in a conceptual model, a faulty encoding of the conceptual model into an executable simulation model, a faulty implementation of the employed simulation engine, to name a few. In this paper, we focus on those errors only that document themselves as irregular, irreversible changes to the state of a discrete event simulation model, which are errors typically encountered in the development of an executable simulation model.

Sadowski (2005) discusses a number of pitfalls in simulation modeling and gives advice how to avoid them. In particular, she recommends to review a model and other deliverables early and often and recommends a structured walk-through with colleagues and clients as ideal to discover errors in models. Krahl's tutorial (Krahl 2005) on debugging simulation models gives further hands-on advice from a practitioner's point of view. A common technique is to modify the model itself to reveal a particular behavior for testing purposes. This includes the reduction or partition of a model and to analyze those parts in isolation as well as the modification of rates, timings, and priorities to see a certain dynamics happen. In addition, a modeler often enhances the model by assertions added to the simulation code, that are checked at runtime and that do not contribute to the model itself but helps to recognize the presence of errors. Assertions are limited to safety properties that can be checked as a particular statement in code; they require a modeler to be aware of such properties and being able to express them in the input language of a simulator and to do so, usually in a manual way. Complementarily to enhancements of a simulation model, professional simulators like Arena (Kelton, Sadowski, and Sadowski 2002), Automod (Banks 2000) among others provide support for animation to check face validity of a model plus debugging functionality as known from software development in general, i.e., step by step computation, breakpoints, inspecting variables and data structures. This is all valuable and useful and in addition one can also document what happens in a simulation run by writing a trace. Analysis of simulation traces is usually described as a tedious step by step control of what a simulator does (Krahl 2005, Law and Kelton 2000) and of course one can provide support for an automated analysis of traces. Trace analysis is also an issue in other fields. For instance in runtime verification, monitoring software reads a trace to diagnose problems, and applies model checking or statistical hypothesis testing on the fly to identify erroneous behavior, as supported for instance by the MaC analyzer (Sammapun, Lee, and Sokolsky 2005).

However, in tracing a simulation model, a modeler may find himself in the situation that a simulation run gives somehow obscure results, but it is unclear for what properties to ask for to be checked by a model checker or what hypothesis to test for by some statistical analysis. For this situation, we propose a simple visual inspection technique

that gives an indication if a particular type of error is present in a trace. The underlying assumption is (based on an observation we made for many performance and dependability models) that a model shows a cyclic behavior, i.e., that states in a simulation trace are reached repeatedly, and that certain coding errors disturb this regularity. Surprisingly, we observed this for models with very large state spaces or even infinite state spaces (of which a simulator would explore only a finite subset in finite time). We believe that the cyclic behavior results from common ways of describing a simulation model, e.g., a work load generator typically loops between a load generation phase and an idle phase, a server loops between different stages of service and an idle stage, a dependable subsystem may cycle between different levels of operation, failure and repair stages. Given that simulation is used to feed a statistical analysis with a set of samples, and if more than one sample is generated from single run, it is likely that the model loops through a potentially large set of states but may occasionally visit certain states again. So a cyclic behavior is an expected, "good" behavior, while certain errors may disturb that. For example, an event that makes a faulty increment operation to a state variable as result of a typing error would disturb that property. A second example is a model of an open system where customers arrive, get served and depart, and that has a partial deadlock. This model will show some dynamics in a simulation run even after the deadlock has been reached due to newly arriving entities, but the accumulating effect of blocked entities will prevent the model to return to previous states of less customers in the system. So, the non-returning, progressing part of a trace may deserve particular attention.

In this paper, we describe how to measure that progressing part and by visualizing that measure over the length of a simulation run, we propose to detect the presence of events that disturb the cyclic behavior and to give guidance to a modeler at which parts of a lengthy trace to look for causes of that effect. The rest of the paper is structured as follows. Section 2 gives some basic definitions and in particular defines the measure of progress that we use to devise the visual inspection technique we discuss with the help of examples in Section 3. We describe corresponding tool support in Section 4 and conclude in Section 5.

2 PROGRESSIVE BEHAVIOR IN TRACES

A trace is a sequence $\sigma = s_0 e_1 s_1 \ldots e_n s_n$ of states $s_0, \ldots, s_n \in S$ and events $e_1, \ldots, e_n \in E$ over some (finite or infinite) sets S, E for an arbitrary but fixed $n \in \mathbf{N}$. For elements of S, we assume an equivalence relation denoted by "=". For example, if $s \in S \subseteq \mathbf{N}$ is the state of an automaton, $=$ may be the usual equality among integer values, if $s \in S$ is a marking of a Petri net, then $=$ is the equality of markings (integer vectors), if $s \in S$ is the description of a term of a process algebra, then $=$ may be defined as a weak or strong bisimulation, and similarly for other formalisms with some notion of bisimulation for states like the multi-paradigm models of Möbius (Deavours et al. 2002). In general, it is to the modeler to decide which fragment of the state of a simulator (current state and future event list) should be documented as a state s. This allows for abstractions with the benefit of reducing the considered amount of data and the risk of missing important details. Note that events are irrelevant in the following formal treatment, but events are important pieces of information to document what happens in a trace. Hence, we keep events within our considerations. We define some common operations for sequences. The length of $\sigma = s_0 e_1 s_1 \ldots s_n$ is defined as $|\sigma| = n = \#events$. The concatenation $\circ$ of two sequences $\sigma = s_0 e_1 s_1 \ldots s_n$ and $\sigma' = s'_0 e'_1 s'_1 \ldots s'_m$ where $s_n = s'_0$ is defined as $\sigma \circ \sigma' = s_0 e_1 s_1 \ldots s_n e'_1 s'_1 \ldots s'_m$. Obviously, if $\sigma'' = \sigma \circ \sigma'$ then $|\sigma''| = |\sigma| + |\sigma'|$. For $\sigma = s_0 e_1 s_1 \ldots s_n$ and $0 \le i < j \le n$, we define a projection or substring operation as $sub(\sigma, i, j) = s_i e_{i+1} \ldots s_j$. Let $\sigma_i = sub(\sigma, 0, i)$ denote the special case of a prefix of length i of σ. A cycle is a substring $sub(\sigma, i, j)$ with $i < j$ and $s_i = s_j$. We use the notation $[i, j]$ for a cycle in σ. A cycle $[i, j]$ is elementary if $s_i \neq s_k$ for $i < k < j$. Let $\mathscr{C}_{all} = \{[i, j] | 0 \le i < j \le n, s_i = s_j\}$ be the set of all cycles of σ, $\mathscr{C} = \{[i, j] | [i, j] \in \mathscr{C}_{all}, s_k \neq s_i, i < k < j\}$ be the set of all elementary cycles. Cycles allow us to define a reduction operation. For a $\sigma = s_0 e_1 s_1 \ldots e_n s_n$ with cycle $[i, j]$, we define a reduction operation $red(\sigma, i, j) = sub(\sigma, 0, i) \circ sub(\sigma, j, n)$. The reduction operation is consistent with the notion of length, $|red(\sigma, i, j)| = |\sigma| - |sub(\sigma, i, j)|$ (given that $[i, j]$ is cycle of σ).

Let the sequence reduction problem (SRP) denote the problem to reduce a given σ with the help of the reduction operation to the shortest possible sequence σ^* and let $\mathscr{C}^*$ be a set of cycles that yields that reduction. We define this formally as follows.

Definition 1 *For a trace σ, SRP is the problem to determine a $\mathscr{C}^* \subseteq \mathscr{C}$, such that $\sum_{[i_1, i_2] \in \mathscr{C}^*} (i_2 - i_1)$ is maximal and for any two elements $[i_1, i_2], [j_1, j_2] \in \mathscr{C}^*$ holds that $i_2 \le j_1$ or $j_2 \le i_1$.*

The former condition ensures that we obtain a maximal reduction of σ. The latter condition ensures that we can apply $red(red(\sigma), j_1, j_2), i_1, i_2)$ (or vice versa), i.e. the cycles are at most adjacent but do not overlap. Note that the condition also excludes intervals $[i_1, i_2], [j_1, j_2]$ with $i_1 < j_1 < j_2 < i_2$, however in that case $red(\sigma, i_1, i_2) = red(red(\sigma, j_1, j_2), i_1, i_2)$, so $[j_1, j_2]$ is irrelevant for a maximal reduction and can safely be excluded. Since σ_i is a trace as well, let $\sigma_i^* = (sub(\sigma, 0, i))^*$ denote the shortest possible sequence for a prefix of length i of σ that is obtained as a solution of SRP for σ_i. We use a solution of SRP to define a measure of progress for a trace.

Table 1: A trace σ with σ_i^*, and $p_\sigma(i), i \in \{0,1,\ldots,12\}$.

σ	$= A_0eB_1eC_2eD_3eC_4eE_5eF_6eE_7eG_8eD_9eH_{10}eG_{11}eI_{12}$	
σ_0^*	$= A_0$	$p_\sigma(0) = 0$
σ_1^*	$= A_0eB_1$	$p_\sigma(1) = 1$
σ_2^*	$= A_0eB_1eC_2$	$p_\sigma(2) = 2$
σ_3^*	$= A_0eB_1eC_2eD_3$	$p_\sigma(3) = 3$
σ_4^*	$= A_0eB_1eC_2$	$p_\sigma(4) = 2$
σ_5^*	$= A_0eB_1eC_2eE_5$	$p_\sigma(5) = 3$
σ_6^*	$= A_0eB_1eC_2eE_5eF_6$	$p_\sigma(6) = 4$
σ_7^*	$= A_0eB_1eC_2eE_5$	$p_\sigma(7) = 3$
σ_8^*	$= A_0eB_1eC_2eE_5eG_8$	$p_\sigma(8) = 4$
σ_9^*	$= A_0eB_1eC_2eD_3$	$p_\sigma(9) = 3$
σ_{10}^*	$= A_0eB_1eC_2eD_3eH_{10}$	$p_\sigma(10) = 4$
σ_{11}^*	$= A_0eB_1eC_2eE_5eG_8$	$p_\sigma(11) = 3$
σ_{12}^*	$= A_0eB_1eC_2eE_5eG_8eI_{12}$	$p_\sigma(12) = 4$

Definition 2 *For a trace σ, we define $p_\sigma : \{0,1,\ldots,|\sigma|\} \to \{0,1,\ldots,|\sigma|\}$ with $p_\sigma(i) = |\sigma_i^*|$ as the progress of σ.*

p_σ is well-defined since the length $|\sigma_i^*|$ that is achieved by a maximal reduction is unique even if the SRP for σ_i has several solutions. Table 1 illustrates the concept for a trace σ with S being capital letters, $E = \{e\}$ and $\mathscr{C} = \{[2,4],[3,9],[5,7],[8,11]\}$. The progress of σ has a number of obvious properties. The following equations are mainly based on the observation that SRP of σ_{i+1} can use a solution of SRP of σ_i, so any reduction of σ_{i+1} removes at least as many states the reduction of σ_i.

For any trace σ and $0 \le i \le n$,

$$0 \le p_\sigma(i) \le i, \tag{1}$$

$$0 \le p_\sigma(i+1) \le p_\sigma(i) + 1 \le i+1, \tag{2}$$

$$p_\sigma(i) + c = i \Rightarrow p_\sigma(j) + c \le j \qquad \forall i \le j \le n. \tag{3}$$

An algorithm to solve SRP is described in Kemper and Tepper (2007), namely algorithm AOPT, which has linear time and space complexity and is implemented within Traviando (Kemper and Tepper 2005a, Kemper and Tepper 2006). It is immediate to use algorithm AOPT to compute p_σ. Since the algorithm is based on dynamic programming and makes use of $\sigma_0^*, \sigma_1^*, \ldots, \sigma_{i-1}^*$ for the computation of σ_i^*, we can compute $p_\sigma(i)$, $0 \le i \le n$ with a single application of AOPT. So for any σ, we can compute all values of p_σ in $O(n)$. In the following, we discuss what p_σ is able to reveal for a simulation trace.

3 VISUAL INSPECTION

In this section we discuss typical patterns we observed for p_σ by evaluating a number of example traces generated from a variety of stochastic models for performance and dependability assessments. p_σ turns out to be useful to recognize a failure of a safety property, that informally denotes that "once something bad has happened, the system/model is unable to recover from that event."

Table 2: Set of example models.

Model	Tool	$\lvert SV\rvert$	$\lvert ET\rvert$
Models without errors			
Courier	APNN	45	34
ProdCell	APNN	231	202
MM1,$\lambda < \mu$	APNN	1	2
DinPhil	ProC/B	56	95
MultiProc	Möbius	11	16
Conveyor	Möbius	6	8
Database	Möbius	19	18
Models with errors			
Courier, Bug I-III	APNN	45	34
MM1,$\lambda \ge \mu$	APNN	1	2
Store	ProC/B	20	36
Server	Möbius	11	12

3.1 Set of Example Models

We analyze traces of several example models. Table 2 gives a brief summary of their characteristics; it gives the name of the model in the first column, the modeling and simulation framework used to generate traces in column *Tool*, the number of state variables $|SV|$ and the number of different types of events $|ET|$ in columns three and four. Note that events E that are present in a trace σ are typically generated from a much smaller set of "types of events" described by a modeler in the model itself. We consider traces of different lengths for those models.

Courier is a stochastic Petri net model of a Courier protocol taken from Woodside and Li (1991). The APNN toolbox (Bause, Buchholz, and Kemper 1998) is used to model and simulate it to obtain σ. The model has a non-trivial amount of state variables, i.e. a state s in the trace is a vector of length 45, and contains a non-trivial amount of different types of events (i.e., transitions in Petri net terminology) that transform states into successor states. We analyzed the effect of three different errors, that we injected by manipulating incidence functions of certain transitions (Bugs I-III). For Bug I, we release a previously allocated shared resource more than once. For Bug II, we increase the number of packets that are send at a lower level of the protocol. For Bug III, we repeatedly increase the window size. Note that *Courier* is an example of a closed model where the correct version has a finite state space.

ProdCell is a model of a production cell developed by Heiner and Deussen (1996). The original model has been imported into the APNN toolbox. The trace has been generated from a discrete event simulation. This model is the largest we consider with respect to the number of state variables and types of events. *MM1* is a stochastic Petri net model of the classical queueing model (arrival rate λ,

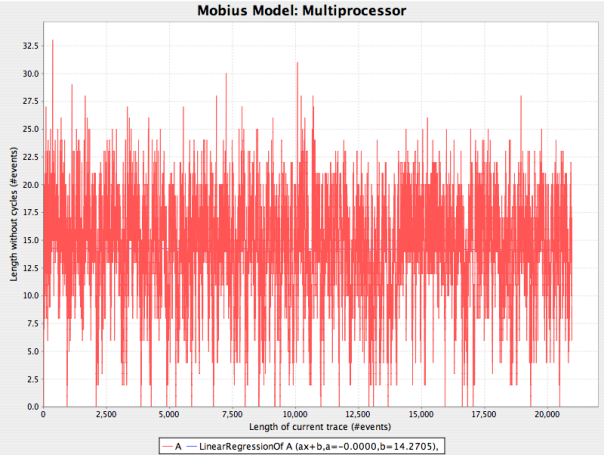

Figure 1: Progress p_σ for a σ of the *MultiProc* model.

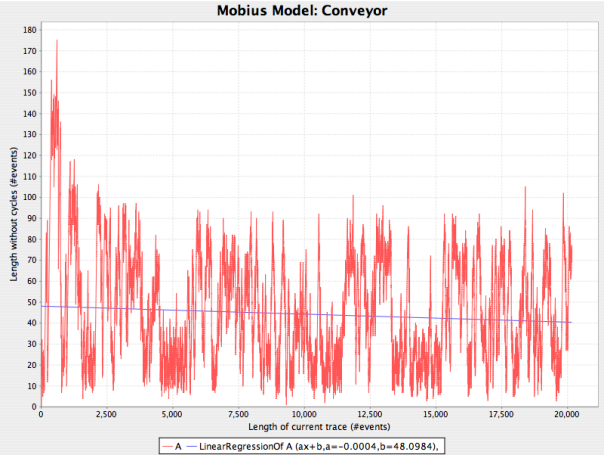

Figure 2: Progress p_σ for a σ of the *Conveyor* model.

service rate μ). We consider this model to illustrate the effect of overload (cases $\lambda = \mu$ and $\lambda > \mu$) in a model of an open system versus a stable configuration ($\lambda < \mu$).

We also consider example models from the Möbius framework (Deavours et al. 2002), a multi-formalism multi-solution framework for stochastic modeling of systems for performance and dependability studies. The simulation engine of Möbius can be configured to generate traces. *MultiProc* is a stochastic model of a fault-tolerant multi-processor system, *Conveyor* models conveyors and *Database* models dependability of a data base system. Those models are available with the Möbius distribution, they are correct models which are modest in the number of state variables and types of events. *Server* is a Möbius model discussed in Kemper and Tepper (2007). It contains an error that rarely takes place and that increases the number of customers in a system that is supposed to be closed and to have a fixed population of customers.

DinPhil is a non-traditional model of the classical dining philosophers developed with the ProC/B toolset (Bause et al. 2002), which is intended to model open systems with a process interaction approach. This model is merely a demonstration model how to model closed models instead of open models within ProC/B. A more typical model for this framework is the *Store* model (Kemper and Tepper 2005a), which describes how goods are delivered and taken away from a storage area. The model follows a process interaction approach to simulation modeling, where entities are trucks that transfer goods from and to a storage area where loading and unloading operations are performed by manned forklifts. *Store* is an example of a model that contains a partial deadlock; the simulation reaches a situation where no trucks can be unloaded or loaded for either lack of storage space or lack of goods. The deadlock is only partial since new entities (trucks) can always enter the system.

3.2 Behavior of Correct Models

We begin with a visual inspection of plots for p_σ generated from traces of correct models to obtain a baseline for our observations.

Figure 1 shows p_σ for a trace of length $n = 20,000$ that has been generated from the *MultiProc* model. (The pdf file of this paper contains color graphics, and zooming in helps to make fonts more readable.) Note that p_σ is a discrete function but the plot in the figure connects points $(i, p_\sigma(i))$ to stress how the function evolves. We can interpret this function as if it oscillates in a regular manner with some random noise added to it. The function occasionally but repeatedly reaches $p_\sigma(i) = 0$ for some $0 \leq i \leq n$ which means that the model returns to the initial state. Its maximum value is in the lower 30s, which indicates that the simulation explores a part of the state space of the model that is rather shallow than deep.

A similar pattern is observed for a trace of $|\sigma| = 20,000$ for the *Conveyor* model. The model is described in a stochastic process algebra, so the modeling formalism is quite different to the SAN formalism used for the fault tolerant multiprocessor model. Figure 2 shows p_σ for this trace and the plot has similar characteristics as the previous one. Both show a function that oscillates in a rather limited range of values. The plot in Figure 2 initially reaches some maximal height that it does not reach afterwards anymore.

Figure 3 shows a plot from a trace we generated from a correct *Courier* model, which again yields a similar curve. If we fit a linear regression model $r(x) = a + b \cdot x$ to p_σ then the slope is close to zero for the curves of all three models. Table 3 gives values for interception a and slope b for an ordinary least square fitting for traces of different length n from the variety of models we consider. The first column gives the name of the model, the second the length of the trace, column three and four give the interception and slope of the fitted linear regression model. If we consider the

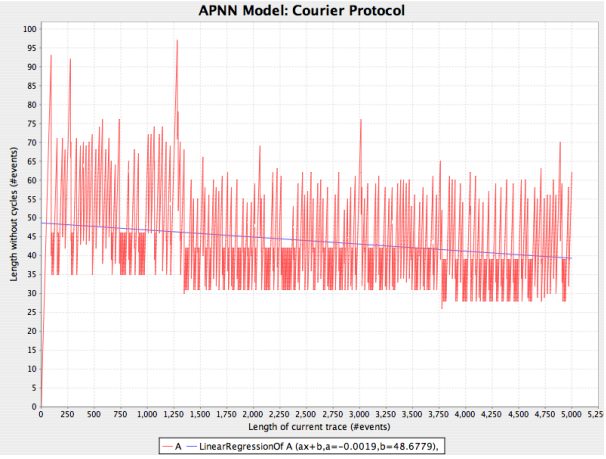

Figure 3: Progress p_σ for a σ of the correct *Courier* model.

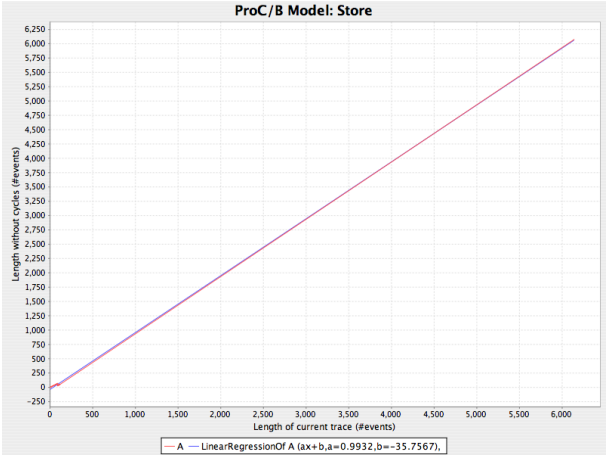

Figure 4: Progress p_σ for *Store* model (with error). Plots for *Courier* with Bug I, II, and III have same characteristics.

slope of correct models, we recognize that in most cases the slope is close to zero and for those cases where it is not (*Courier* with $n = 100$ and *ProdCell* with $n = 1000$), the slope decreases with an increasing length n. So a normal behavior over this range of correct models is that p_σ oscillates without any significant trend. The *MM1* models with no stable solutions due to overload are deliberately put into the category of models with errors. An overload scenario is expected to create a positive slope.

3.3 Effect of Frequent Errors

In this section, we discuss how to recognize if a model contains a safety property violation of the kind discussed above, namely that an error happens that yields an irreversible state change. More formally let $AP : \{s_0, \ldots, s_n\} \to \{tt, ff\}$ be a boolean function for an atomic proposition such that $\exists s_i \in \{s_0, \ldots, s_n\}$ with $AP(s_j) = b$ for $0 \leq j < i$, $b \in \{tt, ff\}$ and $AP(s_j) = \neg b$ for $i \leq j \leq n$ then s_i will be present in all σ^*_j for $i \leq j \leq n$ and $p_\sigma(i-1) < p_\sigma(j)$ for $i \leq j \leq n$. So, in this case $c = p_\sigma(i-1)$ becomes an additive constant to the normal range of values observed for p_σ when the erroneous event does not take place. For this type of error, we expect the long range visual perception of p_σ be that of a discrete monotonously non-decreasing function with some additive noise, i.e., $p_\sigma(i) + \varepsilon(i) = i$ with $0 \leq i \leq n$ and $0 \leq \varepsilon(i) << i$ being relatively small in value for large values of i. In particular, if the erroneous event occurs frequently, the slope of p_σ is expected to be significant and close to 1.

We begin with the *Courier* model as an example of a closed system and observe a significant difference between the progress for a trace of the correct model in Figure 3 and the progress seen for traces of variants of the *Courier* model where we injected Bug I, II, or III, which all three give plots of p_σ of same characteristics as the one in Figure 4. The three errors differ in their interpretation from a modeling point of view (as discussed in Section 3.1) but their common effect is to add additional tokens on certain places of the Petri net. We evaluated three variants to see if the impact on p_σ is consistent. Table 3 gives the corresponding values for the linear regression with a slope consistently close to 1 for faulty models and close to 0 for traces of the correct model if the length of the trace is significant ($n > 100$).

For models of open systems, where entities arrive and depart, we consider the *Store* model and an *MM1* queue. The *Store* model has a deadlock but retains some dynamics since there is an unlimited stream of newly arriving entities over time, that enter the model, traverse a service network and then get blocked at some resource inside. Figure 4 shows p_σ for a trace of the model where the deadlock is reached. Again, the plot appears as a rather straight line, the regression function has a slope close to 1.

Note that the relation between erroneous behavior and the characteristics of p_σ is not one to one. For instance, if a state variable is used to measure the dynamics, e.g., to count the number of failures, or the number of customers served on time, or the number of packets lost due to buffer overflows, then that variable – if taken into consideration – will disturb the observation of cycles in a trace and the resulting p_σ may be misleading. This is an example of a false-positive, a non-fault, for the visualization technique. For the case of such a counter variable, one needs to restrict the state variables that are either exported into the trace by the simulation engine or to refine the notion of equality $=$ used to compare states of σ accordingly. In addition, overload scenarios in models of open systems will give plots of p_σ that appear as a straight line. To illustrate the point, we exercise the classical MM1 queueing model with different degrees of load. Figure 5 shows the stable case, where arrival rate λ is less than the departure rate μ. Figure 6 shows the case $\lambda = \mu$, Figure 7 a case of $\lambda > \mu$ which are both not stable. An overload situation in an open system yields a temporarily delay of individual entities which we

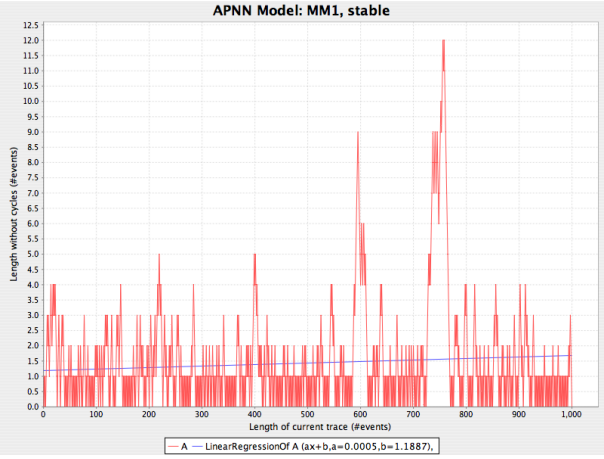

Figure 5: Progress p_σ for *MM1* for stable case $\lambda < \mu$, II.

cannot distinguish from a permanent blocking of entities in a deadlocking situation by considering a finite trace σ.

In summary, p_σ can give an indication of violations of the type of safety property discussed above for open and closed models and in addition to that also indicate extensive blocking (deadlocking or overload) situations for open models. The cases we considered so far are of the kind that erroneous events are present frequently in the trace. This is key to the observed effect of a steady increase in p_σ. In the next section, we investigate how to detect infrequent or rare erroneous events.

3.4 Effect of Rare Errors

The *Server* model is an example of a model, where an error takes place rarely. Figure 8 shows p_σ for a trace σ where the erroneous event takes place only once and around $i = 3500$. If we compare states at $i < 3400$ and $i > 3600$, we recognize that an event caused an increase of an otherwise invariant number of customers in a closed system. The large amount of events between the beginning and the occurrence of the error and the end of the trace allows us to easily detect phases of normal oscillating behavior and the sudden discrete increase to p_σ as abnormal behavior. This is different at a smaller scale, since a temporary increase of p_σ is also observed with a correct model. In particular in an initial phase, we frequently observe a large increase of p_σ. If we try to formalize what is particular about Figure 8, the linear regression model does not(!) give us a good indication. The increase of p_σ is there and imposes a positive slope, however the length of the trace drags the value down to a marginal amount that is in the same range as values observed for correct models in Table 3. Note that the significant piece of information is the minimum of p_σ that we observe for a certain range of values. Since $p_\sigma(0) = 0$, we define a measure for the observed minimum in a backward manner. Let $bmin : \{0,\ldots,n\} \to \{0,\ldots,n\}$ be defined as $bmin(i) = min\{p_\sigma(j) | i \leq j \leq n\}$. Figure 9 shows $bmin()$ computed for several traces of correct models and the *Server* model. Note that all curves have a sharp increase at the end, some also at the beginning, and *Server* as well as *Courier* have a step in an otherwise flat lengthy middle part. The step in *Server* guides us to the erroneous event, the step in *Courier* is a false-positive, that points us to a particular region in the trace of a correct model. Obviously, $bmin()$ is helpful, but not as clear as p_σ itself. Figure 3 show p_σ and the linear regression for that trace of *Courier*. By visual inspection, we do not think that p_σ is in fact increasing in the long run for *Courier*. So at this point we conclude that the visual inspection of p_σ helps us to identify rare errors in a sufficiently long trace, sufficiently long to contain the erroneous event and sufficiently long to avoid the distraction of an initial phase that often shows a stepwise increase of p_σ without giving the right impression on the "normal" oscillating behavior.

Table 3: Parameters of a fitted linear regression.

Model	n	slope	intercept
Models without errors			
Courier	100	0.3928	14.4650
Courier	1000	0.0022	36.3960
Courier	10000	-0.0002	38.5363
ProdCell	1000	0.0551	100.5734
ProdCell	10000	-0.0023	366.2974
MM1,$\lambda < \mu$	1000	0.0005	1.1887
DinPhil	67744	-0.0002	32.6560
MultiProc	4368	-0.0004	15.4978
MultiProc	20961	0.0000	14.2705
Conveyor	5391	-0.0088	76.1606
Conveyor	20160	-0.0004	48.0984
Database	4732	0.0000	0.5052
Database	19974	0.0000	0.5037
Models with errors			
Courier, Bug I	10000	0.9964	0.5863
Courier, Bug II	10000	1.000	0.0000
Courier, Bug III	10000	1.000	0.0000
Store	6140	0.9932	-35.7567
Server	5473	0.0018	0.4516
MM1,$\lambda = \mu$	1000	-0.0027	16.7332
MM1,$\lambda > \mu$	1000	0.3902	-3.5411

3.5 Decomposition

Large and complex simulation models are often composed of submodels. For example, the *ProdCell* model is composed of submodels for individual components of the modeled production cell, namely a press, a robot with two arms, 2 conveyor belts, a table and a crane. This is common for most models and modeling tools typically support this, for example, Möbius provides ways to compose submodels by

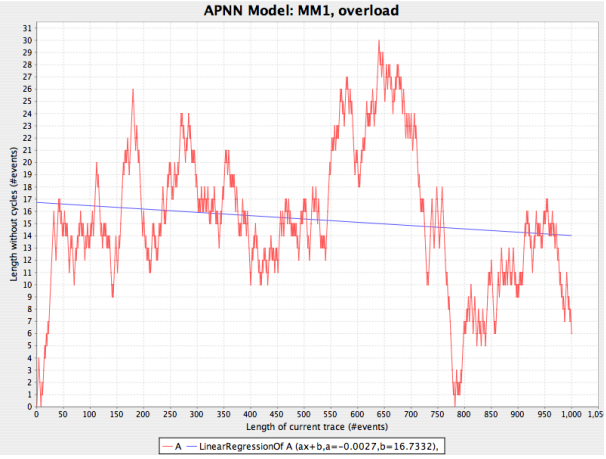

Figure 6: Progress p_σ for *MM1* for borderline case $\lambda = \mu$.

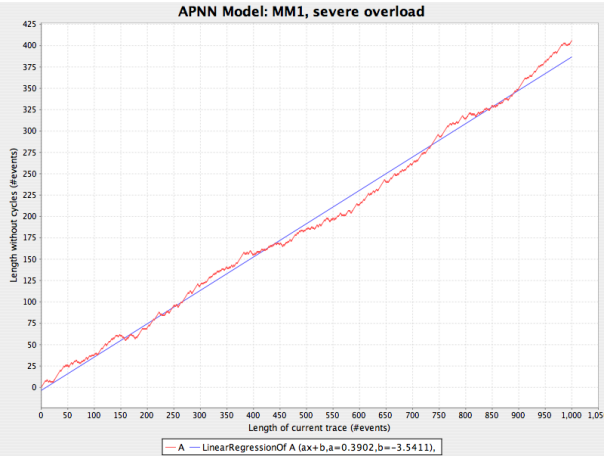

Figure 7: Progress p_σ for *MM1* for overload case $\lambda > \mu$.

action sharing and sharing of state variables. This composition usually implies a state s being a vector $s = (s^1, \ldots, ^m)$ of states of m submodels and allows us to project a state s to a substate s^i for some submodel i. In this case, we can modify our notion of equality "=" among states to focus on certain submodel states or certain state variables only. Consequently, we can obtain different functions $p^i_\sigma()$ for submodels $i = 1, \ldots, m$ and use that decomposition to investigate in the cyclic behavior of individual submodels.

For example, the Möbius model of *Server* consists of submodels *FailureAndRepair*, *Service*, *CompleteServer*, *UserA*, *UserB* and *FullModel* based on a rep/join state variable sharing composition, for details on composition of models see Deavours et al. (2002). *FailureAndRepair* describes details of when the server fails and how long it takes to become operational again. The projected $p^i_\sigma()$ for this submodel shows a cyclic behavior of a correct model with no indication of an error. The projected $p^i_\sigma()$ for the *FullModel* submodel shows a similar pattern as in Figure 8 and guides us to investigate in that part of the model. This

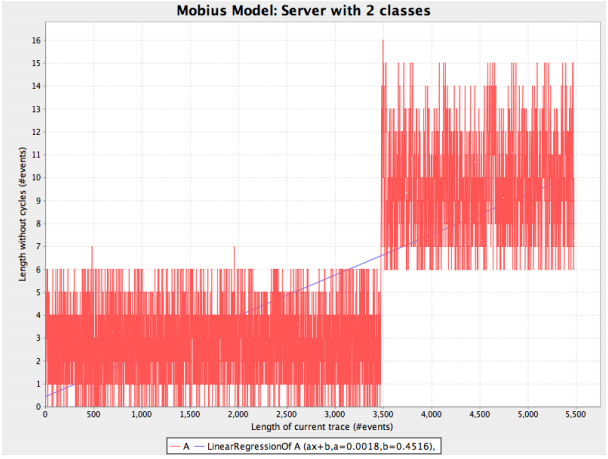

Figure 8: Progress p_σ for *Server* model (with error).

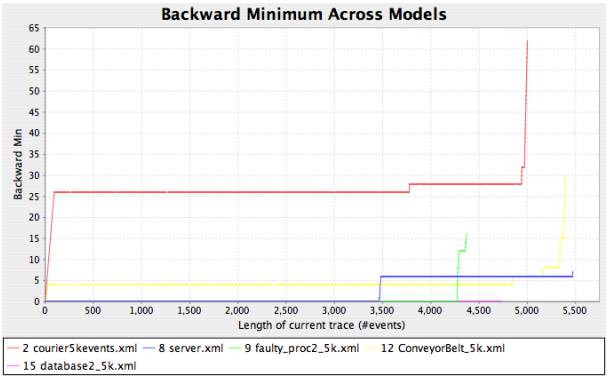

Figure 9: Function *bmin*() for several models.

decomposition can help a modeler to identify submodels that deserve more attention than others. However, we should also note that the error in fact is in submodel *Service* where in case of a failure a customer returns to the waiting queue. The error is that an additional, second customer of the other customer class is created that also returns to the queue. So the effect of the error can be observed in *FullModel* and *UserA* but the cause of that effect is located in *Service* which interacts with the other submodels. An alternative to a top-down approach is to use the evaluation of submodels with respect to $p^i_\sigma()$ in a bottom-up approach to recognize irregular behavior for submodels and how this impacts the overall behavior.

4 TOOL SUPPORT

Traviando is a software tool that analyzes and visualizes traces. It provides a variety of analysis techniques including statistical analysis, model checking, bottleneck and deadlock detection and the visual inspection technique presented in this paper. Its particular purpose is to make information accessible to a human being, that is otherwise hidden in a large trace file. Traviando visualizes a trace as a variant

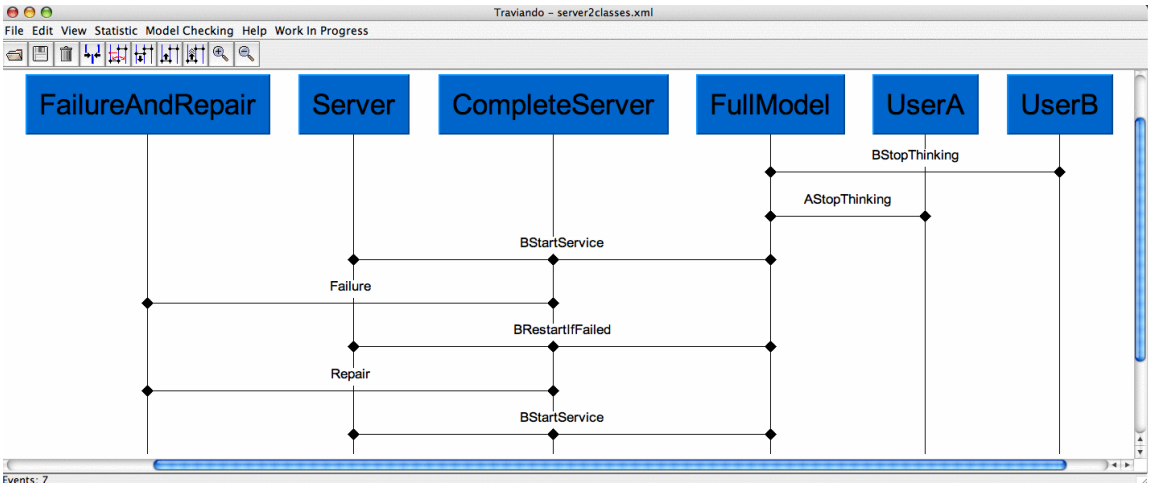

Figure 10: MSC Representation of reduced trace σ^* for the server model with 2 customer classes.

of message sequence charts (MSCs) much similar to UML sequence diagrams. Figure 10 shows the reduced trace σ^* for model *Server* as an MSC where each submodel is an MSC process and where time proceeds from top to bottom. For this example, the interaction is undirected, for others Traviando supports directed arcs with a distinguished sender and receiver as well. MSCs are promising since they focus on the interaction of processes and we believe that this is crucial: in modeling with interacting submodels, the behavior of an individual submodel is rather simple to analyze and debug but the overall effect of interactions contributes much to the complexity of such models. For further information on trace visualization features, we refer to Kemper and Tepper (2005b). Traviando supports a number of operations that are helpful for trace analysis.

Cycle visualization and reduction. In addition to the visual inspection technique discussed in this paper with screen shots of Traviando's presentation of p_σ, the tool is able to remove cycles from a trace to obtain the reduced trace σ^*. For example, Figure 10 shows the resulting σ^* of a cycle reduction of a trace σ of the *Server* model with $|\sigma| = 5473$ events. With few states in σ^*, it is much simpler to track state information (which is displayed in a separate window) and to recognize if there is an error and which event introduced it.

Model checking. Traviando provides LTL model checking on traces to identify states and trace fragments that fulfill or not fulfill certain properties and thus deserve the modeler's attention. The model checker is enhanced with a user interface that provides commonly applied patterns for specifying real world properties in a modal logic like LTL.

Statistical evaluation. Traviando implements a number of statistical measures that can be evaluated and graphically visualized, e.g. the number of occurrences for actions, the empirical distributions for time distances between occurrences and delays, to name a few.

Bottleneck analysis, deadlock detection. Certain formalisms, like the ProC/B modeling notation for process interaction models, allow us to analyze an empirical distribution of life spans of entities as well as population and utilization of resources and response times. We make use of those distributions to identify regions of heavy utilization, bottlenecks and deadlocks. For further details, we refer to Kemper and Tepper (2005a).

Trace format and supporting simulators. Traviando imports sequences in an open XML format that consists of two parts, a prefix and the sequence of events that constitutes the trace. The prefix contains definitions for processes, events, their type and association with processes, state variables and more. The prefix helps to keep the sequence of events concise in its description and also allows for some preprocessing and consistency checks for the trace based on the given structural information. The sequence of events in the second part of a trace can be enhanced in many ways by additional information, for instance by time stamps, information on entities and changes to state variables. Traviando's trace interface is currently supported by the APNN toolbox (Bause, Buchholz, and Kemper 1998), the ProC/B toolset (Bause et al. 2002) and Möbius (Deavours et al. 2002).

5 CONCLUSION

We presented a visual inspection technique to identify errors in stochastic discrete event simulation models with the help of trace analysis. The type of error recognized documents itself as changes to the state of a model that are not taken back afterwards. These are safety properties of the kind that "once a bad thing happened, the system is unable to recover from that." The type of models that benefit from our approach are models of open or closed systems with an inherently cyclic behavior as it is often the case

in dependability and performance modeling. We define a measure of progress p_σ that applies to a trace σ and a plot of p_σ shows particular distinct characteristic that help us to identify traces with errors. We discuss characteristics of p_σ with the help of a rich set of examples obtained from different modeling formalisms and tools, of different topic areas and different levels of complexity. We consider the proposed visual inspection technique as a useful complement to the existing set of debugging techniques for simulation models. Information on corresponding tool support can be found at <www.cs.wm.edu/~kemper/Traviando.html>.

ACKNOWLEDGMENTS

We would like to acknowledge the contributions of Carsten Tepper and several students to the development of Traviando. We thank William H. Sanders and Mike Mc Quinn for providing support for trace generation functionality in Möbius, Adelinde Uhrmacher for her feedback on this paper. Traviando plots of p_σ are produced with the help of the jfree graphics package.

REFERENCES

Banks, J. 2000. *Getting started with AutoMod*. 655 Medical Drive, Bountiful, Utah 84010: AutoSimulations, Inc.

Bause, F., H. Beilner, M. Fischer, P. Kemper, and M. Völker. 2002. The ProC/B toolset for the modelling and analysis of process chains. In *Computer Performance Evaluation / TOOLS*, Springer LNCS 2324, 51–70.

Bause, F., P. Buchholz, and P. Kemper. 1998. A toolbox for functional and quantitative analysis of DEDS. In *Computer Performance Evaluation / TOOLS*, Springer LNCS 1469, 356–359.

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

Heiner, M., and P. Deussen. 1996. Petri net based design and analysis of reactive systems. In *Proc. 3rd Workshop on Discrete Event Systems (WoDES96)*, 308–313.

Kelton, W., R. P. Sadowski, and D. A. Sadowski. 2002. *Simulation with Arena*. 2nd ed. Mc Graw Hill.

Kemper, P., and C. Tepper. 2007. Automated analysis of simulation traces - separating progress from repetitive behavior. In *Proc. QEST*, IEEE.

Kemper, P., and C. Tepper. 2005a. Trace based analysis of process interaction models. In *Proceedings of the 2005 Winter Simulation Conference*, ed. M. E. Kuhl, N. M. Steiger, F. B. Armstrong, J. A. Joines, 427–436. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Kemper, P., and C. Tepper. 2005b. Visualizing the dynamic behavior of ProC/B models. In *Proc. Simulation and Visualization*, ed. T. Schulze, G. Horton, B. Preim, and S. Schlechtweg, 63–74: SCS Publishing House e.V.

Kemper, P., and C. Tepper. 2006. Traviando - debugging simulation traces with message sequence charts. In *Proc. QEST*, 135–136: IEEE.

Krahl, D. 2005. Debugging simulation models. In *Proc. Winter Simulation Conference*, 62–68.

Law, A., and W. Kelton. 2000. *Simulation modeling and analysis*. 3rd ed. McGraw-Hill.

Sadowski, D. A. 2005. Tips for successful practice of simulation. In *Proceedings of the 2005 Winter Simulation Conference*, ed. M. E. Kuhl, N. M. Steiger, F. B. Armstrong, J. A. Joines, 56–61. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Sammapun, U., I. Lee, and O. Sokolsky. 2005. Rt-mac: Runtime monitoring and checking of quantitative and probabilistic properties. In *Proc. RTCSA*, 147–153: IEEE.

Woodside, C. M., and Y. Li. 1991. Performance Petri net analysis of communications protocol software by delay-equivalent aggregation. In *Proc. PNPM*, 64–73: IEEE.

AUTHOR BIOGRAPHY

PETER KEMPER is an associate professor in the Department of Computer Science at the College of William and Mary (previously Universität Dortmund and TU Dresden, Germany). His research interests include modeling techniques and tools for performance, performability and dependability analysis of systems. He contributed to analysis techniques for the numerical analysis of Markov chains, model checking stochastic models, techniques for simulation optimization. His web page can be found via <www.cs.wm.edu/~kemper>.