Automated Trace Analysis of Discrete Event System Models

Peter Kemper, *Member, IEEE,* and Carsten Tepper

Abstract—In this paper, we describe a novel technique that helps a modeler gain insight into the dynamic behavior of a complex stochastic discrete event simulation model based on trace analysis. We propose algorithms to distinguish progressive from repetitive behavior in a trace and to extract a minimal progressive fragment of a trace. The implied combinatorial optimization problem for trace reduction is solved in linear time with dynamic programming. We present and compare several approximate and one exact solution method. Information on the reduction operation as well as the reduced trace itself helps a modeler to recognize the presence of certain errors and to identify their cause. We track down a subtle modeling error in a dependability model of a multi-class server system to illustrate the effectiveness of our approach in revealing the cause of an observed effect. The proposed technique has been implemented and integrated in Traviando, a trace analyzer to debug stochastic simulation models.

Index Terms—cycle reduction, debugging techniques, trace analysis, discrete event simulation

1 Introduction

Debugging software is often the tedious task of executing a program and following a sequence of program statements or events that change the state of a program to see if anything happens other than what is expected. If one logs such an execution as a sequence of states and state changes caused by events, then this is the notion of trace we consider in this paper. While most algorithms perform a sequence of operations for a given input to come up with some calculated output, certain kinds of software iterate in a cyclic manner; examples include reactive systems that keep running and wait for a stimulus to start some calculations, like server software, communication protocols, control software, be it running with a simulated or a real environment to interact with. In this paper, we consider traces generated from running a simulation in a simulation framework for a given stochastic discrete event system model, for instance for a dependability or performance evaluation of a system. However, we believe that the characteristics seen in those traces, namely that certain states are repeatedly reached and a somewhat cyclic behavior, are expected to appear for other types of software as well. In particular, we expect this for control software that is evaluated as software-in-the-loop, since a controller can be seen as a finite automaton that implements a particular rule set to control some system.

A simulation program can be developed with any common programming language from scratch, however, presently it is usually derived with the help of some modeling framework that provides a simulation library and a modeler describes a simulation model with information on what constitutes the state of a model and a set of events that may change that state. Simulation comes with few restrictions so the real crux in simulation modeling is not to obtain numbers as results but to achieve valid results. The task of verification and validation of a simulation model requires a modeler to check that the implementation matches its specification as given by the conceptual model (usually termed verification in this context) and that the model behavior matches the behavior of the system under study (usually termed validation in that context) [19]. Sadowski [17] discusses a number of pitfalls in simulation modeling and gives advice on 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 on debugging simulation models gives further hands-on advice from a practitioner's point of view in [13]. A common technique is to modify the model itself to reveal a particular behavior, this includes reduction or partition of the model and to analyze parts as well as modification of rates, timings, and priorities to see certain dynamics happen. In addition, a modeler often enhances the model by assertions added to the simulation code, that are checked at runtime and do not contribute to the model itself but help 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, to be able to express them in the input language of a simulator and to do this in a manual way. Complementarily to enhancements of a simulation model, simulators like Arena [8] and Automod [1], 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.

- *P. Kemper is with the Department of Computer Science, College of William and Mary, Williamsburg, VA, 23187, USA.E-mail: kemper@cs.wm.edu*
- *C. Tepper is with ITGAIN Consulting, Hanover, Germany.*

Manuscript received....

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 [13, 14]. However, an automated trace analysis takes place in many other fields. For instance in runtime verification, monitoring software reads a trace to diagnose problems, apply model checking, or statistical hypothesis testing on the fly, as supported for instance by the MaC analyzer [18]. However, in tracing a simulation model, a modeler finds himself in the situation where it is unclear what properties to ask to be checked by a model checker or what hypothesis to test. This is the situation for which we want to provide support. In modeling for performance and dependability studies, we observed that models are often built from components that return to states repeatedly, 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 statistic analysis with a set of samples, and if more than one sample is generated from a single run, it is likely that the model loops through a potentially large set of states but may occasionally visit certain states again. Thus, cyclic behavior is an expected, "good" behavior, while certain errors may disturb that. For example, events that make irregular changes to state variables such that there is no inverse/reverse operation in the model, or a partial deadlock in an open process interaction model, where newly arriving entities create events but certain entities never depart. So, the non-returning, progressing part of a trace may deserve particular attention.

In this paper, we discuss how to automatically identify and remove repetition from a simulation trace. The resulting fragment sheds light on how a simulation progresses. The technical contribution of this paper is in the description and evaluation of heuristic and exact methods to extract and remove repetitive fragments from a trace. In particular we derive a linear time algorithm that gives a maximal reduction and that is novel to the best of our knowledge.We also describe two simple approximate algorithms for trace reduction. We propose to make use of cycle detection and reduction for the following purposes:

1. to obtain a graph that shows how the length of the minimal progressive fragment evolves with the length of the trace (the prefix considered for reduction). A visual inspection of that function often helps to recognize irregularities and pinpoint parts of a trace that deserve a closer look.
2. if a particular state of interest is found, a trace reduction helps to extract those events that are necessary to reach that state. This information reduction can be massive and simplifies tracking the cause of the effect that is observed at that state of interest.
3. to detect a set of cycles and to analyze their properties (which is not in the focus of this paper).

Obviously, a trace reduction by removing cycles preserves only those erroneous events that have no inverse counterpart. Formally, these can be seen as safety properties that once "something bad has happened" the system (the model) cannot overcome that bad situation in its future behavior. For this type of error, we consider our trace reduction approach a useful addition to the existing set of debugging techniques. The approach (cycle reduction as well as cycle visualization) is implemented in Traviando [11], a trace analyzer that tracks performance figures, provides statistical evaluation of timed and untimed traces as well as model checking functionality.

The rest of the paper is structured as follows. Section 2 gives basic definitions and states the sequence reduction problem. Section 3 describes how to debug a dependability model of a server with two classes and failure and repair to motivate the subsequent effort for reduction algorithms. Section 4 describes an exact algorithm that yields an optimal reduction. Section 5 describes approximate algorithms. Section 6 evaluates those algorithms with the help of several examples. Section 7 details ways to use the results of the exact algorithm - the length of the minimal progressive fragment - to recognize and detect errors in models. We conclude in Section 8.

2 DEFINITIONS

A trace is a sequence $\sigma = s_0 e_1 s_1 \dots e_n s_n$ of states $s_0, \dots, s_n \in S$ and events $e_1, \dots, e_n \in E$ over some (finite or infinite) sets S, E for an arbitrary but fixed $n \in \mathbb{N}$. For elements of S, we assume an equivalence relation denoted by "$=$". The length of $\sigma = s_0 e_1 s_1 \dots s_n$ is defined as $|\sigma| = n = \#events$. Let $\sigma_i = s_0 e_1 s_1 \dots s_i$ denote the prefix of length i of a trace σ, $0 \leq i \leq n$. The concatenation $\circ$ of two sequences $\sigma = s_0 e_1 s_1 \dots s_n$ and $\sigma' = s'_0 e'_1 s'_1 \dots s'_m$ where $s_n = s'_0$ is defined as $\sigma \circ \sigma' = s_0 e_1 s_1 \dots s_n e'_1 s'_1 \dots s'_m$. For $\sigma = s_0 e_1 s_1 \dots s_n$ we define a projection $pro(\sigma, \{i_0, \dots, i_m\}) = s_{i_0} e_{i_1} s_{i_1} \dots s_{i_m}$ that selects a subset of states and events from σ and retains elements in the same order as in σ; it selects states with index set $\{i_0, \dots, i_m\} \subseteq \{0, \dots, n\}$, events with index set $\{i_1, \dots, i_m\} \subseteq \{0, \dots, n\}$ and yields a sequence. We denote the special case of a substring by $sub(\sigma, i, j) = pro(\sigma, \{i, i+1, \dots, j\}) = s_i e_{i+1} \dots s_j$. A cycle is a substring $sub(\sigma, l, u)$ with $l < u$ and $s_l = s_u$. We use the notation $[l, u]$ for a cycle in σ. A cycle $[l, u]$ is elementary if $s_l \neq s_k$ for $l < k < u$. Let $\mathcal{C}_{all} = \{[l, u] | 0 \leq l < u \leq n, s_l = s_u\}$ be the set of all cycles of σ, $\mathcal{C} = \{[l, u] | [l, u] \in \mathcal{C}_{all}, s_k \neq s_l, l < k < u\}$ be the set of all elementary cycles. Two cycles $[l_1, u_1]$ and $[l_2, u_2]$ are non-overlapping if $(u_1 \leq l_2) \vee (u_2 \leq l_1)$, i.e., they are at most adjacent; a set of cycles is non-overlapping if any pair of its elements is non-overlapping.

We use the notion of cycles to define a reduction operation. Let $C' = \{[l_i, u_i] | 1 \leq i \leq k\}$ be a set of cycles of σ, we define a reduction $red(\sigma, C') = pro(\sigma, \{0, \dots, n\} \setminus \cup_{i=1}^{k} \{l_i + 1, l_i + 2, \dots, u_i\})$. If C' is a set of non-overlapping cycles, the length of a reduced trace is $|red(\sigma, C')| = |\sigma| - \sum_{i=1}^{|C'|} |sub(\sigma, l_i, u_i)| = |\sigma| - \sum_{i=1}^{|C'|}(u_i - l_i)$. For a single cycle $[l, u]$, $red(\sigma, \{[l, u]\}) = sub(\sigma, 0, l) \circ sub(\sigma, u, n)$.

Problem Statement. We are interested in reducing a given σ with the help of the reduction operation to the shortest possible sequence σ^*, yet σ^* should represent a dynamic behavior that is possible. For the latter, we assume that the state information s_i is sufficient to define the subsequent behavior that is present in the trace $sub(\sigma, i, n)$. This implies that removal of a single cycle $[l, u]$, $red(\sigma, \{[l, u]\}) = sub(\sigma, 0, l) \circ sub(\sigma, u, n)$ yields a possible trace due to the presence of $s_l = s_u$ in it. We can iterate this argument to argue that $red(\sigma, \mathcal{C}')$ yields a possible trace if cycles in $\mathcal{C}'$ are non-overlapping.

The validity of this assumption depends on the modeling formalism in use and the amount of information exported by the simulator into the trace file. It is usually fulfilled in untimed automata if $s \in S$ characterizes the state of an automaton and also in Markov models since the current state by definition defines the potential future behavior in a Markov process. For Petri nets, state information is given by the markings of all places, possibly extended by some supplementary variables depending on the kind of Petri net employed. For certain modeling formalisms, like stochastic well-formed nets (SWNs) and Rep/Join or graph composed stochastic activity networks (SANs) in Möbius[4], symmetries in the model description can be used to establish a performance bisimulation which yields a notion of equivalence for states. Such an equivalence can be used as "=" for the identification of cycles as well. We point this out to note that "=" is not necessarily limited to identity. However, in the general case, i.e., discrete event simulation of non-Markovian models, the state of a simulator takes the current event list, which is not necessarily represented in full in a trace file and is likely to show no repetitive behavior. However, exporting a trace from a simulator allows for abstractions. It comes with the degree of freedom to select which parts of a state description are considered relevant or of interest to a modeler. So we assume for the following that the state information in σ is chosen to be detailed enough such that a reduced trace σ^* is a possible outcome of a simulation, yet to be coarse enough to allow for cycles in a simulation.

Note that events are irrelevant in the following formal treatment, but events are important pieces of information in a trace in order to document not only the state of the system but also what happens. Hence, we keep events within our considerations. We formally define our reduction problem as follows.

Definition 1. *The sequence reduction problem (SRP) is the problem to determine a set of non-overlapping cycles $\mathcal{C}^* \subseteq \mathcal{C}$ for σ, such that $red(\sigma, \mathcal{C}^*)$ is of minimal length.*

Let $SRP(\sigma) = \mathcal{C}^*$ denote the solution of SRP for a particular σ and $\sigma^* = red(\sigma, \mathcal{C}^*)$. Note that $SRP(\sigma)$ is not necessarily unique; multiple solutions may exist that yield reduced traces of same length.

We restrict SRP to select only non-overlapping cycles and we need to discuss the reason for this restriction. Let $c_1 = [l_1, u_1]$, $c_2 = [l_2, u_2]$, $c_1 \neq c_2$ be two cycles in σ. If these are non-overlapping, then we can remove both cycles from σ in any order $red(\sigma, \{c_1, c_2\}) = red(red(\sigma, \{c_i\}), \{c_j\})$ with $i, j \in \{1, 2\}, i \neq j$ and yield the same reduced trace. If the cycles are overlapping and nested, e.g. c_2 is nested in c_1, $l_1 \leq l_2 < u_2 \leq u_1$ then we can only remove c_2 before c_1 and $red(\sigma, \{c_1, c_2\}) = red(red(\sigma, \{c_2\}), \{c_1\}) = red(\sigma, \{c_1\})$. Obviously, there is no need for a nested cycle like c_2 in $\mathcal{C}^*$ if $c_1 \in \mathcal{C}^*$. Finally, if c_1 and c_2 overlap, i.e., $(l_1 < u_2) \wedge (l_2 < u_1)$, then removing both cycles from σ would leave us with a transition $s_{l_i} e_{u_j+1} s_{u_j+1}$ in the resulting sequence (if $u_j < n$), where $l_i = min(l_1, l_2)$, $u_j = max(u_1, u_2)$. If the cycles are not nested, such a transition is not present in σ and would introduce new dynamic behavior, which is contrary to the idea that the reduction operation should yield a reduced, yet possible trace for a given model. In summary, selecting only non-overlapping cycles for the solution of SRP is natural for the reduction operation we are interested in.

We also restricted SRP to focus on elementary cycles $\mathcal{C}$. The reason for this is that for any non-elementary cycle $[l, u]$ in σ there exists a sequence of $m > 1$ elementary and non-overlapping cycles $[l_1, u_1]_1, [l_2, u_2]_2, \ldots, [l_m, u_m]_m$ in σ with $l = l_1, u = u_m$ and $s_l = s_{l_1} = s_{u_1} = \cdots = s_{l_m} = s_{u_m} = s_u$ that describes the same substring of σ, i.e., $sub(\sigma, l, u) = sub(\sigma, l_1, u_1) \circ sub(\sigma, l_2, u_2) \ldots \circ sub(\sigma, l_m, u_m)$. Consequently, restricting SRP to elementary cycles will give us the same amount of reduction as we could obtain with non-elementary cycles for the price of a set $\mathcal{C}^*$ that may have a larger cardinality than one with non-elementary cycles. We see two advantages of this restriction, the one is $|\mathcal{C}| \leq |\mathcal{C}_{all}|$, the other is that for any state s_u in σ there is at most one elementary cycle $[l, u]$ that ends at s_u. The latter will be beneficial to develop of a recursive solution to SRP.

To measure the reduction we can achieve on σ, we define the following function.

Definition 2. *For a trace σ, we define $p_\sigma(i) = |red(\sigma_i, SRP(\sigma_i))|$ as the progress of σ.*

$p_\sigma(i)$ is well-defined since the length $|red(\sigma_i, SRP(\sigma_i))|$ that is achieved by a maximal reduction is unique even if $SRP(\sigma_i)$ has several solutions.

Table 1 illustrates the concept for an example trace σ with S being capital letters, $E = \{e\}$ and $\mathcal{C} = \{[2, 4], [3, 9], [5, 7], [8, 11]\}$. Column i identifies which prefix σ_i of σ is considered, column $SRP(\sigma_i)$ gives the corresponding solution of SRP, column σ_i^* shows the reduced trace and column $p_\sigma(i)$ gives the amount of reduction, e.g., in line $i = 4$, we consider prefix $\sigma_4 = A_0 e B_1 e C_2 e D_3 e C_4$ and remove cycle $[2, 4]$ to achieve the reduced prefix $A_0 e B_1 e C_2$, which has length $p_\sigma(i) = 2$. The table also shows that $SRP(\sigma_{i+1})$ does not necessarily contain $SRP(\sigma_i)$, (lines for $i = 8$ and 10). Nevertheless, $SRP(\sigma_{i+1})$ could use a solution of

i	$SRP(\sigma_i)$	σ_i^*	$p_\sigma(i)$
0	$\emptyset$	A_0	0
1	$\emptyset$	A_0eB_1	1
2	$\emptyset$	$A_0eB_1eC_2$	2
3	$\emptyset$	$A_0eB_1eC_2eD_3$	3
4	$\{[2,4]\}$	$A_0eB_1eC_2$	2
5	$\{[2,4]\}$	$A_0eB_1eC_2eE_5$	3
6	$\{[2,4]\}$	$A_0eB_1eC_2eE_5eF_6$	4
7	$\{[2,4],[5,7]\}$	$A_0eB_1eC_2eE_5$	3
8	$\{[2,4],[5,7]\}$	$A_0eB_1eC_2eE_5eG_8$	4
9	$\{[3,9]\}$	$A_0eB_1eC_2eD_3$	3
10	$\{[3,9]\}$	$A_0eB_1eC_2eD_3eH_{10}$	4
11	$\{[2,4],[5,7],[8,11]\}$	$A_0eB_1eC_2eE_5eG_8$	4

TABLE 1
$SRP(\sigma_i)$, σ_i^*, and $p_\sigma(i)$ for
$\sigma = A_0eB_1eC_2eD_3eC_4eE_5eF_6eE_7eG_8eD_9eH_{10}eG_{11}$

$SRP(\sigma_i)$, so any reduction of σ_{i+1} removes at least as many states as the reduction of σ_i. With this observation and Def. 2, we can state some properties of $p_\sigma(i)$.

For any trace σ and $0 \leq i \leq n$,

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

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

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

If we understand σ as a random walk through the state space of a given model and assume that the reduction operation $red()$ results in a possible sequence of events, then $p_\sigma(i)$ gives an upper bound on the minimal number of events it takes to reach a state s_i from the initial state s_0. $p_\sigma()$ gives only an upper bound since it is based on the limited and in general incomplete information on the possible dynamic behavior of the considered simulation model. A plot of $p_\sigma(i)$ for $0 \leq i \leq |\sigma|$ provides us with an estimate on the depth of the state space, e.g., as an average value obtained from p_σ, and with some insight if the trace has a tendency to go deeper and deeper into a state space of unknown depth or to stay within some bounds. The latter may be caused by the limited depth of a given state space or by the way the simulator proceeds. In Sections 3 and 7 we discuss how a plot of p_σ can be used to debug simulation models.

Related work. SRP is related to the problem of cycle detection in periodic functions for which linear time algorithms with low memory requirements are long known [16, 20]. There, the problem is to analyze a function $f : D \rightarrow D$ on some domain D and to decide if f has a finite leader $x, f^1(x), f^2(x), \ldots, f^l(x)$ of length l where all values are different and a cycle of length c such that $f^i(x) = f^{i+c}(x)$ for all $i \geq l$. The problem does not match well with simulation traces, i.e., if $s_i = s_{i+c}$, then it is by no means guaranteed that $s_{i+j} = s_{i+j+c}$ will hold for $j > 0$ in a simulation trace. Note that algorithms for cycle detection address the problem to determine the starting point l of the first cycle and the cycle length c while SRP is a selection problem that selects a particular set of cycles $\mathcal{C}^*$ from the set of cycles $\mathcal{C}$. These problems are different by nature. So classical algorithms like [16, 20] do not solve SRP; nevertheless it is straightforward to adapt them to compute an approximate solution of SRP. We demonstrate this for the algorithm of Nivasch [16] in Section 5. The drawback is that the approximation error is unknown. With the help of our exact algorithm we are able to measure the approximation error of approximate algorithms. We do so for the two approximate algorithms we describe in Section 5 with a set of examples we evaluate in Section 6.

There is also work on interval graphs, e.g. [7], which considers graphs whose vertices can be mapped to distinct intervals in the real line such that the vertices in the graph have an edge between them if and only if their corresponding intervals overlap. However, the problems that have been considered in that area (to the best of our knowledge) did not match SRP, e.g., the minimum cover problem for node-weighted interval graphs considers the problem to find a subset of intervals (with smallest sum of weights) from a given set of intervals with smallest value l_{min} and largest value u_{max} such that the subset covers $[l_{min}, u_{max}]$, [7]. So, an overlap of intervals is acceptable although not preferred (minimum sum of weights) in that context, while in SRP intervals need to be non-overlapping.

At this point, we formulated the problem that we address. We claim that in some stochastic dependability models, such cycles are indeed present (we give evidence of that with the help of several examples in Sections 3 and 6) and that we also observed $|\mathcal{C}_{all}| >> |\mathcal{C}|$ (we show this effect for the example discussed in Section 3). We present an application example to motivate our approach before we discuss algorithmic solutions for SRP.

3 An Example Dependability Model

We consider a dependability model of a server that is subject to failure and repair. Let failures happen mainly due to software failures and be handled by rebooting the system and restarting tasks that are performed. The model is a closed queueing model with two customer classes A and B and corresponding finite population $N(A)$ and $N(B)$. Class B is used to describe "normal" customers that give the baseline utilization. For customers of class A, we want to measure the probability that their service takes place without failures and in a timely manner to accommodate certain service level agreements for quality of service. We do not provide further details of the timing because we will focus on debugging a corresponding simulation model which we develop with Möbius [4]. We choose the Möbius' SAN formalism to model each aspect of our system individually and the composition operation that is based on shared variables to join the individual atomic models into an overall model. Figure 1 shows the compositional structure of the model. The *FullModel* combines a *CompleteServer* submodel of the server with submodels *UserA* and *UserB*, which describe the user behavior. Users interact with the server by changing values of variables for the occupation of input and output buffers at the server. Those variables are shared via the *FullModel*. The *CompleteServer* submodel encapsulates how a service is performed and a submodel *failureAndRepair* that models availability of the server. The failure and repair model describes a cyclic behavior and switches between "available" and "failed" states to indicate the status of the server. It shares its boolean state variables "avail" and "failed" with the server model. The server models a queue with random scheduling and no preemption for two customer classes. If a customer is served when a failure occurs, its service is interrupted and it is positioned back into the queue. Its service time is not memorized. Customers cycle between the server and their own class-dependent submodel that delays them for a thinktime. A simulation run reveals measures that in the long run deviate from what is expected. For instance, the throughput of class B is slightly too low, for A slightly too high. At one point, we suspect that the model contains an error. Möbius allows us to generate a trace where variable settings yield state information and changes yield events. So we downscale the model, as a first try with respect to populations $N(A) = 1$ and $N(B) = 1$, to check what happens and generate a trace σ. Following a simple pragmatic heuristic, we created a trace with a few thousand states, here with $n = 5473$ events, with the hope to cover relevant behavior for a model of this size and timing characteristics.

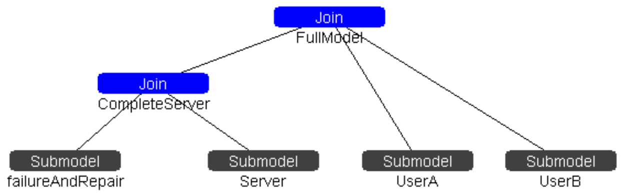


Fig. 1. Composed Model

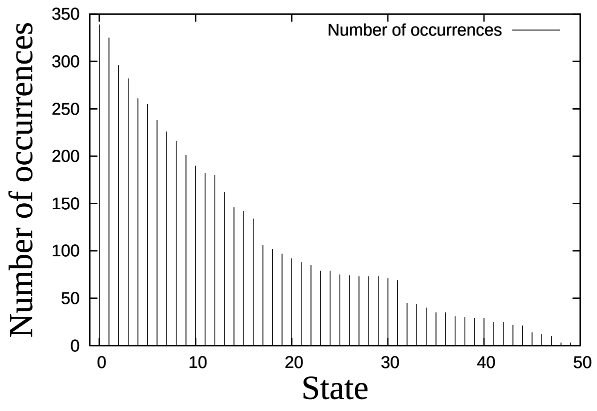


Fig. 2. Number of occurrences of states

Cycles do exist. Before we address how to find the error by cycle reduction, let us observe to what extent cycles are indeed present in the trace. Figure 2 gives the number of occurrences for certain states in σ. States

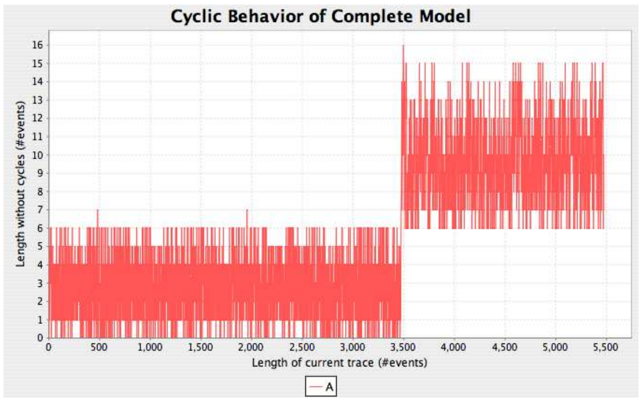
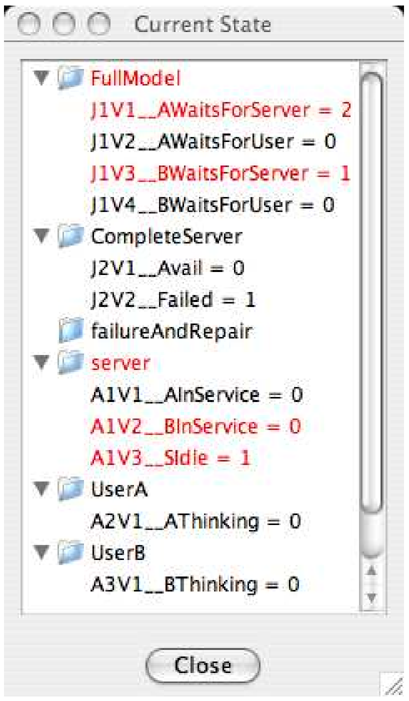


Fig. 3. Length of trace after cycle reduction as a function of $|\sigma| = n$


Current State

FullModel
J1V1__AWaitsForServer = 2
J1V2__AWaitsForUser = 0
J1V3__BWaitsForServer = 1
J1V4__BWaitsForUser = 0
CompleteServer
J2V1__Avail = 0
J2V2__Failed = 1
failureAndRepair
server
A1V1__AInService = 0
A1V2__BInService = 0
A1V3__SIdle = 1
UserA
A2V1__AThinking = 0
UserB
A3V1__BThinking = 0

Fig. 4. State right after erroneous event

are ordered by decreasing number of occurrences. For instance, the first state in the figure occurs 339 times in σ. The impact on the cardinality of $\mathcal{C}_{all}$ is significant. If a state occurs k times, that state alone generates $k \cdot (k-1)/2$ elements of $\mathcal{C}_{all}$. The calculation reflects the number of possible selections of two states among k states where permutations count only once. For $k = 339$, that state contributes 57291 intervals to $\mathcal{C}_{all}$, 338 to $\mathcal{C}$. Note that from a conceptual point of view the presence of cycles is natural in dependability and performability models with failure and repair. This model is a set of finite submodels that communicate via shared state variables. The behavior of each submodel is cyclic or repetitive, and we are interested in the behavior of the composed model with respect to the timing of certain events.

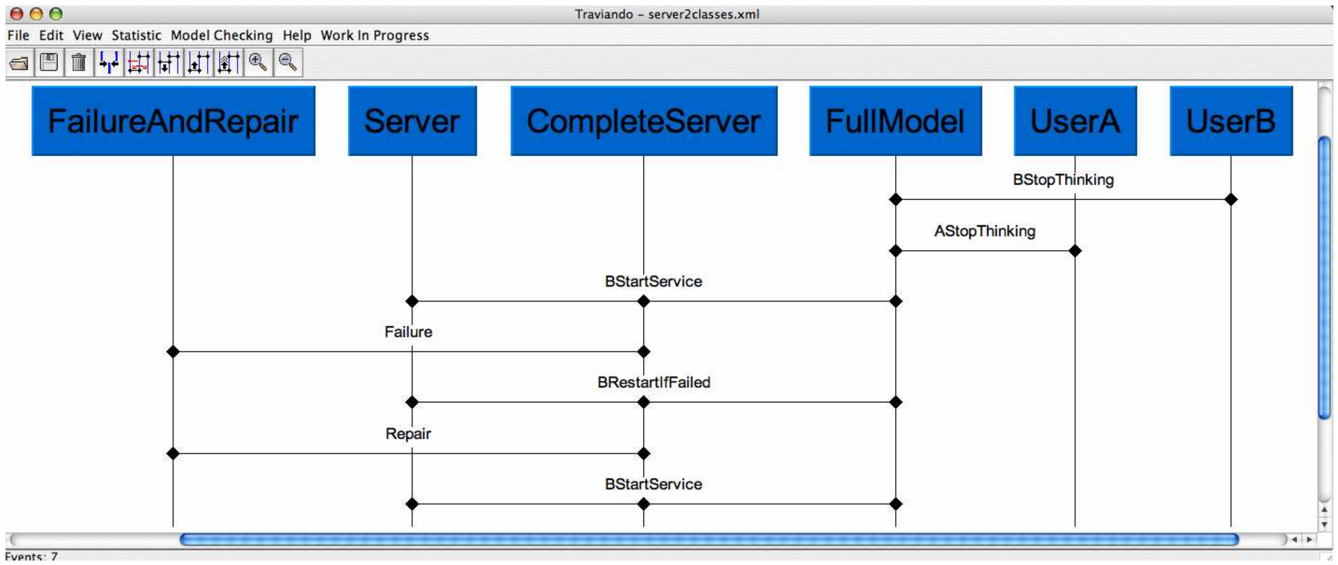


Fig. 5. Trace visualization of reduced trace σ^* in Traviando

Searching for errors. Working with cycle reduction has two features that we consider useful. One helps us to recognize if a certain type of error is present in σ, the other helps us to identify which events are used to reach a particular state of interest. Traviando [11] is a trace visualization and analysis tool that supports the cycle reduction we propose. Figure 3 shows a screen shot of $p_\sigma(i)$ for $0 \leq i \leq n$. The initial part of the plot shows that the model proceeds and returns to states in a cyclic manner for the first 3500 events, then an event creates a state change that does make a permanent difference, after which the model proceeds and returns to states in a similar cyclic manner. However that particular state change is never taken back. Fig. 3 tells us that this trace contains behavior that is irregular and also where to look for it, namely around the 3514-th event where the switch took place. Traviando provides information on any selected state in a separate window and in a directory-like, graphical structure as shown in Fig. 4. In this way, we can check values of state variables of a state $s_i, i > 3514$ and see that $N(A)$ has increased by 1 which violates that the model has an invariant customer population for customers of class A. To search for its cause, we can either track back the changes looking at σ or at σ^*. For this example, σ^* is preferred since $|\sigma^*| = 7$. With those few states and events and a trace analyzer tool like Traviando, it is immediate to recognize the root of the error. Figure 5 shows Traviando's visualization of σ^* as a variant of a message sequence chart (MSC) with one MSC process per submodel, dots on the lifeline of a process indicate local actions, undirected horizontal lines connecting several processes show joint actions. With this information, we realize that the reason is an unwanted side effect of the action that puts a customer of type B back into the queue if the server fails while serving a customer of class B; the resulting state has a value of 2 for the number of customers of class A that wait for the server (variable J1V1_AWaitsForServer for process FullModel in Fig. 4, variables are highlighted if they change value from the last state selected to the one currently shown). The error was induced when we extended the model from one customer class to two customer classes. When we fix the error, the corrected model creates traces that give a plot of $p_\sigma(i)$ as in the initial part of the one in Fig. 3.

In what follows, we investigate algorithms for an exact or approximate solution of SRP to support this way of debugging stochastic models.

4 Exact Solution of SRP

In this section, we describe an exact linear time algorithm for SRP that is based on the following observations. An optimal solution C^* is a sequence of non-overlapping cycles. Let $C^* = \{[l_1, u_1], \ldots, [l_m, u_m]\}$ be ordered such that $u_{c-1} < u_c$ for $1 < c \leq m$ (we will use c as an index for cycles, i as an index for states in σ). Note that $u_{c-1} \neq u_c$ since cycles do not overlap. If we select a cycle $[l_c, u_c] \in C^*$ then $\{[l_1, u_1], \ldots, [l_{c-1}, u_{c-1}]\}$ is an optimal solution for $s_0, \ldots, s_{l_c}$ and $\{[l_{c+1}, u_{c+1}], \ldots, [l_m, u_m]\}$ is an optimal solution for $s_{u_c}, \ldots, s_n$. If we focus on the former, then the optimal solution on $s_0, \ldots, s_{l_c}$ is a subproblem that has an optimal solution on a set of cycles $[l_j, u_j] \in \mathcal{C}$ with $u_j \leq l_c$. This observation helps us to sequentially solve a sequence of SRP problems[1] for sequences σ_i for $0 \leq i \leq n$. We obtain a recursive procedure to solve SRP for σ and subsequently employ a dynamic programming approach based on a Bellman equation [3]. Note that for elementary cycles, for all $0 \leq i \leq n$ there is at most one $[l, u] \in \mathcal{C}$ with $u = i$.

1. We also investigated the possibility of a binary partitioning strategy but did not get a better result.

Definition 3. *For a given trace σ of length n, we define function $SRP(\sigma_n)$*

$$= \begin{cases} \emptyset & \text{if } n = 0 \\ SRP(\sigma_l) \cup \{[l,n]\} & \text{if } n > 0, [l,n] \in \mathcal{C}, a \leq b+1 \\ & a = |red(\sigma_l, SRP(\sigma_l))| \\ & b = |red(\sigma_{n-1}, SRP(\sigma_{n-1}))| \\ SRP(\sigma_{n-1}) & \text{otherwise} \end{cases}$$

Theorem 1. *$SRP(\sigma)$ of Definition 3 gives a correct solution of SRP for a given trace σ.*

Proof: For $n = 0$, $\mathcal{C} = \emptyset$ and the result is correct. For $n > 0$, we need to prove two properties: 1) $SRP(\sigma_n)$ yields a set of non-overlapping, elementary cycles and 2) $red(\sigma_n, SRP(\sigma_n))$ is of minimal length.

The first property is straightforward. We see that only the second case adds elements to set $SRP(\sigma_n)$ and since $[l,n] \in \mathcal{C}$, the resulting set contains only elementary cycles. Furthermore, the resulting set $SRP(\sigma_n)$ is non-overlapping, because in the second case, the recursion proceeds with σ_l so cycles added to the solution in the recursion do not overlap with $[l,n]$, and the third case only applies if no cycle is added.

It remains to prove the second property (the reduction is maximal), which we do by contradiction. Let σ be a counterexample with smallest possible $n > 0$ such that there is a set $\tilde{C} \subseteq \mathcal{C}$ with $|red(\sigma_n, \tilde{C})| < |red(\sigma_n, SRP(\sigma_n))|$. Since n is the smallest possible value, this implies $SRP(\sigma_i)$ for $0 \leq i < n$ is a correct solution of SRP for σ_i. We consider the last state s_n in σ_n and two cases: a) there exists a cycle $c = [l,n] \in \mathcal{C}$ and b) such a cycle does not exist.

Case a)
If c exists, we distinguish four cases based on whether c is an element of $SRP(\sigma_n)$ and/or $\tilde{C}$ or not.

Case a1: $c \in SRP(\sigma_n)$ and $c \in \tilde{C}$
This implies $red(\sigma_n, \{[l,n]\}) = \sigma_l$ and hence $|red(\sigma_n, SRP(\sigma_l) \cup \{[l,n]\})| = |red(\sigma_l, SRP(\sigma_l))|$. Since n is the smallest possible value, $SRP(\sigma_l)$ is correct and $|red(\sigma_n, SRP(\sigma_l) \cup \{[l,n]\})| = |red(\sigma_l, SRP(\sigma_l))| \leq |red(\sigma_l, \tilde{C} \backslash \{[l,n]\})| = |red(\sigma_n, \tilde{C})|$ which contradicts our assumption that $\tilde{C}$ yields a shorter trace.

Case a2: $c \in SRP(\sigma_n)$ and $c \notin \tilde{C}$
$\tilde{C} \subseteq \mathcal{C}$, c is an elementary cycle and c is the only elementary cycle that ends at s_n. So if $c \notin \tilde{C}$, then s_n remains and must be the last state of $red(\sigma_n, \tilde{C})$. So $|red(\sigma_n, \tilde{C})| = |red(\sigma_{n-1}, \tilde{C})| + 1$. However, $c \in SRP(\sigma_n)$, which implies $|red(\sigma_n, SRP(\sigma_l) \cup \{[l,n]\})| = |red(\sigma_l, SRP(\sigma_l))|$. c is added to $SRP(\sigma_n)$ only if $a \leq b+1$, i.e., $|red(\sigma_l, SRP(\sigma_l))| \leq |red(\sigma_{n-1}, SRP(\sigma_{n-1}))| + 1$. Since $SRP(\sigma_{n-1})$ is correct, $|red(\sigma_{n-1}, SRP(\sigma_{n-1}))| \leq |red(\sigma_{n-1}, \tilde{C})|$ which if put altogether contradicts our assumption that $\tilde{C}$ yields a shorter trace.

Case a3: $c \notin SRP(\sigma_n)$ and $c \in \tilde{C}$
$c \notin SRP(\sigma_n)$ implies that $|red(\sigma_n, SRP(\sigma_n))| = |red(\sigma_n, SRP(\sigma_{n-1}))| = |red(\sigma_{n-1}, SRP(\sigma_{n-1}))| + 1$ and $SRP(\sigma_{n-1})$ is correctly given. However, $|red(\sigma_n, \tilde{C})| = |red(\sigma_l, \tilde{C} \backslash \{[l,n]\})| \geq |red(\sigma_l, SRP(\sigma_l))| \geq |red(\sigma_{n-1}, SRP(\sigma_{n-1}))| + 1$, where the first inequality is true because we assumed n is minimal and and the second is implied by the condition for $c \notin SRP(\sigma_n)$ in Def. 3.

Case a4: $c \notin SRP(\sigma_n)$ and $c \notin \tilde{C}$
Since c is the only cycle that ends at n, $c \notin SRP(\sigma_n)$ and $c \notin \tilde{C}$, there is no other way to remove s_n from σ, so it has to remain and $|red(\sigma_n, SRP(\sigma_n))| = |red(\sigma_{n-1}, SRP(\sigma_{n-1}))| + 1$ and $|red(\sigma_n, \tilde{C})| = |red(\sigma_{n-1}, \tilde{C})| + 1$. But since $SRP(\sigma_{n-1})$ is correct, we have $|red(\sigma_{n-1}, SRP(\sigma_{n-1}))| + 1 \leq |red(\sigma_{n-1}, \tilde{C})| + 1$ which contradicts our assumption.

Case b)
If there is no cycle $[l,n]$, then the last state s_n cannot be removed and remains in the reduced sequence independently on the selection of $\mathcal{C}^*$. Hence $|red(\sigma_n, SRP(\sigma_n))| = |red(\sigma_{n-1}, SRP(\sigma_{n-1}))| + 1$ and $|red(\sigma_n, \tilde{C})| = |red(\sigma_{n-1}, \tilde{C})| + 1$. But since $SRP(\sigma_{n-1})$ is correct, we have $|red(\sigma_{n-1}, SRP(\sigma_{n-1}))| + 1 \leq |red(\sigma_{n-1}, \tilde{C})| + 1$ which contradicts our assumption.
In summary, there is no way the assumed $\tilde{C}$ can exist. □

```
AOPT($\sigma$)
 0  $p[0] = u[0] = l[0] = w[0] = c = 0$; $n = |\sigma|$;
 1  $h$ = empty hash map; $\mathcal{C}' = \emptyset$; $\sigma' = \sigma$;
 2  for $i = 0$ to $n$ with stepsize 1
 3      if ($h$ contains $s_i$)
 4      then // cycle identified
 5          $c = c + 1$;
 6          $u[c] = i$;
 7          $(l[c], k)$ = getValue($h, s_i$);
 8          if ($w[c-1] < w[k] + u[c] - l[c]$)
 9          then // consider new cycle
10              $w[c] = w[k] + u[c] - l[c]$;
11              $p[c] = k$;
12          else // ignore new cycle
13              $w[c] = w[c-1]$;
14              $p[c] = c - 1$;
15          setValue($h, s_i, (i, c)$);
16      else // no cycle yet, just add state
17          addValue($h, s_i, (i, c)$);
18      $p_\sigma(i) = i - w[c]$; // optional: compute progress of $\sigma$
19  while ($0 < c$)
20      if ($w[c] \neq w[c-1]$)
21      then $\sigma' = red(\sigma', \{[l[c], u[c]]\})$; $\mathcal{C}' = \mathcal{C}' \cup \{[l[c], u[c]]\}$
22      $c = p[c]$;
23  return $\sigma', \mathcal{C}'$
```

Fig. 6. Algorithm AOPT

Fig. 6 gives a detailed pseudocode description of an iterative algorithm AOPT that implements $SRP(\sigma_n)$ of Def. 3 and avoids the recursion. It computes $\mathcal{C}' = \mathcal{C}^*$ and $\sigma' = red(\sigma, \mathcal{C}')$. AOPT iterates through states s_i of σ in a forward manner (lines 2-17) and adds s_i to a hash map h (line 17) to be able to identify cycles (line 3). In order to consider elementary cycles, the index of a state s_i in h is updated to the new value i if a cycle is found that ends in s_i (line 15) to prepare for the next time that state may be seen again. So AOPT visits all cycles in $\mathcal{C}$ in an ordered sequence of increasing values u_c when $i = u_c$, and for each cycle $[l_c, u_c]$ we base our decision whether to consider it for the optimal solution of SRP for σ_{u_c} by comparing the optimal solution for σ_{u_c-1} with the reduction achieved by the optimal solution for σ_{l_c} plus the contribution of $[l_c, u_c]$ in line 8. We memorize the better variant of the two as a solution of σ_{u_c}. AOPT uses four arrays (l, u, w, p) and one hash map h as data structures. Let $\mathcal{C}$ be ordered such that $u_{c-1} < u_c$, then AOPT stores the c-th elementary cycle $[l_c, u_c]$ of σ in $l[c] = l_c, u[c] = u_c$. Note that $|red(\sigma_i, SRP(\sigma_i))| = i - \sum_{j \in SRP(\sigma_i)} u_j - l_j$, since the cycles in $SRP(\sigma_i)$ are non-overlapping. Entry $w[c] = \sum_{j \in SRP(\sigma_{u_c})} u_j - l_j$ stores how many states can be removed and the optional step in line 18 derives $p_\sigma(i)$ with the help of $w[c]$. Note that the entry $(l[c], k)$ obtained from h in line 7 is used to make sure that we select the corresponding cycle with index k, that is closest to l_c such that $u_k \leq l_c$. Entries in $p[]$ form chains of downward references towards 0. Together with entries of $w[]$, they characterize elements of C^* for the sequence of SRP problems for sequences σ_i for $0 \leq i \leq n$. Note that values of c only increase in that loop, let c_{max} denote the maximal value for c that we observe in AOPT, ($c = c_{max}$ when the for-loop in lines 2-17 finishes after $i = n = |\sigma|$). Hashmap h stores tuples $(s_i, (i, k))$ with s_i being key, (i, k) being the value where k is the index of a cycle that s_i corresponds to in arrays w, l, u, p. In line 15, an existing entry in h is updated, while in line 17 a new entry is added to h.

For a detailed proof of the correctness of AOPT, we refer to [12]. The worst case time complexity of AOPT is in $O(n)$ for $|\sigma| = n$ if search, insert and change operations on a hash map are in $O(1)$. The time complexity follows from the observation that n states are considered, $c_{max} \leq n$ cycles are identified. The removal of at most c_{max} non-overlapping intervals in a decreasing order can be performed in $O(n)$ with one iteration through the array of n states of σ, e.g. by copying all remaining elements to a new array of length $n - w[c_{max}]$ (if σ is stored in an array), or by removing a sequence of individual elements in decreasing order (if σ is stored in double linked list) with a total cost of one run through σ. The optional computation of $p_\sigma(i)$ in line 18 lets us compute $p_\sigma(i)$ for $0 \leq i \leq n$ in $O(n)$ as a byproduct of performing AOPT for σ, which is better than a naive $O(n^2)$ computation of $p_\sigma(i)$ for $0 \leq i \leq n$ that applies AOPT to each prefix of σ individually.

The space complexity is $n(5 + size(s))$ where $size(s)$ is the space needed to represent a single state and integer in the hash map h and $5n$ reflects on the 4 integer arrays plus one hash map that are used. We assume that $p_\sigma(i)$ (line 18) does not take any additional space and values of $p_\sigma(i)$ are given as output as they are computed.

Related Approaches. SRP can be encoded by a directed acyclic graph with edge-weights such that the longest path in that graph yields the solution of SRP [15]. We only briefly sketch the concept because it is inferior to the presented approach. Let $G = (V, E)$ be a directed graph with a set of nodes $V = C \cup \{\perp\}$. C are the elementary cycles of a given trace σ and $\perp$ is an extra artificial sink node where paths in G will end. The set of edges $E \subseteq V \times V$ contains all tuples $e = (v, v')$ with $v = [l, u]$ and $v' = [l', u']$ where $u \leq l'$ and all tuples $(v, \perp), v \in C$. An edge $e = (v, v')$ has a weight $w(e) = w([l, u], v') = u - l$ based on the length of the cycle encoded by the starting node v. Note that 1) each path in the graph gives a sequence of non-overlapping elementary cycles and the sum of the weights of its edges gives the amount of reduction if those cycles are removed from σ, and 2) the longest path will end in $\perp$ since all nodes have an outgoing edge to $\perp$. So we can compute the longest path in G to obtain an optimal solution C^* of SRP. Note that G is acyclic since for any $v = [l, u]$ it holds $l < u$ and for any edge $e = ([l, u], [l', u'])$ it holds $u \leq l'$, so values of l along a path need to increase and cannot repeat. Although efficient linear time algorithms are known for the computation of the longest path in directed acyclic edge-weighted graphs [21], our experimental findings indicated that the size of the graph (number of nodes and edges) makes this approach inefficient and we could not make it scale for long traces with a large number of cycles. The proposed algorithm AOPT is superior to this approach.

5 Approximate Solution of SRP

Since AOPT's time complexity is linear and one cannot do less than reading input σ for an optimal reduction, we only look for approximate solutions of SRP that result in a valid reduction but not necessarily a maximal reduction of σ and that perform with at most same time complexity but less space.

```
AC(σ)
 0  $h$ = empty hash map;
 1  for i = 0 to n with stepsize 1
 2      if ($h$ contains $s_i$)
 3      then
 4          j= getValue($h, s_i$);
 5          $\mathcal{C}' = \mathcal{C}' \cup \{[j, i]\}$;
 6*         update($h, s_i, i$)
 7      else
 8          add($h, s_i, i$);
 9  return $\mathcal{C}'$;
```

Fig. 7. Algorithm to generate $\mathcal{C}' \subseteq \mathcal{C}$

It is fairly straightforward to come up with an algorithm that detects cycles and creates a set $\mathcal{C}' \subseteq \mathcal{C}_{all}$. Fig. 7 gives the pseudocode of an algorithm that iterates through states in σ, adds tuples (s_i, i) to a hash map h with s_i being the key, i being the value of that mapping which is returned by getValue in line 4. The step in line 6 is optional. If it is performed, $\mathcal{C}'$ will contain elementary cycles only, due to the change of value for entry (s_i, i) in h (as in AOPT). If it is skipped, $\mathcal{C}'$ may contain non-elementary cycles and in particular the largest non-elementary cycles in $\mathcal{C}_{all}$. Due to its simplicity, we do not formally prove termination and correctness. Based on $\mathcal{C}'$, we obtain two approximate solutions of SRP.

The first solution is a greedy strategy that removes cycles by weight. Given $\mathcal{C}'$, it sorts elements $[l, u]$ of $\mathcal{C}'$ by weight $u - l$, iterates through $\mathcal{C}'$ in decreasing order and successively removes cycles $[l, u]$ from σ if possible, i.e. $\sigma = sub(\sigma, 0, l) \circ sub(\sigma, u, n)$ if both states s_l and s_u are still present in the current σ (and none of them have been removed in a previous reduction). The time complexity is at least $O(n)$ for creating $\mathcal{C}'$ and $O(|\mathcal{C}'|log|\mathcal{C}'|)$ for sorting $\mathcal{C}'$. Hence, we consider this approach inferior to AOPT and do not investigate this algorithm any further.

The second strategy is a greedy strategy that selects cycles in the order of occurrence (first come first served), which we denote as AFCFS. It is an on-the-fly approach that runs through σ from the beginning, creates a set $\mathcal{C}'$ and removes cycles from σ (and corresponding states from hashmap h) as soon as a cycle is identified. So we formally introduce σ^c to denote the reduced sequence.

```
AFCFS($\sigma$)
 0  $i = c = 0; \sigma^c = \sigma$
 1  $h$ = empty hashmap;
 2  while ($i \leq |\sigma^c|$)
 3      i++;
 4      if ($h$ contains $s_i$)
 5      then // interval identified, remove
 6          $c = c + 1$;
 7          $l_c$ = getValue($h, s_i$);
 8          for $j = l_c + 1$ to $i - 1$ with stepsize 1
 9              removeByValue($h, j$) ;
10          $\sigma^c = red(\sigma^{c-1}, \{[l_c, i]\})$;
11          $\mathcal{C}' = \mathcal{C}' \cup \{[l_c, i]\}$;
12          $i = l_c$;
13      else
14          add($h, s_i, i$);
15  return $\sigma^c, \mathcal{C}'$;
```

Fig. 8. Algorithm AFCFS

Due to the simplicity of the concept, we do not formally prove the approach. Compared to AOPT, AFCFS uses the same detection mechanism for cycles, but there is no need for arrays and we can reduce σ and hashmap h on-the-fly. Note that AFCFS identifies cycles that are either nested or non-overlapping. The obvious benefit is the immediate reduction of the space for σ and h.

A third approach is based on the work of Nivasch [16], which focuses on the detection of cyclic functions (sequences). The main argument is that if a sequence becomes cyclic and repeats a certain loop over and over again, i.e. $\sigma = s_0, \ldots, s_n$ with $i < j < k$ such that $sub(\sigma, i, j) = sub(\sigma, j, k)$ then it is sufficient to focus on $s_m = min\{s_l | i \leq l \leq j\}$ and there is a cycle $[s_m, s_{m+j-i}]$. We can define a total order on states to establish a minimum for a set of states in the following way. If $s_i = (s_{i0}, \ldots, s_{im})$ happens to be a vector where entries s_{ij} have a total order, we define the following order for states: $s_i < s_j$ if $\exists k$ such that $s_{il} = s_{jl}$ for $0 \leq l < k$ and $s_{ik} < s_{jk}$ for $k \leq m$, i.e., we use a lexicographic order. Given such an order, a stack of states is sufficient to memorize the min value. Fig. 9 describes the algorithm in pseudocode and adapted to solve SRP. Nivasch's approach is particularly dedicated to identify cyclic functions and to determine the cycle length, e.g., for random number generators, where it is very promising. Identical subsequences $sub(\sigma, i, j) = sub(\sigma, j, k)$ are naturally present in cyclic functions but this is not necessarily the case for traces obtained from simulation models. For SRP, ASTACK delivers a valid reduction but not necessarily an optimal solution. The algorithm makes use of a stack q and a total order of states. The stack contains a sequence

```
ASTACK($\sigma$)
 0  $q$ = empty stack;$\sigma' = \sigma; \mathcal{C}' = \emptyset$;
 1  for i=0 to n with stepsize 1
 2      while ($s_i$ <state(top(q))
 3          pop(q) ;
 4      if ($s_i$ == state(top(q))
 5      then // interval found
 6          $[l, u] = [index(top(q)), i]$
 7          $\mathcal{C}' = \mathcal{C}' \cup \{[l, u]\}$
 8          $\sigma' = red(\sigma', \{[l, u]\})$;
 9      else
10          push($q, (s_i, i)$);
11  return $\sigma', \mathcal{C}'$;
```

Fig. 9. Algorithm ASTACK

of states in a monotonously increasing order, function top(q) reads the top element (s_j, j) from stack q but does not remove it, push and pop are the usual stack operations, state(s_j, j) returns s_j, index(s_j, j) returns j. For considerations with respect to correctness, time and space complexity details we refer to [16]. Analogously to the adaptation of Nivasch's algorithm, one could adapt the algorithm by Sedgewick et al [20] which we do not follow here. As for AFCFS, ASTACK detects cycles that are either nested or non-overlapping.

6 Experimental Evaluation

In this section, we compare algorithms AOPT, AFCFS, ASTACK and the combined algorithm ACOMB, which denotes that AOPT is applied on the reduced output trace of ASTACK. The motivation for ACOMB is to use an approximate algorithm that is much lower in memory requirements than AOPT to reduce most of a trace and then apply the exact algorithm to squeeze remaining cycles out of the resulting sequence. Since the combination of AFCFS with AOPT did not yield any improvement in our experiments, we report only results for the combination of ASTACK and AOPT. All algorithms have been integrated in Traviando, such that we can make use of traces generated with three different modeling frameworks, namely Möbius [4], the APNN toolbox [5], and the ProC/B toolset [2]. Annotations in Table 2 relate modeling tools with example models.

Selection of Examples. We selected a number of examples from the literature to see how well the reduction applies in practice.

The *Courier* model refers to the Courier protocol model by Woodside and Li [22]. It is intended for performance analysis and generates a recurrent, finite Markov chain. It is a stochastic Petri net whose initial marking is chosen such that the model has a small state space and is expected to show a lot of cyclic behavior. *Production cell* denotes a large Petri net model by Heiner et al [6]. It considers the control of a production cell which consists of a rotating table, a robot with two arms, a press, a crane, and two transportation belts. The Petri net has 231 places, 202 transitions and generates a state space with more than a million states. We selected this model since the length of the state descriptor and the size of the state space makes it unlikely to observe many cycles. *DinPhils* refers to a model of the classical dining philosophers; it is a mere modeling exercise performed in the ProC/B toolset. *Store* refers to model of a storage area described in [10]. It is a ProC/B simulation model based on a process interaction approach. It models the transfer of goods into and out of a store by trucks that are allocated to ramps and that are loaded or unloaded with the help of forklifts that are manned with workers. It describes an open system. The state representation that is chosen for the trace abstracts from certain details of the simulation model. In particular, identities of entities are incorporated only as attributes of actions. The model has a defect in the sense that it reaches a situation, where the loading/unloading operations are all blocked and the only remaining activities are arrivals of newly generated entities, resp. trucks (since it is an open model). Its traces give little room for reduction. *Server* refers to the Möbius model of a server with failure and repair that we discussed in Section 3. *FaultyProc* is a failure model of a fault tolerant processor that is part of a set of example models available with the Möbius distribution. Similar to this, *Conveyor* refers to a model of a conveyor belt that is described in a process algebra supported by Möbius. *Database* refers to a Möbius model of a database system.

In summary, we selected a set of models with significant variation in $|S|$, $|E|$, $|\sigma|$, the modeling formalism, the generating simulator, the application area, and whether an error is present or not. In this way, we try to avoid a bias from considering specially engineered examples.

Achieved reduction. Since only AOPT gives exact results and it is at this point unclear if ASTACK and AFCFS yield good reductions in practice, we evaluate the reduction capabilities of all three algorithms on a set of traces we generated from the set of examples and in different lengths. Note that traces of different lengths are obtained from separate and independent simulation runs and shorter traces of one model are not necessarily a prefix of a longer trace of the same model. Table 2 gives the resulting values for $max \sum_{[l_i,u_i]\in\mathcal{C}'}(u_i - l_i)$ for the calculated set $\mathcal{C}'$ for algorithm AOPT in column 2, AFCFS in column 3, ASTACK in column 4, and results of the two phases of the combined algorithm separately, namely the reduction by ASTACK if applied first in column 4 and the reduction obtained by AOPT if applied to the output of ASTACK in 5. Note that the sum of values of columns 4 and 5 give the total reduction achieved by ACOMB. For example, AOPT achieves a value of 4938 in line 3, column 2 for model *Courier* with $|\sigma| = n = 5000$ that is $|\sigma^*| = 5000 - 4938 = 62$ and ASTACK gives a reduction of 4860 such that the combination of ASTACK and AOPT (sum of values in columns 4 and 5) gives $4926 = 4860 + 66$ which almost yields the reduction achieved by AOPT alone. Since AOPT guarantees to achieve a maximal reduction for σ, it yields the maximal possible reduction values in the table. Column n describes $|\sigma|$. Considering the reduction achieved by the different algorithms in Table 2, we observe a substantial reduction for all but the *Store* example. *Store* is a model of an open system where the population grows due to an internal deadlock for resource allocation and the absence of cycles indicates this problem. For other models, AOPT achieves a substantial reduction, so the cycles are indeed present, even in models like *Production cell* with a large state descriptor and a large state space. AFCFS often gets

			ACOMB	
n	AOPT	AFCFS	ASTACK	AOPT
APNN: Courier				
(∗) 5000	4938	4938	4860	66
10000	9966	9966	9912	54
APNN: Production cell				
1000	810	648	486	324
10000	9720	9720	1944	7776
160029	159894	159894	42120	117774
ProC/B: Dining Philosophers				
(∗) 67744	67721	67721	66441	1260
ProC/B: Store				
6140	65	65	65	0
Möbius: Server				
5473	5466	5462	5252	214
(∗) 583880	583410	583186	500504	82878
1170380	1169570	1169114	1001110	168428
Möbius: Fault Tolerant Processor				
4368	4352	4345	2687	1665
20961	20952	20952	16167	4785
Möbius: Conveyor System				
5391	5361	5334	1874	3487
(∗) 20160	20084	20048	19805	252
Möbius: Database System				
4732	4732	4732	4698	34
19974	19974	19974	19926	48

TABLE 2
Reduction achieved for examples

results that are equal or reasonably close. However, for the production cell we can recognize that the length σ may have an influence as well, e.g., AFCFS does well for certain values of n but not for others. ASTACK, if applied on its own (column 4), also yields a substantial reduction but occasionally falls behind significantly, e.g., for *Conveyor* and $n = 5391$, the fault tolerant processor and the production cell model. However, if combined with AOPT the overall results match with the one of AOPT for almost all cases. Symbol $(*)$ in rows of Table 2 indicate where the total effect of ACOMB, i.e., the sum of values in columns 4 and 5, do not give the maximal reduction as obtained by AOPT (column 2).

Note that those differences are in fact small. We observe that differences vary for the same model across traces of different lengths. In order to see if differences in the reduction capability depend on the selection of $|\sigma| = n$ we compare solutions for any prefix of a trace. Table 3 lists the maximum and average difference in length (rounded to two digits) between $|\sigma_i^*|$ for $i = 0, \ldots, n$ as computed by AOPT, ASTACK and AFCFS. For example, the entry in row *Courier* and column AFCFS-max is $66 = max\{|red(\sigma_i, C')| - p_\sigma(i)|0 \leq i \leq n\}$ where C' is the set of cycles removed from σ_i by algorithm AFCFS. The minimal difference is always 0 (at σ_0). Small values indicate a good approximation. We also measured the standard deviation, for AFCFS (ASTACK), it ranges between 0 (2.05) for *Store* and 73.97 (2396.89) for *Production cell*. The experimental results indicate that AFCFS performs consistently better than ASTACK for all examples with respect to the approximation error measured as max, mean, and standard deviation of the difference to the result of AOPT.

		AFCFS		ASTACK	
Model	n	max	avg	max	avg
Courier	10000	66	3.81	174	65.28
Prodcell	10000	324	44.46	7776	3890.99
DinPhils	67744	210	50.57	3060	1517.13
Store	6140	0	0.00	36	0.12
Server	5473	16	1.79	342	115.66
FaultyProc	20961	66	11.23	4956	2442.20
Conveyor	20160	198	45.89	4050	1460.97
Database	19974	2	0.00	94	46.73

TABLE 3
Approximation error

Comparison of computation times. The time complexity of all considered algorithms is linear in the length of σ, however this may imply significant differences in practice. In this section, we report on computation times measured on a Pentium PC running Linux with 2 CPUs (3.4GHz), 2MB cache and 2GB main memory. All times given are wall clock times in seconds.

We assume that the main characteristics that influence the performance are $|\sigma|$, $|\mathcal{C}|$ and the size of the state descriptor. $|\mathcal{C}|$ influences the effort necessary to perform calculations and reductions, so we consider two artificial and extreme scenarios in this respect: one that has no cycles and one that has a very regular structure with many cycles. The size of the state description influences the cost of memory management, instantiation of state objects, comparisons among states, evaluations of hash functions and so forth. The first two scenarios are extremely lightweight on state descriptors, so we consider a third scenario with the Production cell model that is particularly heavy with its state representation and that also has a substantial set of cycles to make reductions happen. Finally, we report results for the Server model as our fourth scenario since it contains many cycles and an error. We do not provide detailed results for the other models of Table 2 since all other traces mainly resulted in insignificant computation times of less than a second.

The first scenario considers traces with no cycles. We generated a trace σ with a single state variable that is successively incremented so $s_i = i, 0 \leq i \leq n$. So the space needed to represent a state and the cost involved to compare states or to compute a hash function is minimal. All algorithms but ACOMP performed in less than 1 s for traces $|\sigma| \leq 600,000$, ACOMP used 3.9 s for $|\sigma| = 600,000$. Note that this is also a worst case scenario for ASTACK, since values are increasing.

For the second scenario, we consider a trace that has a large number of simple cycles to detect and to remove such that $\sigma^* = s_0$. We generated σ with a single state variable and produced an alternating sequence of values $0, 1, 2, \ldots, 8, 9, 8, \ldots, 2, 1, 0, 1, 2, \ldots$. All algorithms performed within 1 s for $|\sigma| \leq 180,000$. For much higher values, e.g., for $|\sigma| = 540,000$, AFCFS performed best with 0.95 s, ASTACK and ACOMP both with about 1.3 s, and AOPT with 3.09 s. Although AOPT was slower by a factor of 3-4 times, we do no want to overemphasize the effect due to the small nominal values.

Table 4 gives the wall clock time observed for the Production cell and server examples.

n	AOPT	AFCFS	ASTACK	ACOMB
Production Cell				
48070	1.39	0.86	0.47	1.14
79900	2.54	1.47	0.83	2.06
113117	4.00	2.09	1.11	3.84
160029	5.75	2.89	1.59	5.53
Server				
9812	0.17	0.08	0.25	0.20
114653	0.78	0.63	1.00	1.02
583880	11.90	3.20	94.26	99.35

TABLE 4
Computation times in seconds

For the third scenario, where the state descriptor is heavy – the Production cell model has 231 variables – the computation times indicate that ASTACK is faster than AFCFS and both are faster than AOPT. The combined algorithm ACOMB does not pay off timewise.

Finally, for the Server model, we see large differences. For $n = 583880$, we observe that AOPT is about four times slower than AFCFS, however ASTACK dramatically falls back. Profiling ASTACK reveals that the on-the-fly reduction with iterators is particularly time consuming. Furthermore, many of the cycles that are removed are in fact nested and covered by cycles detected at a later point in time. An alternative version of ASTACK with a post reduction that generates a new reduced trace σ' after completely scanning σ takes only 1.89 seconds for $n = 583,880$ (9.98 seconds for $n = 1,170,380$), which also takes advantage of the fact that only few states need to be copied to generate σ'. We denote this to show the strong impact of implementation decisions that may distort conclusions.

Across all four scenarios, we recognize that all variants of the algorithms perform in the range of a few seconds for traces with a length in the order of 10^5. This is also consistent with results observed for other members of our set of examples.

Comparison of memory requirements. To compare algorithms memory wise, let us recall that AOPT uses four integer arrays of length n, a hash map that contains as many entries as there are different states in σ and a set with tuples to represent C'. The number of tuples in C' is limited to $0.5n$, so given that the tuples have length 2, the total space for C' is in the order of n. AFCFS mainly uses a hash map with states as entries and a representation of C' like AOPT. The key difference is that the hash map contains only elements of the on-the-fly reduced σ^c which can be much less than AOPT and due to the proximity of observed results for AOPT and AFCFS we can expect the number of elements in the AFCFS hash map to be approximately that of p_σ. ASTACK uses C' like the other algorithms and a stack with states as entries. The stack height depends on the longest subsequence of monotonously growing states in the trace and with respect to the selected order. The combined algorithm ACOMB matches with the space used by ASTACK for its first phase but then applies AOPT on a usually much smaller trace, so its overall space requirements have those of ASTACK and AOPT as an upper bound. We recognize that the number of elements in the hash map (stack) is the interesting characteristics of AOPT and AFCFS (ASTACK) computations that deserve an experimental evaluation. The pathological examples for scenario I with no cycles and II with many trivial cycles perform as expected, e.g., for scenario II all algorithms store the 10 different states that occur in the trace. For other examples, Table 5 shows that algorithms significantly differ. While values for AOPT grow with n, AFCFS is able to retain much lower values and those do not necessarily grow with n. ASTACK manages to retain values smaller than 100 for correct models, only models that contain an error like the Server and the Store models enforce larger values. The combined approach profits from the preprocessing with ASTACK only for certain configurations.

n	AOPT	AFCFS	ACOMB	
			ASTACK	AOPT
Courier				
5000	577	128	15	436
10000	763	91	18	482
Dining Philosopher				
67744	1225	233	26	604
Store				
6140	6095	6076	2026	6076
Faulty Proc				
4368	820	82	27	531
20961	1570	82	32	588
Conveyor				
5391	2099	246	70	1715
20160	3446	246	84	330
Database				
4732	23	4	10	14
19974	38	4	12	15
Scenario III: Production Cell				
10000	5206	680	61	4700
160029	34466	899	66	30097
Scenario IV: Server				
5473	60	27	17	52
583880	10942	741	162	9292
1170380	19899	1300	252	16766

TABLE 5
Number of elements in hashtable or stack

7 Working with Plots of p_σ

In this section, we discuss how plots of p_σ can shed light on what happens in a trace. From experiences with a number of examples, we recognize four distinct patterns: oscillations, a straight line, a step function, and convergence.

The case of continuing oscillations. Figure 10 shows an example of this case. We observe an initial linear increase followed by a phase of oscillations without a trend to grow. A linear regression function for that sequence of values shows a slope close to zero. We interpret this as follows: the simulator runs through an initial, transient warm up phase, then proceeds in a normal range of operation and occasionally returns to previously visited states. Note that same values of progress do not imply equality of states, but only equal distance from the initial state. Oscillations do not indicate an error, we may use them to also check for a position of a truncation point of the warm up phase that is usually taken into account for steady state performance evaluations. The case of oscillations is considered to be the behavior of a correct model; it gives an indication of the depth of the state space as it is explored. Given that we observe only a finite prefix of a potentially infinite behavior, the principal limit is that we do not gain any knowledge on the possible future behavior. For example, the considered trace may be too short to observe any erroneous behavior.

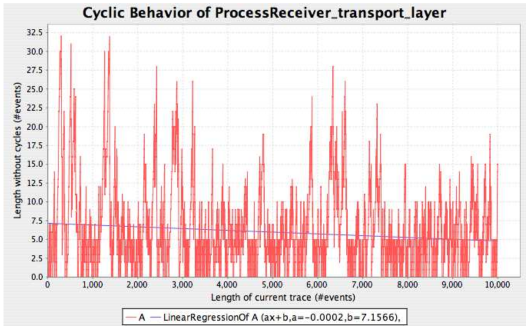


Fig. 10. Oscillations as seen for a submodel of the Courier protocol model

The case of a straight line. Figure 11 shows an example of this case. We see an almost linear trend to grow and the slope of a linear regression function is close to one. We see that the simulator runs and assigns values to state variables such that no cycles show up or cycles are not sufficient to give a substantial reduction. We observed this pattern for a variety of causes: 1) state

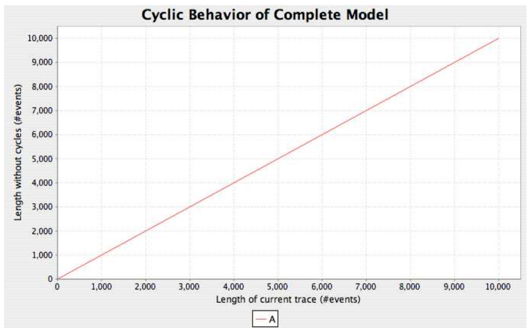


Fig. 11. Straight line as seen for the Store model and the Courier protocol model with an injected fault

variables that are used to count the number of occurrences; i.e., state variables are used as reward variables or act as some sort of logging mechanism to track behavior, 2) faulty irregular changes to state variables that are not taken back and that happen frequently, 3) overload situations in open systems, where the workload is too high for the available resources and state variables track growing queues of waiting jobs, 4) a partial deadlock for resource allocation in open systems where new jobs frequently arrive in a steady stream but the throughput breaks down and queues increase, and 5) the straight line that is always present in the initial phase of a plot of p_σ, so the chosen length of the observed trace requires appropriate consideration to avoid misunderstandings.

In situations 1-4, we can apply the same concept to individual state variables and subsets of state variables to track down which variables cause the overall behavior. This decomposition is possible because the overall case of oscillations results from a logical AND of all projections contributing oscillations. If one projection does not show repetitions, the overall trace cannot do so accordingly. For a straight line to be observed, at least one submodel or variable will show the same pattern.

The case of a step function. Figures 3 and 12 show examples of this case for traces of the Server model. We observe a step function with a phase of oscillations in each step. A linear regression function may show a marginal or significant slope which depends on the frequency of steps in the step function. We can interpret this as follows: the simulator runs in an operational phase with oscillations and occasionally performs singular changes to state variables that are never taken back to previous values in the observed simulation run and which makes a step to the next level in the plot. Causes that we observed for this behavior include 1) rare events that are logged by state variables, 2) rare events that cause erroneous behavior as in the server model, and 3) a lengthy initial warm up phase. This pattern suggests to check the event number or simulation time when a switch to next step occurs and check those events for correctness. One can truncate the trace after a step and apply the reduction operation to evaluate the reduced trace to check what causes the state changes. Decomposition can be applied as well. Note that a step function may look like a straight line if the frequency of steps is high enough with respect to $|\sigma|$.

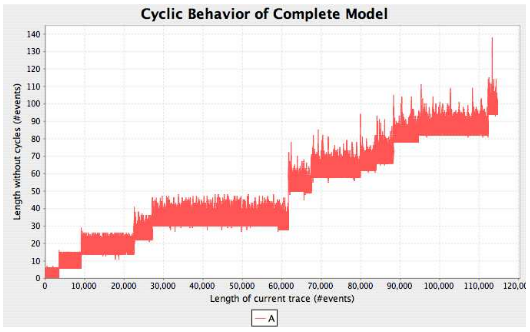


Fig. 12. Steps as seen for the Server model

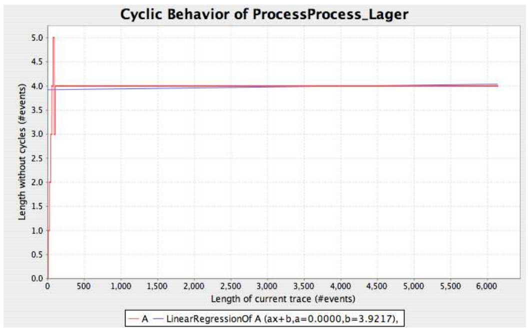


Fig. 13. Convergence as seen for the submodel Storage Area of the Store model

The case of convergence. This case describes a plot where p_σ converges to a constant function after an initial phase with a straight line, a step function or oscillations. We observed this case for models where the simulator performs events but the state description does not change, for example, if the reported state description abstracts from details in a model or if the considered state is a projection of the full state as in the decomposition approach described below. Obviously this case asks for a closer look why states do not change. Any state in the constant phase and events that lead to it are of interest here. One can truncate the trace where the constant phase begins and perform a trace reduction to investigate how that phase is reached. For example, the trace of the *Store* model shown in Fig. 13 shows this pattern if one projects on certain submodels like the Storage Area. The trace reveals a partial deadlock in an open system, where access to certain resources is blocked and new entities frequently enter the system and keep the simulation going.

Decomposition and projections. If the state descriptor contains several state variables, then we can consider projections on any subset of state variables for σ and decompose the observed behavior. We illustrate the approach with the help of the Store model where we consider only state variables of a submodel *Storage Area*.

If we take a trace σ and derive a new trace σ' by projecting each state s_i to only those values that belong to a particular submodel like the *Storage Area*, then we can compute $p_{\sigma'}$ for σ', which is shown in Fig. 13. Since not all events change the state of *Storage Area*, we observe stuttering steps, i.e., cycles of length 1, which makes $p_{\sigma'}$ constant between those events that do change the state. There are cases, where the dynamics of a submodel dies out and the plot becomes a constant function after an initial phase. If this is the case, then the plot for the overall model can still be any of the cases discussed above. For example, while the plot for p_σ of *Storage Area* becomes constant, the overall model shows as pattern as in Fig. 11. If at least one submodel gives a straight line for $p_{\sigma'}$ then the overall plot has to do so as well, which in turn suggests to follow a divide and conquer approach and check the plots of submodels if the trace of the overall model shows a straight line.

In summary, plots of p_σ provide us with a simple visual technique to recognize if a trace shows an irregular behavior and that gives guidance for where to search for its cause.

8 Conclusion

We propose a technique that identifies and removes cycles from a simulation trace. The separation of progressive from cyclic and repetitive fragments of a trace helps to identify errors in simulation models, in particular for dependability models that are composed of submodels that have a cyclic behavior. The proposed exact reduction algorithm has linear time and space complexity and achieves a maximal reduction for a given trace. Additional approximate algorithms are discussed that save on memory requirements. All techniques have been implemented in Traviando [11], a software tool for trace visualization and analysis, and have been evaluated on a range of example models.

Acknowledgments

We thank W.H. Sanders, T. Courtney and M. McQuinn for supporting us with an appropriate XML trace output of Möbius, W. Mao who pointed out that SRP relates to finding a longest path in directed acyclic graphs, R. Lamprecht for her editorial assistance and numerous reviewers for their advice.

References

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

[2] F. Bause, H. Beilner, M. Fischer, P. Kemper, and M. Völker. 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.

[3] R. E. Bellman. *Dynamic Programming*. Princeton, NJ, 1957.

[4] D. D. Deavours et al. The Möbius framework and its implementation. *IEEE TSE*, 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] M. Heiner and P. Deussen. Petri net based design and analysis of reactive systems. In Proc. *3rd Workshop on Discrete Event Systems (WoDES 96)*, pages 308–313, 1996.

[7] O.H. Ibarra, H. Wang, and Q. Zheng. Minimum cover and single source shortest path problems for weighted interval graphs and circular-arc graphs. In Proc. *Thirtieth Annual Allerton Conference on Communication, Control and Computing*, pages 575–584. University of Illinois, Urbana, 1992.

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

[9] P. Kemper. A trace-based visual inspection technique to detect errors in simulation models. In Proc. *Winter Simulation Conference*, ACM, 2007.

[10] P. Kemper and C. Tepper. Trace based analysis of process interaction models. In Proc. *Winter Simulation Conference*, ACM, pages 427–436, 2005.

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

[12] P. Kemper and C. Tepper. Automated analysis of simulation traces - separating progress from repetitive behavior. In Proc. *QEST*, pages 101–110. IEEE CS, 2007.

[13] D. Krahl. Debugging simulation models. In Proc. *Winter Simulation Conference*, ACM, pages 62–68, 2005.

[14] A. Law and W.D. Kelton. *Simulation Modeling and Analysis*. McGraw-Hill, 3rd edition, 2000.

[15] W. Mao. College of William and Mary, private communications.

[16] G. Nivasch. Cycle detection using a stack. *Inf. Process. Lett.*, 90(3):135–140, 2004.

[17] D. A. Sadowski. Tips for successful practice of simulation. In Proc. *Winter Simulation Conference*, ACM, pages 56–61, 2005.

[18] U. Sammapun, I. Lee, and O. Sokolsky. RT-MaC: Runtime monitoring and checking of quantitative and probabilistic properties. In Proc. *RTCSA*, IEEE CS, pages 147–153, 2005.

[19] R. G. Sargent. Verification and validation of simulation models In Proc. *Winter Simulation Conference*, ACM, pages 130–143, 2005.

[20] R. Sedgewick, T. G. Szymanski, and A. C. Yao. The complexity of finding cycles in periodic functions. *SIAM Journal on Computing*, 11(2):376–390, 1982.

[21] R. Sedgewick Algorithms in C++ Part 5: Graph Algorithms. Addison Wesley, 3rd edition, 2001.

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

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 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. He develops Traviando at the College of William and Mary.

Carsten Tepper has a Diploma degree in computer science (Dipl.-Inform., Universität Dortmund, Germany, 2000). In 2001-07, he conducted research in modeling and simulation with P. Buchholz at the Universität Dortmund, partially funded by SFB 559, Modeling Large Networks in Logistics. He is interested in the verification and validation of process-oriented systems and the performance analysis/tuning of database systems. He is a software engineer at ITGAIN Consulting, Hanover, Germany.