Automated Analysis of Simulation Traces - Separating Progress from Repetitive Behavior

Peter Kemper
Department of Computer Science
College of William and Mary
Williamsburg, VA 23187-8795, USA
kemper@cs.wm.edu

Carsten Tepper
LS Informatik IV
Universität Dortmund
44221 Dortmund, Germany
carsten.tepper@udo.edu

Abstract

Among the many stages of a simulation study, debugging a simulation model is the one that is hardly reported on but that may consume a considerable amount of time and effort. In this paper, we describe a novel technique that helps a modeler to gain insight in 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.

1 Introduction

Discrete event simulation is a common technique in the performance and dependability assessment of systems. Many professional and academic tools are available and applied in practice to make modeling and simulation productive. Simulation comes with few restrictions so the real crux in simulation modeling is not to obtain numbers as results but to achieve valid results. Sadowski [16] 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 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 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 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 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 [9], 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. 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 [17]. However, in tracing a simulation model, a modeler finds himself in the situation that it is unclear for what properties to ask for to be checked by a model checker or what hypothesis to test for. This is the situation we want to provide support for. In modeling for performance and dependability studies, we observed that models are often built from com-

ponents 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. So a cyclic behavior is an expected, "good" behavior, while certain errors may disturb that, e.g. by 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[1]. 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 errors, we consider our trace reduction approach a useful addition to the existing set of debugging techniques; see [10] for a detailed discussion how to make use of our technique. The approach (cycle reduction as well as cycle visualization) is implemented in Traviando [12], a trace analyzer that tracks performance figures, provides statistical evaluation of timed and untimed traces as well as model checking functionality.

[1]We discuss related work at the end of Section 2.

The rest of the paper is structured as follows. Section 2 gives basic definitions. 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 evaluates the overall approach. 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 "=". For example, if $s \in S \subseteq \mathbb{N}$ 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 stochastic well-formed nets (SWNs), and the multi-paradigm models of Möbius. 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.

Let us define some common operations for sequences. The length of $\sigma = s_0 e_1 s_1 \dots s_n$ is defined as $|\sigma| = n = \#events$. 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 $\sigma \circ \sigma' = s_0 e_1 s_1 \dots s_n e'_1 s'_1 \dots s'_m$. Obviously, if $\sigma'' = \sigma \circ \sigma'$ then $|\sigma''| = |\sigma| + |\sigma'|$. For $\sigma = s_0 e_1 s_1 \dots s_n$ and $0 \leq i < j \leq n$, we define a projection or substring operation as $sub(\sigma, i, j) = s_i e_{i+1} \dots s_j$. 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 $i < k < j$. Obviously, for any non-elementary cycle $[i, j]$ in σ there exists a sequence of $m > 1$ elementary cycles $[i_0, i_1], [i_1, i_2], \dots, [i_{m-1}, i_m]$ in σ with $i = i_0, j = i_m$ and $s_i = s_{i_0} = s_{i_1} = \dots = s_{i_m} = s_j$ that describes the same substring of σ, i.e., $sub(\sigma, i, j) = sub(\sigma, i, i_1) \circ sub(\sigma, i_1, i_2) \dots \circ sub(\sigma, i_{m-1}, j)$. Let $\mathcal{C}_{all} = \{[i, j] | 0 \leq i < j \leq n, s_i = s_j\}$ be the set of all cycles of σ, $\mathcal{C} = \{[i, j] | [i, j] \in \mathcal{C}_{all}, s_k \neq s_i, i < k < j\}$ be the set of all elementary cycles. Cycles allow us to define a reduction operation. For a $\sigma = s_0 e_1 s_1 \dots e_n s_n$ 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 $\mathcal{C}^*$ be a set of cycles that yields that reduction. We define this formally as follows.

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

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. We use the notation of an interval, since it matches what we see as index values for states in a cycle of σ. Note that $\mathcal{C}^*$ is not necessarily unique, since a non-elementary cycle $[i_1, i_2] \in \mathcal{C}^*$ could be replaced by a set of elementary cycles that form a sequence that also describes $[i_1, i_2]$. In light of this observation, we can reduce the set of cycles that we need to consider for SRP from $\mathcal{C}_{all}$ to $\mathcal{C}$ and still obtain a set $\mathcal{C}^*$ that gives the same maximal reduction of σ. 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 Section 6) and that we also observed $|\mathcal{C}_{all}| >> |\mathcal{C}|$ (we show this effect for the example discussed in Section 3) which guides us to develop algorithms that focus on $\mathcal{C}$.

Related work. SRP is related to the problem of cycle detection in periodic functions for which linear time algorithms with low memory requirements are known for long [15, 18]. The problem there is to analyze a function $f : D \to D$ on some domain D, for having 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 all cycles $\mathcal{C}_{all}$. These problems are different by nature. So classical algorithms like [15, 18] 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 [15] 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 and we do so for the two approximate algorithms we describe in Section 5 for a set of examples we evaluate in Section 6.

There is also work on interval graphs, e.g. [8], 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 $[l_{min}, u_{max}]$, [8]. So, an overlap of intervals is acceptable although not preferred (minimum sum of weights) in that context, while in SRP intervals need to be disjoint.

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. Failures happen mainly due to software failures and are handled by rebooting the system and restarting tasks that are performed. The workload can be partitioned into two types of tasks according to their service demands. The user population is limited, such that we decide to model the system by a closed 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 and class A customers are particular ones with different service requirements. 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 [6].

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 input and output buffer occupation at the server, those variables are shared via the *FullModel*. The *CompleteServer* submodel encapsulates how a service is performed and a submodel *failureAndRepair* that mod-

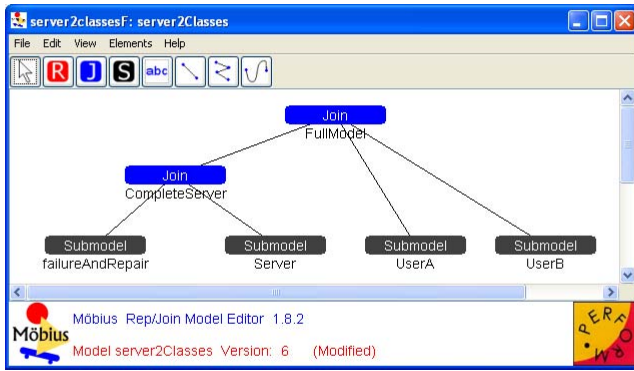

Figure 1. Composed Model

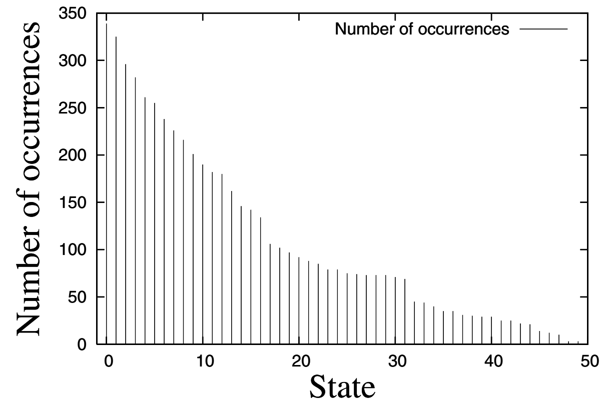

Figure 2. Number of occurrences of states

els 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 a boolean state variable "avail" 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 think-time. 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 down scale 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, here with $n = 5473$ events. The trace seems correct, but we somehow lack insight what to search for.

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 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 alone 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, 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, repetitive, and we are interested in the behavior of the composed model with respect to the timing of certain events.

Clearly, such a model can contain many types of errors, and cycle reduction can address only certain ones. Namely a modeling error that documents itself in state information and that does not have a complementary activity that resets or reverses the state change. Examples are actions that assign faulty values to a state variable or deadlocks in models of open systems where state variables count the number of customers/entities in the system. If this type of behavior is present in σ then it remains present in σ^*, the solution of SRP. Since (hopefully) $|\sigma^*| << |\sigma|$ it will be easier to track the cause of the error. Working with cycle reduction has two features that we consider useful. One helps us to recognize if an error of the above kind is present in σ, the other helps us to identify which events are used to reach a particular state of interest. For the first feature, let $sub(\sigma, 0, i)^*$ denote the solution of SRP for $sub(\sigma, 0, i)$ of σ. We can plot the length of $sub(\sigma, 0, i)^*$ for $0 \leq i \leq n$. Figure 3 shows this for our example trace. 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 and after that the model proceeds and returns to states in a cyclic manner as behavior. 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. By checking state variables of a state $s_i, i > 3514$, we see that $N(A)$ has increased by 1 which violates that the model has an invariant customer population for customers of class A. In addition to this, σ^* also gives a massive reduction, i.e. the algorithm that we will describe next obtains σ^* with $|\sigma^*| = 7$. With those few states and events it is immediate to recognize the violation of that invariance and that the reason is an unwanted side effect of the action that puts a customer of type B back into the queue if the

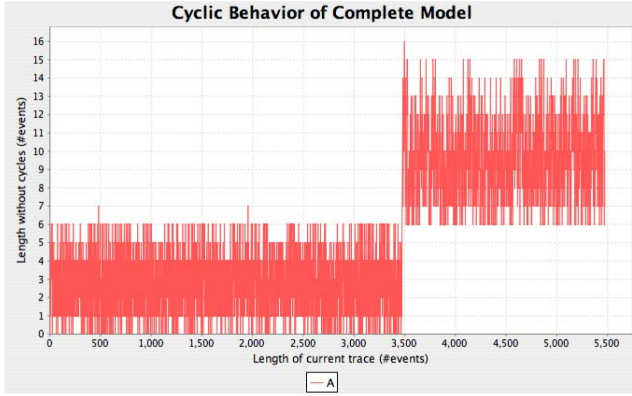

Figure 3. Length of trace after cycle reduction as a function of $|\sigma| = n$

server fails while serving a customer of class B. The error was induced when we extended the model from one customer class to two customer classes. 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

SRP seems to be one of those NP hard combinatorial optimization problems but this is not the case. 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 $C^* = \{[l_1, u_1], \dots, [l_m, u_m]\}$ be ordered such that $u_{c-1} < u_c$ for $1 < c \leq m$ (we will use c as an index for cycles and 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 $\{[l_1, u_1], \dots, [l_{c-1}, u_{c-1}]\}$ is an optimal solution for $s_0, \dots, s_{l_c}$ and $\{[l_{c+1}, u_{c+1}], \dots, [l_m, u_m]\}$ is an optimal solution for $s_{u_c}, \dots, s_n$. If we focus on the former, then the optimal solution on $s_0, \dots, s_{l_c}$ 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[2] for sequences $sub(\sigma, 0, i)$ for $0 \leq i \leq n$. So we can follow a dynamic programming approach here and use memoization. Note that for elementary cycles, for all $0 \leq i \leq n$ exists at most one $[l_c, u_c] \in \mathcal{C}$ with $u_c = i$. We consider all cycles in $\mathcal{C}$ in an ordered sequence of increasing values 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 $sub(\sigma, 0, u_c)$ by comparing the optimal solution for $sub(\sigma, 0, u_c - 1)$ with the reduction achieved by the optimal

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

```
AOPT(σ)
 0  p[0] = u[0] = l[0] = w[0] = c = 0; n = |σ|;
 1  h = empty hash map; C' = ∅; σ' = σ;
 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  while (0 < c)
19     if (w[c] ≠ w[c − 1])
20     then σ' = red(σ', l[c], u[c]); C' = C' ∪ {[l[c], u[c]]}
21     c = p[c];
22  return σ', C'
```

Figure 4. Algorithm AOPT

solution for $sub(\sigma, 0, l_c)$ plus the contribution of $[l_c, u_c]$. We memorize the better variant of the two as solution of $sub(\sigma, 0, u_c)$. The approach has its fundamentals in a Bellman equation [3]. Fig. 4 gives a detailed pseudocode description of the corresponding algorithm AOPT that solves SRP and computes $\sigma' = \sigma^*$ that remains when we remove all cycles in $\mathcal{C}' = \mathcal{C}^*$ from σ. It 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 $l[c] = l_c, u[c] = u_c$. Entry $w[c]$ gives the weight for the solution of SRP for $sub(\sigma, 0, u_c)$, i.e., the number of states that can be removed to reduce $s_0, \dots, s_{u_c}$. 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 $sub(\sigma, 0, i)$ 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.

Lemma 1. *Algorithm AOPT terminates.*

The proof for termination is straightforward.

Correctness. In order to prove the correctness of AOPT, we factor out several properties and prove them separately. We start with the observation that AOPT indeed removes cycles in line 20 of the algorithm.

Lemma 2. *Value pair* $l[c]$ *and* $u[c]$ *in line 20 of AOPT describes a cycle.*

Proof. If for all $0 < c \leq c_{max}$, $0 \leq l[c] < u[c] \leq n$ and $s_{l_c} = s_{u_c}$ then the statement is obviously true since only a subset of those values for c are considered in line 20. Note that u and l only obtain values in lines 6 and 7 of AOPT. Hashmap h stores tuples $(s_i, (i, k))$ with s_i being key, (i, k) being the value. The condition of line 3 ensures that there is a state s_j in subsequence $s_0, \ldots, s_{i-1}$ with $s_j = s_i$ and $(j, k) = getValue(h, s_i)$ when values are assigned in lines 6, 7. Since that value must have been stored in an aforegoing iteration (line 15 or 17) we have $l[c] = j < i = u[c]$ and $[l[c], u[c]] \in \mathcal{C}_{all}$. □

Lemma 3. *AOPT computes all elementary cycles in arrays* $l[], u[]$*, i.e.,* $\mathcal{C} = \{[l[c], u[c]] | 0 < c \leq c_{max}\}$

Proof. Based on Lemma 2 we know that AOPT generates $\{[l_c, u_c] | 0 < c \leq c_{max}\}$. We prove $\{[l[c], u[c]] | 0 < c \leq c_{max}\} \subseteq \mathcal{C}$ first. Assume the contrary and $[l_c, u_c] \in \mathcal{C}_{all} \backslash \mathcal{C}$ with $s_k = s_{l_c}$ and $l_c < k < u_c$. In iteration $i = k$ (line 2), AOPT has h containing $s_{l_c} = s_k$ (line 3), so $u[c] = k$ and $k = u_c$. Furthermore, line 15 updates the entry of h at $i = k$ with (k, c), such that a subsequent hit for s_{u_c} at $i = u_c$ would pull $l[c] = k$ in line 7 at $i = u_c$ which yields a different, second elementary interval inside $[l_c, u_c]$ so the assumed $[l_c, u_c]$ cannot be in $\{[l[c], u[c]] | 0 < c \leq c_{max}\}$. Secondly, we prove $\mathcal{C} \subseteq \{[l[c], u[c]] | 0 < c \leq c_{max}\}$. We assume the contrary and from all counterexamples, let $[l_c, u_c] \in \mathcal{C}$ be the cycle with smallest index c in $\mathcal{C}$ that is not an element of $\{[l[c], u[c]] | 0 < c \leq c_{max}\}$. Since $l_c < u_c$, the iteration in lines 2-17 reaches $i = l_c$ before $i = u_c$. At $i = l_c$, let c' denote the current value of variable c, then the entry with key s_{l_c} in h is either updated in line 15 by setValue($h, s_{l_c}, (l_c, c')$) or a new entry with key s_{l_c} with value (l_c, c') is added to h (line 17). Since $[l_c, u_c]$ is elementary, the next time in the iteration where we have $s_i = s_{l_c}$ is for $i = u_c$ and in that situation condition of line 3 is satisfied and $u[c] = u_c$, $(l[c], k) = (l_c, c')$ =getValue(h, s_{u_c}). So $[l_c, u_c] \in \{[l[c], u[c]] | 0 < c \leq c_{max}\}$. □

Next we recognize that elements of $\mathcal{C}'$ are all non-overlapping.

Lemma 4. *For any two intervals* $[l_x, u_x], [l_y, u_y] \in \mathcal{C}'$ *holds a)* $[l_x, u_x] \neq [l_y, u_y]$ *and b)* $u_x \leq u_y \implies u_x \leq l_y$*, i.e. the intervals of* $\mathcal{C}'$ *do not overlap.*

Proof. $[l_x, u_x] \neq [l_y, u_y]$ must hold since the assignment $u[c] = i$ in line 6 gives a unique value to $u[c]$ and since the loop in lines 18-21 follows a monotonously decreasing sequence of values for c ($p[c] < c$ for $0 < c$), no interval is considered twice. For the second property, we assume the contrary, i.e., $l_y < u_x \leq u_y$. Since $u_x < u_y$ (unique values stored in u), $x < y$ and $[l_y, u_y]$ is removed in line 20 before $[l_x, u_x]$. In that case, $c = p[y]$. Due to condition $w[c] \neq w[c-1]$ (line 19), the value of $p[y]$ cannot result from line 14, which in turn implies that $p[y] = k = max\{c' | 0 \leq c' < c, u[c'] \leq l[c]\}$ (line 11 and 7,15,17) which immediately implies that the next interval k must obey $u_k \leq l_y$, so $k \neq x$. Since $u_k \leq l_y$ and for any $x < k$ we have $u_x < u_k$, the assumed interval $[l_x, u_x]$ cannot exist. □

So we recognize that AOPT computes all elementary cycles and a set $\mathcal{C}' \subseteq \mathcal{C}$ of non-overlapping intervals such that the sequence of reduction operations in the loop in lines 19-21 is at least defined.

Lemma 5. *Set* $\mathcal{C}'$ *gives a maximal reduction.*

Proof. by induction over the length n of σ.

Theorem 1. *AOPT computes a solution* $\mathcal{C}' = \mathcal{C}^*$ *for SRP and* $\sigma' = \sigma^*$.

Proof. Follows from the fact AOPT only removes cycles (Lemma 2), that it considers all cycles in $\mathcal{C}$ (Lemma 3), that $\mathcal{C}$ is sufficient to consider for finding $\mathcal{C}^*$ (discussed in Section 2), that all removed cycles are non-overlapping (Lemma 4), so the reduction is valid, and it is maximal (Lemma 5). □

Lemma 6. *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* $O(1)$.

Proof. 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 constant removal costs for a single s_i. □

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.

5 Approximate Solution of SRP

Since AOPT's time complexity is linear and one cannot do less than reading 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

```
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           C' = C' ∪ {[j, i]};
6*          update(h, s_i, i)
7       else
8           add(h, s_i, i);
9   return C';
```

Figure 5. Algorithm to generate $\mathcal{C}' \subseteq \mathcal{C}$

that perform with at most same time complexity but less space.

It is fairly straightforward to come up with an algorithm that detects cycles and creates a set $\mathcal{C}' \subseteq \mathcal{C}_{all}$. Fig. 5 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 immediately obtain two approximate solutions of SRP.

The first solution is a greedy strategy that removes cycles by weight. Given $\mathcal{C}'$, it sorts elements $[i, j]$ of $\mathcal{C}'$ by weight $j - i$, iterates through $\mathcal{C}'$ in decreasing order and successively removes cycles $[i, j]$ from σ if possible, i.e. $\sigma = red(\sigma, i, j)$ if s_i and s_j 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), resp. identification, 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.

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. The obvious benefit is the immediate reduction of the space for σ and the entries in the hashtable.

A third approach is based on the work of Nivasch [15],

```
AFCFS(σ)
0   i = c = 0; σ_c = σ
1   h = empty hashmap;
2   while (i ≤ |σ_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          σ_c = red(σ_{c−1}, l_c, i);
11          C' = C' ∪ {[l_c, i]};
12          i = l_c;
12      else
13          add(h, s_i, i);
14  return σ_c, C';
```

Figure 6. Algorithm AFCFS

which focuses on the detection of cyclic functions (sequences). The main argument is that if a sequence becomes cyclic and repeats a loop, 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 $[s_m, s_{m+j-i}]$. Note that 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. 7 describes the algorithm in pseudocode and adapted to solve SRP. Nivasch's approach is particularly dedicated to identify cyclic functions and to determine cycle length, e.g. for random number generators, where it is very promising. 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 of states in an monotonously increasing order, function peek(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 [15]. Analogously to the adaptation of Nivasch' algorithm, one could adapt the algorithm by Sedgewick et al [18] which we do not follow here.

6 Evaluation of algorithms

In this section, we compare the 3 algorithms we considered in detail. AOPT, AFCFS, and ASTACK have all linear time but differ in the quality of results and in their space

```
ASTACK(σ)
0  q = empty stack;σ' = σ; C' = ∅;
1  for i=0 to n with stepsize 1
2      while (s_i <state(peek(q))
3          pop(q) ;
3      if (s_i == state(peek(q))
4      then // interval found
5          C' = C' ∪ {[index(peek(q)),i]}
5          σ' = red(σ',index(peek(q)),i);
6      else
7          push(q, (s_i, i));
8  return σ', C';
```

Figure 7. Nivasch' Algorithm

requirements, AOPT delivers an exact solution for SRP, the other two deliver valid reductions but not necessarily maximal ones and with less memory. In particular, ASTACK is extremely memory efficient. According to [15], the space complexity of ASTACK is logarithmic with high probability. We evaluate the performance of those algorithms on a number of traces taken from various example models, generated with various simulators, and in various lengths. Table 1 gives the resulting values for $max \sum_{[i_1,i_2]\in \mathcal{C}'}(i_2 - i_1)$ for the calculated set $\mathcal{C}'$ for algorithms AOPT, AFCFS, and ASTACK. For example, AOPT achieves a value of 54 in line 3, column 3 for model *Courier* with $|\sigma| = n = 100$ that is $|\sigma^*| = 100 - 54 = 46$. Since AOPT guarantees to achieve a maximal reduction for σ, i.e., it yields the maximal possible reduction values in the table. Column n describes $|\sigma|$ and column Model denotes which model has produced that trace. Models are sorted according to the generating environment.

Models entitled with APNN are generalized stochastic Petri net models, whose traces are generated with the simulator of the APNN toolbox [5]. Traces for rows *Courier* are generated from a stochastic Petri net model of the Courier protocol model by Li and Woodside [19]. The model is intended for performance analysis and generates a recurrent, finite Markov chain. The initial marking is chosen such that the model has a small state space and is expected to show a lot of cyclic behavior. Traces for rows *Prodcell* result from a large model of a production cell by Heiner et al [7]. The model contains hundreds of places and transitions and considers control of a production cell which consists of a feeding belt, a rotating table, a robot with two arms, a press, a second belt to remove processed parts and a crane.

Models entitled with ProC/B are simulation models based on a process interaction approach, the modeling formalism is supported by the ProC/B toolset [2]. Row *DinPhils* refers to a model of the classical dining philosophers. Row *Store* refers to model of a storage area described in [11]. It models the transfer of goods into a store and out

Model	n	AOPT	AFCFS	ASTACK
APNN				
Courier	100	54	54	0
Courier	1000	948	948	882
Courier	5000	4938	4938	4860
Courier	10000	9966	9966	9912
Prodcell	1000	810	648	486
Prodcell	10000	9720	9720	1944
ProC/B				
DinPhils	67744	67721	67721	66441
Store	6140	65	65	65
Möbius				
Server	5473	5466	5462	5252
FaultyProc	4368	4352	4345	2687
FaultyProc	10595	10586	10575	8460
FaultyProc	20961	20952	20952	16167
Conveyor	5391	5361	5334	1874
Conveyor	10578	10520	10428	10302
Conveyor	20160	20084	20048	19805
Database	4732	4732	4732	4698
Database	10170	10170	10170	10100
Database	19974	19974	19974	19926

Table 1. Reduction achieved for examples

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. The model describes an open system. The state representation that is chosen for the trace abstracts from certain details of the ProC/B 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).

Models entitled with Möbius are simulation models developed with the multi-paradigm, multi-solution framework Möbius [4]. Row *Server* refers to the Möbius model of a server with failure and repair that we discussed in Section 3. Rows *FaultyProc* is a failure model of a fault tolerant computer system that is part of a set of example models available with the Möbius distribution [4]. Similar to this, rows *Conveyor* refer to a model of a conveyor belt that is described in a process algebra supported by Möbius. Rows *Database* refer to a 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. Considering the reduction achieved by the different algorithms in Table 1, 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

		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 2. Approximation error

internal deadlock for resource allocation and the absence of cycles indicates this problem. For other models, AOPT achieves a substantial reduction, AFCFS often gets results that are equal or reasonably close, e.g. for the Möbius models. However, for *Prodcell* we can recognize that the length σ may have an influence as well, e.g., AFCFS does well for $n = 10,000$ but not at all for $n = 1,000$. It seems that differences get less for longer traces in the considered examples. So, we measured the approximation error of AFCFS and ASTACK for all examples. We compare solutions for any prefix of a trace to also see how much of the differences varies depending on the length of the trace. Note that the minimal difference is always 0 (at $sub(\sigma, 0, 0)$). Table 2 lists the maximum and average difference (rounded to two digits) we observe between $sub(\sigma, 0, i)^*$ for $i = 0, \ldots, n$ as computed by AOPT and approximate algorithms ASTACK and AFCFS. 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 *Prodcell*. The experimental results indicate that AFCFS performs consistently better than ASTACK for all examples wrt to the approximation error measured as max, mean, and standard deviation of the difference to the result of AOPT.

We measured computation times on a Pentium PC with 2 CPUs (3.4GHz), 2MB cache, 2GB main memory, running Linux. On all given examples (cf. table 1) our implementation usually needed less than 1 second and in the DinPhils example it slightly exceeded this bound. Only in the case of applying AOPT to the largest Database example our implementation needed 6.39 seconds.

Although the quality of ASTACK falls back, note that the maximal stack height and thus the space requirements are extremely small. Table 3, column *Stacksize* gives the maximal stack height for the examples. This motivates us to consider combinations of algorithms. The idea is to apply a space efficient heuristic first to reduce most of a trace and the exact algorithm to squeeze remaining cycles out of the resulting sequence. AFCFS+AOPT did not yield any improvement. Table 3 gives results for the combination of ASTACK+AOPT, namely column *AOPT(2)* gives the reduction achieved by AOPT when applied 2nd and column *Total* gives the total reduction obtained by the combination of both algorithms. Note that the combined method achieves optimal values for all experiments but for Courier 5000, DinPhils, Conveyor 10578 and 20160 where the achieved reductions are nevertheless near the optimum. Given that the height of the stack is less than 60 states for all experiments but for the Store examples (which has a deadlock in an open system where newly arriving customers are counted and thus state vectors can give a monotonously growing sequence), ASTACK is performing extremely well in terms of space which makes ASTACK favorable to reduce long traces where memory consumption is a concern with the option to improve the total reduction subsequently with AOPT if considered necessary.

Model	n	AOPT(2)	Total	Stacksize
APNN				
Courier	100	54	54	3
Courier	1000	66	948	9
Courier	5000	66	4926	5
Courier	10000	54	9966	7
Prodcell	1000	324	810	8
Prodcell	10000	7776	9720	13
ProC/B				
DinPhils	67744	1260	67701	12
Store	6140	0	65	1013
Möbius				
Server	5473	214	5466	13
FaultyProc	4368	1665	4352	19
FaultyProc	10595	2126	10586	19
FaultyProc	20961	4785	20952	21
Conveyor	5391	3487	5361	51
Conveyor	10578	153	10455	54
Conveyor	20160	1252	20057	54
Database	4732	34	4732	8
Database	10170	70	10170	8
Database	19974	48	19974	8

Table 3. Reduction by ASTACK+AOPT

7 Evaluation of overall approach

In general, the usefulness of the reduction operation is based on a key assumption: the state of a system s_i must be sufficient to define the subsequent behavior that is present in the trace and starting at s_i. 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 defines the potential future behav-

ior 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 process algebras, one would document the state as a textual description of the algebraic term for the state of each individual agent (process) and define "=" by some bisimulation. In the general case, i.e., discrete event simulation of non-Markovian models, the state of a simulator rather 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.

As indicated in Section 3, a plot of $sub(\sigma, 0, i)^*$ can be helpful to identify if an error of the kind that is preserved by the reduction operation is present in the trace. We do not claim that our technique is a silver bullet to all errors, however, we consider it an advantage that it can be applied for visual inspection without any further input by a modeler, i.e. no particular invariant, safety property has to be known, specified and evaluated for a first hand visual inspection of the cyclic behavior to decide which piece of a trace shall obtain further attention, see [10] for further details.

Trace reduction is also useful if a modeler wants to focus on how a particular state in a trace is reached and what events are necessary for that. This reduces the amount of events one needs to look at to identify the cause of an error. Trace reduction can also be used in combination with model checking of traces, to reduce subsequences in which a formula of interest is constantly true (resp. false).

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 [12], a software tool for trace visualization and analysis and evaluated on a range of example models. For further details on how to apply the approach see [10], here we demonstrate its usefulness by detecting a subtle modeling error in a dependability model of a server system with two classes.

Acknowledgments We thank W.H. Sanders, Tod Courtney and Michael Mc Quinn for supporting us with an appropriate XML trace output of Möbius, Weizhen Mao who pointed out that SRP relates to finding a longest path in directed acyclic graphs and numerous reviewers whose suggestions we tried to incorporate within the given space.

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] T. Courtney et al. The Möbius modeling environment: Recent developments. In Proc. *QEST*, IEEE CS, pages 328–329, 2004.

[7] 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.

[8] 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.

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

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

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

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

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

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

[17] 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.

[18] 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.

[19] 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.