Proceedings of the 2009 Winter Simulation Conference
M. D. Rossetti, R. R. Hill, B. Johansson, A. Dunkin, and R. G. Ingalls, eds.

RECOVERING MODEL INVARIANTS FROM SIMULATION TRACES WITH PETRI NET ANALYSIS TECHNIQUES

Peter Kemper

Department of Computer Science
College of William and Mary
Williamsburg, VA 23187, USA

ABSTRACT

Modern modeling frameworks allow us to generate and simulate discrete event system models of great complexity. The use of existing environments, and the use, calibration, and configuration of existing submodels to build large models in a productive manner can rise the question what really happens in a simulation run for a particular experiment. In this paper, we describe ways to make use of invariant analysis techniques that originate from the theory of Petri nets, to be applied in the broader setting of invariant identification from simulation traces. The key idea is to provide feedback to a modeler on constraints that hold for the observed behavior of a model in a simulation run.

1 INTRODUCTION

Stochastic discrete event systems (DES) models have been in use for many years to conduct a dependability or performance assessment of a system under study. Sophisticated and powerful modeling and analysis frameworks have been developed to make a model-based evaluation possible and productive. A discrete event simulation is frequently applied to derive values for dependability and performance measures based on a statistical evaluation of the observed dynamic behavior. This is a very low level technique compared to how models are specified in modeling formalisms supported in modeling tools. Most tools hide such details from a user and come up with results at a level that corresponds to the abstraction level used in modeling. This is appropriate if a given model is valid and its implementation is correct. However, often a more or less laborious learning process is needed to understand how the real system under study works and how the modeling framework is supposed to be used to describe and evaluate a simulation model. Necessary steps for the verification and validation of simulation models are frequently presented at the Winter Simulation Conference, see for instance (Sargent 2008) and (Balci 2004).

We propose an approach to obtain feedback for a modeler on the detailed dynamic behavior of a model as it is exercised by a discrete event simulator. The approach contributes to the verification, testing and debugging of a simulation model, where verification means to check that an executable simulation model correctly implements the conceptual model that a modeler has in mind. A simulator internally generates a sequence of states and events that most simulators are able to export into a trace file. This detailed documentation of a simulation run requires a modeler to analyze a large volume of data to verify correctness of an implementation or to debug a simulation model. Without appropriate tool support, this can be tedious, unproductive, or even infeasible for a modeler with insufficient expertise on how a discrete event simulation internally operates. However, a trace is a rich source of data and one can extract information from it about properties of the simulation model that has been executed. We use a simulation trace to conduct an analysis of invariants for a simulation model. We assume that a simulation model consists of state variables, whose values constitute the state of a model, and of actions, which occur as events when the model is simulated. Invariants deduced from a finite observation like an execution trace cannot be guaranteed to hold in general, so they are commonly referred to as *likely invariants* in the literature. We will omit the *likely* attribute for readability.

In this paper, we describe how invariants of two kinds can be checked or deduced from a simulation trace. The one kind, which we call state variable invariants, denotes that a weighted sum of state variable values remains constant throughout a given trace. Such invariants are useful to describe model properties such as each resource is in one out of several states at any given point of time, e.g., the sum of values for state variables *up* and *down* matches with the number of resources of a

particular kind and the invariant confirms that those resources are neither created nor destroyed throughout a simulation. The second kind of invariants describes how often certain combinations of actions need to occur to have the simulation return to the same state again. We name these invariants action invariants and they characterize possible cyclic or repetitive behavior of a simulation model.

The invariant analysis we propose is derived from invariant analysis traditionally known for Petri nets, more precisely place/transition nets, a modeling formalism rooted in concurrency theory and with a rich theory on structural properties of Petri nets, analysis techniques and applications. The key issue in this paper is to apply invariant analysis as it is known for place/transition nets to a much broader class of DES models and based on trace data obtained from DES models.

In software engineering, we find related work for the detection of *likely invariants*, which are detected at runtime by observing a program in execution. For instance, in Daikon (Ernst, Perkins, Guo, McCamant, Pacheco, Tschantz, and Xiao 2007), an inference engine is used to detect invariants. The key idea is to use a generate-and-check algorithm to test a set of potential invariants against execution traces of instrumented programs. Daikon reports those invariants that are tested to a sufficient degree without falsification. Those invariants can be used to obtain preconditions and postconditions over observed variables that are fulfilled by calls to observed methods. Similar concepts are found in the Agitator product by Agitar for test generation, in the DIDUCE tool for Java programs to monitor weakenings of invariants, and the IODINE tool to extract design properties from hardware designs. (Sahoo, Li, Ramachandran, Adve, Adve, and Zhou 2008) use likely invariants to recognize hardware errors. In software engineering, potential usages of likely invariants include program documentation, debugging, testing, generating input for program verifier and program repair. However, the invariants observed usually infer relations among variables like $x \leq y$, bounds $0 < i$ and references $p! = null$ which are of general use in programming. The invariant analysis we propose in this paper aims at a different characteristic: simulation models create traces that frequently repeat a relatively small set of actions and return to previously visited states (at least for steady state simulations).

The rest of the paper is structured as follows. In Section 2, we introduce notation to describe a simulation trace. In Section 3, we recall definitions and properties of place/transition nets and its invariant analysis. In Section 4, we describe how to apply invariant analysis to simulation traces. In Section 5, two example models show how to apply and take advantage of invariant analysis for the verification and debugging of simulation models. Our approach is fully implemented in a tool for the analysis and visualization of simulation trace. We briefly describe this tool in Section 6. We summarize in Section 7 and discuss benefits and limitations of the proposed approach.

2 TRACING SIMULATION RUNS

We consider a discrete event system that has a finite set $\mathscr{V}$ of m state variables and a finite set of actions $\mathscr{A}$. A state s is an m-dimensional vector with a value for each of the state variables $v_1, \ldots, v_m$ in $\mathscr{V}$. State variables are numerical variables with a domain in $\mathbf{R}$, $\mathbb{Z}$, or $\mathbf{N}$. The dynamic behavior is due to actions that occur and that cause a change in value for state variables. The occurrence of an action is denoted as an event. An event e is labeled with a corresponding action $a \in \mathscr{A}$ denoted as $a(e)$ and optionally also with a time stamp $t(e) \in \mathbf{R}$; it changes the state of a model from a predecessor or starting state usually denoted by s to a successor state s'. We are mainly interested in observing the behavior of DES from a sequence of state transformations, which we define as a trace. A trace is a sequence $\sigma = s_0 e_1 s_1 \ldots e_n s_n$ of states $s_0, \ldots, s_n \in S$ and events $e_1, \ldots, e_n \in E$ over some (finite or infinite) sets S and E for an arbitrary but fixed $n \in \mathbf{N}$. The length of $\sigma = s_0 e_1 s_1 \ldots s_n$ is defined as $|\sigma| = n = \#events$. Let $\sigma_{ij} = s_i e_{i+1} s_{i+1} \ldots s_j$ denote a subsequence or substring of length $j-i$ of a trace σ. A cycle is a substring σ_{lu} with $l < u$ and $s_l = s_u$. For those actions that are present in σ, we can observe circumstances (states) where they can occur (enabling conditions) and what changes an action performs to which state variable (firing effect). Note that the trace neither reveals any reasons why certain events occur nor why certain others do not. At any event, it is unclear if choices exist and why the one observed has been chosen. On the contrary, events that are present are obviously possible in the given situation. For instance, the overall dynamic behavior may be purely time-driven and the state information may just report on measurement data taken over time. However, our underlying assumption is that the current state information is at least partially responsible for enabling or disabling events and is of relevance for the future behavior of the simulated model.

In the following, we assume that sets $\mathscr{V}$ and $\mathscr{A}$ are known, and we consider a particular finite trace σ, generated by the simulator, to identify characteristic properties of the model. We do not assume any particular modeling formalism, or that the given information contains details why a particular event is or is not present at any position in the trace, or that it contains information on reasons or constraints on the value settings of variables. For notational simplicity, we adopt the common assumption that a simulation run starts at time 0, i.e. all time stamps in σ form a monotonously non-decreasing sequence of non-negative real values and that all variables are given an initial value in s_0. We assume that variables have

values with a numerical domain as floating point or integer such that basic algebraic operations as addition, subtraction, multiplication and division are defined.

3 PETRI NET INVARIANT ANALYSIS

In this section, we recall invariant analysis for place/transition nets (Silva, Teruel, and Colom 1998), a special class of Petri nets. Petri nets are directed bipartite graphs with two types of nodes called places and transitions which are connected by arcs. A place represents a state variable with range $\mathbf{N}_0$. Its graphical notation is a circle. Its current value is considered as the number of tokens that reside at that place. A transition represents a rule for changing those state variables to which it is connected. Its graphical notation is a rectangle. State changes are based on addition and subtraction of constant values. Formally, a place/transition net (PN) is defined as a 5 tuple $PN = (P,T,I^-,I^+,M_0)$, where $P = \{p_1,\ldots,p_m\}$ is a finite and non-empty set of places, $T = \{t_1,\ldots,t_k\}$ is a finite and non-empty set of transitions ($P \cap T = \emptyset$), I^-, $I^+ : P \times T \rightarrow \mathbf{N}_0$ are backward and forward incidence functions, and $M_0 : P \rightarrow \mathbf{N}_0$ is the initial marking. The initial marking M_0 is a special case of a marking $M : P \rightarrow \mathbf{N}_0$. A marking M denotes that state of a Petri net, it can be interpreted as an integer (column) vector which includes per place p one integer value describing the number of tokens on place p. Let $\bullet t := \{p \in P | I^-(p,t) > 0\}$ and $t\bullet := \{p \in P | I^+(p,t) > 0\}$ denote the input and output places of a transition t; analogously, $\bullet p := \{t \in T | I^+(p,t) > 0\}$ and $p\bullet := \{t \in T | I^-(p,t) > 0\}$ for place p. A Petri net describes a dynamic behavior, i.e., changes to the current marking that are performed by the firing of transitions. A transition $t \in T$ is enabled at marking M, denoted by $M[t>$, iff $M(p) \geq I^-(p,t)$, $\forall p \in P$. A transition $t \in T$ that is enabled at marking M may fire and yield a new marking M' where $M'(p) = M(p) - I^-(p,t) + I^+(p,t)$, $\forall p \in P$, denoted by $M[t > M'$. The incidence matrix I is defined as a $\mathbb{Z}_0^{(m \times k)}$ matrix with $\mid P \mid = m$, $\mid T \mid = k$ and $I(p,t) = I^+(p,t) - I^-(p,t)$.

The notion of invariants is well known in Petri net theory, so we only briefly recall definitions and properties. A vector $v \in \mathbb{Z}^m, v \neq 0$ is called P-invariant if $v^T \cdot I = 0$ with $m = \mid P \mid$ and I denotes the incidence matrix. A vector $w \in \mathbb{Z}^k, w \neq 0$ is called T-invariant if $I \cdot w = 0$ with $k = \mid T \mid$. We are mostly interested in semi-positive invariants, i.e., $v \in \mathbf{N}_0^n, w \in \mathbf{N}_0^m$, and those places (transitions) that have a positive value in v (w) give the support of an invariant. Invariants are additive - the sum of invariants is an invariant again - so if a net has at least one semi-positive invariant, then the set of invariants is infinite, which motivates consideration of a generating set of minimal invariants. Algorithms for the computation of that set are known, in particular a variant of the Fourier-Motzkin method described by Martinez and Silva in (Martinez and Silva 1982), see also Algorithm 8 in (Silva, Teruel, and Colom 1998). The algorithm has an exponential worst case time and space complexity, but in practice, it works extremely well and is efficient in most cases.

A semi-positive P-invariant v means that the weighted number of tokens on the support of v is constant for all reachable markings M of the net, formally, $v^T \cdot M = v^T \cdot M_0 = const$. If all places of a net are covered by a semi-positive P-invariant then the net is structurally bounded, i.e., the set of reachable states for any initial marking M_0 is finite and the marking of any place $p \in P$ has a finite upper bound (whose specific value depends on the choice of M_0). A semi-positive T-invariant w means that if a sequence of transitions is fired that contains each transition t as often as w_t, then the change of markings in total is zero, which means that the firing sequence is a loop that leads back to the marking where it started. A T-invariant considers only the effect but neither checks for the enabling of transitions nor gives information of a possible order of transitions in that sequence. Nevertheless, the good news is that invariants can be computed independently from an initial marking and independently from the size of the state space, even for unbounded nets with an infinite number of markings.

4 INVARIANT ANALYSIS FOR SIMULATION TRACES

If a place/transition net is simulated to generate a trace σ, then it is obvious how to recover all relevant information of $PN = (P,T,I^-,I^+,M_0)$ from σ if each transition occurs at least once in σ. However, place/transition nets are not a main stream formalism applied in modeling and simulation. In fact, Petri nets come with numerous extension and variations in an attempt to obtain a formalism that can be broadly applied in practice. The real question in our context is to investigate how Petri net invariant analysis can be applied to a broad class of DES simulation models and what insights can be gained from the results obtained.

In Petri net theory, one strong limitation of invariants is that invariants do not consider the enabling rule. From a different point of view, we can think of this as a degree of freedom: invariant analysis applies to any vector addition system with a state descriptor of numerical variables where there is a finite set of possible state changes due to an event e that change a state s to s' by $s' = s + \delta(e)$. If a given simulation trace σ contains states $s_i \in \mathbf{R}_0^m$ for a set of m real-valued state variables $\mathcal{V}$, then the state transformation by event e_i can be described as $s_i = s_{i-1} + \delta(i)$ with $\delta(i) = s_i - s_{i-1}$. We can extend this to a matrix notation with Δ_σ being a $(m \times n)$-dimensioned real-valued matrix with columns $\delta(1),\ldots,\delta(n)$. With this formation, $s_i = s_0 + \Delta_\sigma \cdot x$, where $x \in \mathbf{N}_o^n$ and $x(1) = \ldots = x(i) = 1$ and $x(i+1) = \ldots = x(n) = 0$. We can expect

Δ_σ to contain many identical columns, so we can define a more concise $(m \times k)$-dimensioned real-valued matrix Δ with k being the number of different columns in Δ_σ and all elements of the set of different columns in Δ_σ constitute the columns of Δ. Hence $s_i = s_0 + \Delta x$ where $x \in \mathbf{N}_0^k$ and $x(i)$ accounts for the number of times $\delta(j) = \Delta(.,i)$ in σ for $j = 1,\ldots,i$. This formation looses information on the order of events and also on the possibility of execution (enabling). For a given vector x, it is not clear if there exists a corresponding trace σ that can be produced for the model under consideration. An ordinary Petri net as the original simulation model (in case all transitions differ in their firing effect and σ is long enough so that each transition occurs at least once) will result in the special case of $\Delta = I$ given that Δ is appropriately permuted to match with the order of transitions for I.

There is an obvious similarity to $M' = M_0 + Ix$ for Petri nets and we purposely chose the cardinalities of sets $|\mathcal{V}| = |P| = m$ and $|T| = k$ being the number of columns in Δ to stress this similarity. The main argument for P-invariants and T-invariants carry over. Namely, a vector $v^T \in \mathbb{Z}_0^m$ with $v^T \cdot \Delta = 0$ has the property that $v^T \cdot s_i = v^T \cdot s_0 + v^T \cdot \Delta \cdot x = v^T \cdot s_0$ is constant for any given s_0, so we name such a vector v a state variable invariant. Analogously for a vector $w \in \mathbf{N}_0^m$ with $\Delta \cdot w = 0$, we see that $s_j = s_i + \Delta \cdot w = s_i$ and the corresponding combination of events in w describe a cyclic behavior of the model. Again, for any given w it is not clear if a corresponding trace σ_{ij} can be produced. However, for any given σ with a corresponding vector w, $\Delta w = 0$ implies that σ describes a cycle. Hence, we denote such a vector an action invariant.

We consider a part of an example model of a communication network with failures and repair to illustrate the concept. The model is discussed in more detail in Section 5.2., we focus on a model fragment the describes failures and repair of a single network node. Figure 4 shows this fragment as a stochastic activity network in the Möbius framework. A single node can be in each of the states *up*, *down*, or *inrepair*. Additional state variables like *requestrepairman* are used to deal with the allocation and release of a repair worker, a shared resource. The intended sequence of actions that this part of the model shall produce is *(failure, repairbegin, (interruptrepair,repairbegin)*,repairends)** where $*$ denotes an arbitrary number of occurrences including zero and for an initial state where the node is *up*. The graphical notation in Figure 4 matches with that of a Petri net extended with so-called input and output gates that describe a complex enabling condition (input gate) or firing effect (output gate) with C++ code fragments for a particular action. The graphical representation of those gates is a triangle. Action *repairends* is an example with an inputgate *IG1*, which specifies that a repair activity can only end if there is no request for a release (a nonzero value for variable *requestrelease*), and an outputgate *OG1*. An expected state variable invariant is $1 \cdot up + 1 \cdot down + 1 \cdot inrepair = 1$. The state transformations of all actions are of the kind such that whenever an action like *failure* occurs as event e_i, the corresponding vector $\delta(i)$ contains the same three nonzero entries, namely -1 for *up*, 1 for *down* and 1 for *requestrepairman*. The effect is that while Δ_σ contains n columns, matrix Δ contains only $|\mathcal{A}| = k$ columns and in particular one column for action *failure*. Both matrices contain one row for each variable in the model, so the fragment in Figure 4 contributes eight rows for variables and four columns for actions.

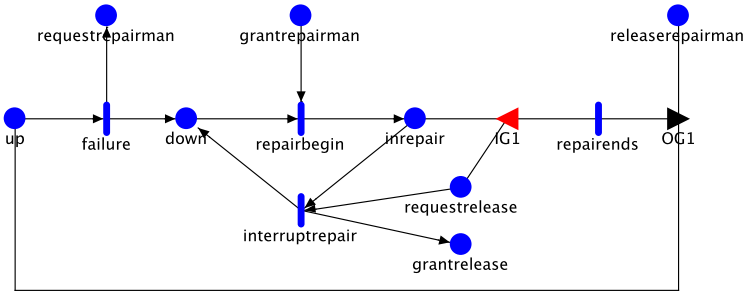

Figure 1: Faulty Failure-Repair Model, atomic model of a single node

4.1 State Variable Invariants

A matrix Δ which we obtain from a simulation trace can be used for at least two purposes. One is to verify if any given vector v establishes a state variable invariant. This is possible by checking $v^T \cdot \Delta = 0$ which requires a simple vector matrix multiplication. This can be useful to verify a set of safety properties, e.g., the number of resources in an operational or failed state is a particular constant value or the total number of tasks in a closed system model is a particular constant value.

The second purpose of Δ is in the calculation of invariants. If we restrict ourselves to integer-valued matrices Δ and we are interested in semi-positive invariants, then we can compute a generating set V of such invariants with the Fourier-Motzkin method in the same way as for an ordinary Petri net. As before, a modeler can investigate elements of V to see if it matches

with his or her expectations. In addition to that, one can recognize that the set of states the simulation model can reach is a subset of potential states $PS = \{s | v^T \cdot s = v^T \cdot s_0 \text{ forall } v \in V\}$ if Δ contains a column for each possible event $\delta(e)$ the model is able to produce. *PS* is an overapproximation that can include many states that are not reachable for a model. Nevertheless, we can use *PS* to make the case that any state $s \notin PS$ is not possible to reach, which may help argue towards the correctness of a model. In Petri net theory, the usefulness of this line of argumentation is often demonstrated with models of mutually exclusive access to a shared resource or a critical section. Finally, we can use an invariant v to argue towards boundedness of variables. However, this works only under certain restrictions. For simplicity assume that v covers all variables in $\mathscr{V}$. If v is semi-positive, i.e., $v \in \mathbf{N}_0^m, v \neq 0$ and variables in $\mathscr{V}$ have a domain $\mathbf{N}_0$, then there are only finitely many ways to distribute a total of $v^T \cdot s_0$ over variables $v \in \mathscr{V}$, which implies existence of an upper bound, since $\{s | v^T \cdot s = v^T \cdot s_0\}$ is finite. The last aspect may appear overly restrictive, however note that v need not cover all variables. If there is a subset of variables $\mathscr{V}' \subset \mathscr{V}$ that fulfills the above constraints and the support of v is a subset of $\mathscr{V}'$, then we can raise the argument on upper bounds for reachable values of all variables in the support of v. Hence we can pursue this type of analysis also in the presence of mixed models, where some variables are discrete while others are truly continuous.

So far, we focused on variables that are covered by some invariant v. If invariant analysis reveals that a variable is not covered by any invariant, but a modeler expects it to be covered, then this can be valuable information to verify state transformation functions in the model that modify that variable. Furthermore, if a non-trivial subset $\mathscr{V}' \subseteq \mathscr{V}$ is not covered, we can compute connected subgraphs G with a set of nodes $\mathscr{V}' \cup \mathscr{A}'$ where $\mathscr{A}' \subseteq \mathscr{A}$ denotes all actions that modify at least one variable $v \in \mathscr{V}$. Edges in G connect state variables and actions according to non-zero entries in Δ. Using a graphical format like that for Petri nets, and Petri net notation, the net has places $P = \mathscr{V}'$, transitions $T = (\bullet\mathscr{V}' \cup \mathscr{V}'\bullet)$ and edges according to incidence funtions obtained from Δ. This visualization may help a modeler to focus on actions that may be worth considering.

The absence of a state variable invariant does not necessarily indicate an error. Some models contain state variables that are used as counters to measure activities and that are only increasing in value over time. Other models encode different states of an automaton by a single state variable and state dependent actions, such that a variable invariant covering that state variable is unlikely to exist.

4.2 Action Invariants

As for state variable invariants, we can use the concept of invariants for three purposes. A modeler can check the existence of a particular invariant, compute a generating set of invariants, and identify a set of actions that are not covered by any action invariant. In the following, we elaborate these three cases a bit further.

We check if a given vector w is an action invariant by checking $\Delta w = 0$ with a simple matrix-vector multiplication. This is straightforward if there is a 1-1 mapping for columns of Δ with actions in σ. However, if there are several different vectors $\delta(i_1), \ldots, \delta(i_o)$ for events $e_{i_1}, \ldots, e_{i_l}$ that all result from the same action a, i.e., $a(e_{i_1}) = \ldots = a(e_{i_o})$, then it is more difficult for a modeler to carefully choose the appropriate number of occurrences for each of those cases of a that are expected to generate a cycle.

The action invariants contain values that describe how often an action needs to occur to see a sequence of events return to its starting state. Therefore it is natural to consider action invariants as vectors in $\mathbf{N}_0^k$ for a given matrix Δ. To compute a generating set of invariants, we can make use of the Fourier-Motzkin method used in Petri net invariant analysis. A modeler can then check those invariants if they match with expected behavior. In certain cases, it is necessary to exclude certain state variables from consideration in the invariant analysis. For example, this is the case if a state variable is incorporated in a model to count how often a certain action a occurs. This implies that a cannot be present in any action invariant, which usually effects other actions that modify state variables that a changes. This effect may ripple through the overall model and reduce the effectiveness of invariant analysis to reveal structural information. If we exclude variables like those counter variables from consideration, we set each row in Δ to null that corresponds to such a variable. As an effect, invariant analysis can then compute action invariants for a model where certain state variables have been removed and reveal characteristics that would normally not be revealed due to artifacts like those counters. This is useful to focus on parts of model that describe the control of the model's dynamic behavior.

Finally, if certain actions are not covered by an action invariant and if this is contrary to the modeler's expectations, then a modeler can check the state transformation functions associated with each of those actions. It is often helpful to draw submodels as a directed graph with nodes that result from all uncovered actions and the state variables that are modified by their state transformation functions. Edges can be added between state variables and actions that modify them with directions determined by the sign of the modification, e.g., a state variable that correspond to row i in Δ and an action that corresponds to column j in Δ has an edge directed towards the action if $\Delta(i,j) < 0$ and vice versa if $\Delta(i,j) > 0$. The resulting graph can

be joined with a similar one we obtain from uncovered state variables in the invariant analysis for state variables discussed before. These graphs illustrate which model elements may be jointly affected by a single mistake in a model.

5 EXAMPLE RESULTS

We illustrate the usage of invariant analysis for simulation traces with the help of two example models. The first model is a stochastic model for the dependability assessment of a satellite network developed by (Athanasopoulou, Thakker, and Sanders 2005). We provide some evidence with invariants that some intrinsic mechanism that is described by a complex case distinction specified with embedded C++ code works as expected. The second model is an availability model of a system with two classes of resources that fail and get repaired and a shared finite set of repairmen. The model is a toy example but resembles common aspects of dependability models that include failure and repair. It contains an error that is identified with insight obtained by invariant analysis and, for the improved model, we can show that invariants indicate that the simulation run is performed as expected.

Both models have been developed and simulated with Möbius (Deavours, Clark, Courtney, Daly, Derisavi, Doyle, Sanders, and Webster 2002) and the generated traces have been analyzed with Traviando. Traviando implements the proposed invariant analysis and also generates a set of webpages to document analysis results. The results for both models can be explored in full detail in the example sections of Traviando's webpage, see (Traviando 2009).

5.1 LEO Satellite Model

In this section, we discuss how to use invariants to argue towards the correct functioning of a dependability model of a LEO satellite system as described in (Athanasopoulou, Thakker, and Sanders 2005). The model describes communication among a network of satellites and a ground station where different components of satellites may be subject to failure. In addition to its concrete purpose for the design of a real satellite that has been built and launched (but was destroyed with its carrier rocket in an unsuccessful transport operation), the model itself is interesting to consider for verification purposes. Since satellites move on different orbits, communication between satellites and between a satellite and the ground station is possible only in certain positions. The continuous movement on each orbit is modeled with a discrete, finite set of positions which are held for a certain amount of time and which either enable a communication between two satellites or not. This mechanism implies an underlying deterministic sequence of events in a simulation run which solely represents the changes in positions of different satellites on their corresponding orbits. The mechanism is not trivial. It is implemented in a single action *Sat_counter* and with a lengthy sequence of nested if-then-else C++ statements that encode changes in positions. We consider a minimal network of 2 satellites whose atomic models consider failures of various components and a network model for communication.

For a verification of the model, one needs to check that the cyclic movements return to the same position after a particular number of steps. We can do this with the help of action invariants as discussed in Section 4. We need to run the simulation model for a configuration of interest to obtain a simulation trace σ. This trace must be long enough such that we can expect that all aspects of *Sat_counter* have been exercised, i.e., that all satellites have gone through their orbit positions. The particular trace we consider contains about 117,000 events, which is about 2.5 time units of the model, and *Sat_counter* accounts for more than 99% of all events. With this length, we are confident that σ is long enough. It is straightforward to check how often individual actions occur in σ. From σ, we obtain a set of state transformation vectors δ for all the different ways (cases) *Sat_counter* occurs, which yields 10 cases for this model.

We compute action invariants including all cases for state transformation vectors of all actions and with respect to all state variables. This results in several invariant vectors of which some exclusively consider cases of *Sat_counter*. We list those invariants in the columns of Table 1. From the totals line, we see that there are cycles of length 64 and 1472, where the latter seems a combination of the cycle of length 64 with one of length 23. A cycle length of 64 relates to the 64 different positions of a satellite with respect to the ground station represented by values of state variables *Sat1_Loc2GS* and *Sat2_Loc2GS*. By checking the δ vectors of individual cases, we can see that certain combinations are equivalent, e.g., $\delta(case1) + \delta(case4) = 2\delta(case3)$, which explains the relationship between inv_1 and inv_2. We do not present the state transformation functions of the 10 cases in further detail, since our main point is to illustrate that invariants can be helpful to check if the overall combination of state transformations can result in cycles of expected length.

In addition to the computation of action invariants, we can derive vectors of action invariants directly from σ by counting the number of occurrences of actions for cycles that are present in σ (Kemper and Tepper 2009). For this example, we observe action invariants that show cycles of length 128 for action *Sat_counter* which is 2 times the 64 step cycle for the positions of each of the 2 satellites with respect to the ground station. This is consistent with the expected behavior.

The computation of state variable invariants yields invariants that only cover state variables that describe the operational state of the satellites and the network. We do not obtain invariants that cover the state variables that capture the position of the satellites. This is consistent with the modeling approach of using explicit state-dependent transformation rules, such that a state variable like *Sat1_Loc2GS* goes through as sequence of values but those transformations do not necessarily establish a balanced flow such that there is a constant weighted sum of values of state variables including *Sat1_Loc2GS*.

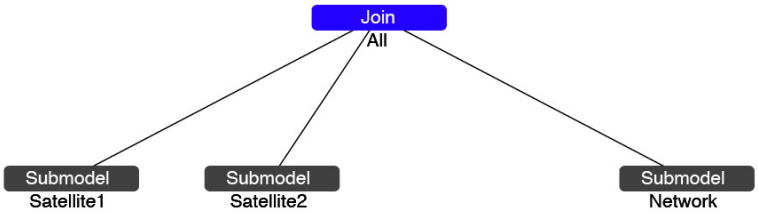

Figure 2: Satellite model, composed model view

Table 1: Action Invariants of the Satellite Model

case	inv_1	inv_2	inv_3	inv_4	inv_5	inv_6	inv_7	inv_8
1	0	15	0	345	0	345	0	15
2	1	1	23	23	23	23	1	1
3	30	0	690	0	690	0	30	0
4	0	15	0	345	0	0	0	15
5	8	8	184	184	184	184	8	8
6	16	16	368	368	368	368	16	16
7	8	8	64	64	120	120	0	0
8	0	0	105	105	56	56	7	7
9	1	1	23	23	23	23	1	1
10	0	0	15	15	8	8	1	1
total	64	64	1472	1472	1472	1472	64	64

5.2 Faulty Failure-Repair Model

We now consider a communication network with two categories of nodes, basic nodes and priority nodes. Priority nodes are essential to keep the communication network working while basic nodes are not. All nodes are subject to failure and share a crew of repair workers who repair failed units to keep the system operational. For obvious reasons, priority nodes have a higher priority for repairs than basic nodes. We develop a model to compute the utilization of the repair workers and the availability of the overall network for different numbers of staff. We consider a model where the priority policy to schedule repair workers is preemptive, i.e., if a priority node fails while all workers are busy and there is a worker who is repairing a failed basic node, then that repair job gets interrupted and the worker starts working on the failed priority node. The Möbius model uses two atomic models. A single atomic *node* model used for both types of nodes and *repairscheduler* model for the repair workers. The atomic models are both described in the stochastic activity network (SAN) formalism, much like a stochastic Petri net and without elaborated state transformation functions as in the previous case of the satellite model. The overall model is a composed rep/join model, where the *node* model is instantiated b times to describe b individual basic nodes that all behave in the same manner. Likewise, the *node* model (named *Vipnode* in the composed model) is also instantiated p times to obtain p priority nodes. All these instances are composed with a single instance of the *repairscheduler* model by the *Join1* node in the composed model shown in Figure 2.

When running a test run of the simulator with parameter settings $b = 5$, $p = 2$, and $r = 1$ repair worker, the measurements for utilization show a modest utilization of the worker and low availability of basic nodes. From the simulation trace σ of 116,815 events, we see that repair activity for priority nodes takes place throughout the trace but for basic nodes it seems to die out after a while. The model may have a partial deadlock.

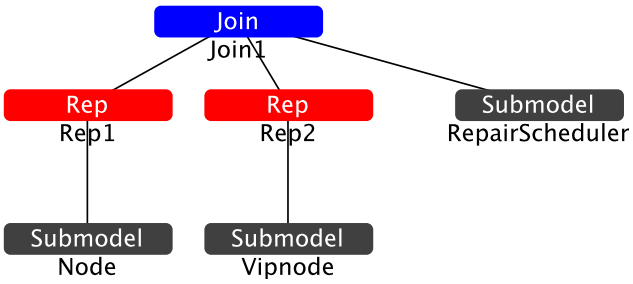

Figure 3: Faulty Failure-Repair Model, composed model view

There are different ways to recognize that there is a problem and to investigate into it. From the number of values assignments to state variables, we see that there are about 45,000 assignments to state variables that indicate that the repair worker is idle or serving a priority node. There are only a marginal amount of assignments to show that the repair worker serves a basic node. From a more detailed breakdown, we see that state variables of basic nodes are manipulated only up to event 1007 and basic nodes remain in the failed state till the end of the trace. From a plot of the different states visited by the simulator with σ, we see that after some initial short exploration of about 350 states, the simulator visits only a small set of about 20 states. From a breakdown of occurrences of various actions, we see that actions related to basic nodes occur only up to event 1007, and the last of those actions is action *rescheduleffinish*, which models the treatment of a preemption of service for a failed basic node due to a failure of a priority node.

From the state variable invariant analysis, we see that all state variables of a basic node are covered by an invariant that shows that any instance of a node, basic or priority, is at any point of time in one of three states *up*, *down*, or *in repair*, which is correct. There is also a state variable invariant that shows that the worker circulates through three states *idle*, *busywithnode*, or *busywithprioritynode*, which is correct. There are a few more invariants that show that the communication between models seems to operate fine. The only variable that is not covered by any invariant is *requestrepairman*, which is a variable that is shared among all replicated instances of basic nodes and the repair scheduler. This is not as expected; a request is issued when a node fails, then it is supposed to be granted by the scheduler and removed when the repair is done, so there should be an invariant covering this state variable. Priority nodes are instances of the same atomic *node* model as the basic nodes, with the same corresponding state variable to request a repair worker, but for priority nodes this variable is actually covered by several state variable invariants. This indicates that the atomic *node* model is not used properly for basic nodes.

From action invariant analysis, we see that actions that 1) describe the failure of a priority node, 2) request a reschedule from the scheduler, 3) model the finishing of rescheduling, and 4) model that the repairing of a basic node is interrupted, are not covered by any action invariant. This is also consistent with what is seen from the invariants that are deduced from cycles in σ. If we consider the uncovered actions together with state variables that are modified by the state transformation function of those actions plus the uncovered state variables and actions that modify them, we can recognize a connected submodel as shown in Figure 3. This submodel shows us a certain aspect of the model that is distributed over different atomic models and their composition. The Petri net is used only as a visualization aid with the understanding that edges directed towards a transition describe a reduction of values, edges directed away from a transition describe an increase in values.

With this information, we check the state transformations performed by actions that are not covered with action invariants and realize that action *interruptrepair* does correctly set the state of the basic node back to the failed state, but does not request a repair worker from the scheduler again. The implications of this error are that failed basic nodes whose service get interrupted will remain unserved until some other basic node fails and requests repair. Since replicated nodes share their shared variables and *requestrepairman* is a shared variable, this request may result in a repair service for any of the then two failed basic nodes. In this manner, repairing of basic nodes remains possible for the price of a prolonged downtime till *interruptrepair* occurred b times in total. After that, all basic nodes are in the failed state with no one left to generate a request for repair.

To fix this problem, we need to change the definition of *interruptrepair* to also generate a request for a repairman. In terms of the illustration in Figure 3, we need to add an edge from *interruptrepair* to *requestrepairman*. By doing so, we see in Figure 3 that we can expect that the existence of a state variable invariant that covers *grantrelease* will imply existence of

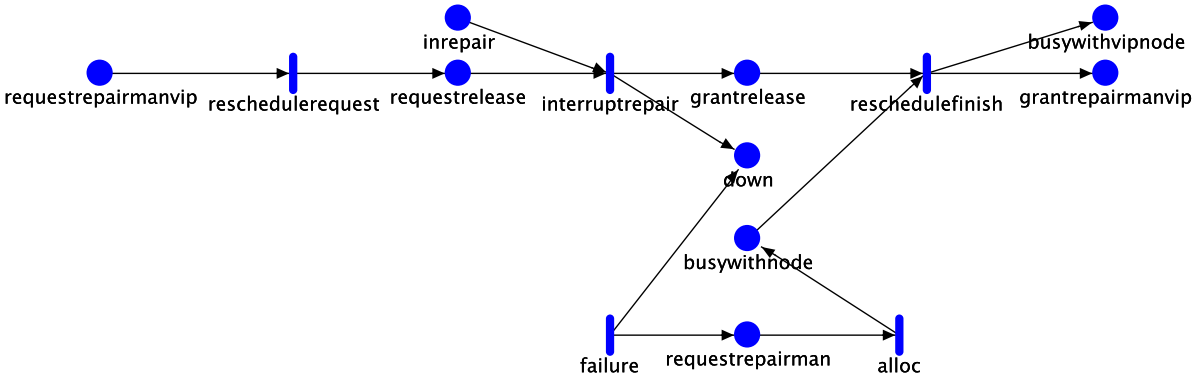

Figure 4: Uncovered Actions and State Variables

a similar one that covers *requestrepairman*. From running the improved model, we see that all state variables and actions are covered by invariants as expected and the partial deadlock disappears.

6 TOOL IMPLEMENTATION

The approach presented in this paper has been implemented in Traviando (Traviando 2009), a trace visualizer and analyzer for simulation traces of DES models. Traviando is not dedicated to a particular modeling formalism. Its input is an XML-formatted trace with a prefix that defines finite sets of state variables and actions and a suffix that contains data as we formally defined it for a trace σ.

Traviando has been extended over several years, its most recent version (Kemper 2009) provides a command-line interface to generate a report, an HTML formatted document, of various properties seen in a given trace σ. A selection of example reports can be accessed at (Traviando 2009). The idea is to provide a modeler with human readable feedback with graphics, data and explanations for a simulation trace in a format that requires no further expertise to explore what a simulator really does with a given model. In addition to the generated web report, Traviando comes with an interactive graphical user interface that visualizes any location in a trace σ in full detail with a variant of a message sequence chart and that provides various features intended to help a modeler locate areas of interest in a lengthy simulation trace. For example, Traviando provides an event browser that allows a modeler to investigate sequences of events (Schmidt and Kemper 2008) and a model checker for LTL formulas where areas in a trace that fulfill given formulas are highlighted with particular colors (Kemper and Tepper 2008). Traviando also provides a reduction mechanism to remove cycles in a trace which helps to generate plots for a measure of progress that show particular patterns to reveal certain errors in simulation models (Kemper and Tepper 2009).

Traviando has been used to analyze traces generated from various sources, most notably from Möbius, a modeling framework for the dependability and performance assessment of systems developed by W. H. Sanders at the University of Illinois at Urbana-Champaign.

7 DISCUSSION

The trace-based analysis of a simulation model faces a number of conceptual limitations. Any trace σ we consider is one single and finite example of a potentially infinite set of traces of potentially infinite length. The principal insight gained from an example is limited to demonstrate that a certain behavior - be it good or bad - is possible, e.g., states reached in σ are truly reachable. This puts the quality of results that can be achieved with trace-based analysis in the range of what can be achieved with testing. It can help us gain confidence in a model, verify certain properties, and debug and find causes of errors, but it cannot help to prove the absence of errors. In comparison to other verification techniques like a structured walk through of the code, trace-based analysis has the advantage of evaluating what the simulator does with a model and not what a modeler understands as the semantics of a model description.

For the invariant analysis we presented, the quality of invariants that are computed highly depends on the assumption that all possible state transformations are represented in Δ, which requires the presence of corresponding events in σ. What seems severe from a conceptual point of view need not be cumbersome in practice. A modeler should be familiar enough with a model to know which actions perform state transformation functions in a state-dependent manner (and in how many variations) and which actions do not in order to be able to judge if Δ is a complete representation or not. Furthermore, a simulation trace can be made sufficiently long by over-provisioning (traces in the example section had all more than 100k events for models with less than 100 actions) and the number of occurrences for individual actions is a straightforward statistical measure to check. If the tool implementation is able to digest lengthy simulation traces, simply using very long traces as an approximation of long enough traces is a viable solution. For the problematic case of rare event simulation, one can adjust the initial state and/or increase probabilities of events for the sole purpose of generating a complete matrix Δ. Furthermore, an automated analysis that exercises a long simulation trace is more thorough than a manual stepwise trace analysis that would consider state transformation functions of actions only at very few state settings.

The effort to obtain Δ is linear in the length of σ by a simple on-the-fly filtering process that inserts vectors $\delta(i)$ into a hash table and does not store any duplicates. There is no need to store the overall trace. The overall space required for Δ depends on $|\mathscr{V}|$, $|\mathscr{A}|$ and on the extent a single action may produce different vectors, e.g., a worst case example is a model with actions that have a state- or time-dependent state transformation function that create different vectors each time they occur such that $\Delta = \Delta_\sigma$. For a given matrix Δ, the effort for the Fourier-Motzkin method is exponential in m and k in the worst case but this method has been successfully applied in practice with Petri nets for broad classes of models. Alternatively to seeking an integer solution to $v^T\Delta = 0$, one can compute a real-valued solution with cubic effort. However, at least for action invariants an integer solution is easier to interpret.

The successful derivation of state variable invariants is subject to different modeling styles. To illustrate the point, we consider a small automaton that starts at a state 1 and sequentially visits states 2, 3, ..., n and terminates at state n. We consider two extreme ways to model it. We can think of an implementation with a single state variable s and domain $\{1,...,n\}$ and a single action that performs all necessary state transformations with a code statement that contains $n-1$ cases, one for each state transformation from state i to $i+1$. For this encoding, no state variable invariant will be computed. As an alternative implementation, we can use n binary state variables and $n-1$ actions, which switch the value settings for s_i to 0 and that for s_{i+1} to 1. For the latter implementation, we can expect to find a state variable invariant $v = (1,...,1)$ that states that in every state there will be exactly one state variable having a value of 1 while all others are set to 0. With appropriate tool support for model specification as well as model analysis, the latter style of modeling may become more advantageous and more frequently used than the former. In practice, we see a combination of both styles.

A strong point of invariant analysis is that both action and state variable invariants reveal structural information that connects multiple model elements. While we can check any single detail of a model description, more global, structural constraints are usually harder to verify. A model description itself usually reveals only very specific structural information. For example, the rep/join formalism for model composition in Möbius tells us which atomic models get instantiated with how many identical instances and which state variables are shared among submodels. However, state variable invariants may cover variables from various different submodels and reveal an overall global constraint. Similarly, an action invariant can reveal that a lengthy sequence of actions throughout a set of submodels would return to its starting state. For our second example, such a sequence may involve a failure of a basic node, the start of a repair operation, the failure of a priority node, the rescheduling of the repair worker, the repair of the priority node, or the start and finish of the repair operation at the basic node. Obviously those sequences are naturally present and a modeler may want to confirm the way they operate by checking a corresponding action invariant.

In practice, the ability to exclude certain state variables or certain actions for an invariant analysis is a useful one to extend the applicability of the approach. For instance, certain state variables that are used as counters to memorize how often an action occurs only increase in value. Those variables are easily identified by have a corresponding row $\Delta(i,.) \geq 0$ of non-negative values. To exclude such a variable from consideration, one only needs to set $\Delta(i,.) = 0$. A similar pattern with negative entries is present in dependability models with failures but no repair. As for counters, excluding those variables from analysis allows a modeler to check more properties with invariant analysis.

The invariant analysis we presented applies broadly to DES model with a static state description (a fixed size vector of numerical state variables). It is not limited to a specific Petri net formalism.

The presence as well as the absence of invariants can be used in debugging a simulation model. Model elements that are not covered by invariants can be graphically displayed and connected subgraphs can guide a modeler in his or her search for reasons for that behavior.

8 CONCLUSION

We present an approach that performs an invariant analysis of discrete event simulation models based on a simulation trace. The class of simulation models that can be analyzed needs to have a finite set of numerical state variables that constitute the state of a model. Its dynamic behavior is documented in a trace of a simulation run that contains states and events. We conduct an invariant analysis that has its roots in classical invariant analysis known for place/transition nets (a particular class of Petri nets). We demonstrate the usefulness of the approach with two example models, one is a dependability model of a satellite network and taken from the literature, the other is a small model of a network with failures and repairs that contains a non-trivial error, which we identify with the help of invariant analysis. Both models have been developed in Möbius. The corresponding trace analysis is performed with Traviando, a trace analysis tool that implements the approach presented in this paper.

ACKNOWLEDGMENTS

We thank the four reviewers and Ruth Lamprecht for their help to improve this paper, Sam Klock for evaluating Daikon and the Möbius group at UIUC for their support with models and trace generation.

REFERENCES

Athanasopoulou, E., P. Thakker, and W. H. Sanders. 2005. Evaluating the dependability of a LEO satellite network for scientific applications. In *Proceedings of the 2nd Int. Conf. on Quantitative Evaluation of Systems*, 95–104: IEEE Computer Society.

Balci, O. 2004. Quality assessment, verification, and validation of modeling and simulation applications. In *Proceedings of the 2004 Winter Simulation Conference*, eds. R. G. Ingalls, M. D. Rossetti, J. S. Smith, and B. A. Peters, 122–129. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

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

Ernst, M. D., J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.* 69 (1-3): 35–45.

Kemper, P. 2009. Report generation for simulation traces with Traviando. In *Proceedings of the 2009 IEEE/IFIP Int. Conf. on Dependable Systems and Networks*, 347–352: IEEE Computer Society.

Kemper, P., and C. Tepper. 2008. Traviando - a trace analyzer to debug simulation models. *Simulation News Europe* 18/3-4.

Kemper, P., and C. Tepper. 2009. Automated trace analysis of discrete event system models. *IEEE Transactions on Software Engineering* 35,2:195–208.

Martinez, J., and M. Silva. 1982. A simple and fast algorithm to obtain all invariants of a generalized Petri net. In *Application and Theory of Petri nets*, Number 52 in Informatik Fachberichte. Berlin: Springer.

Sahoo, S., M.-L. Li, P. Ramachandran, S. Adve, V. Adve, and Y. Zhou. 2008. Using likely program invariants to detect hardware errors. In *Proceedings of the 2008 IEEE/IFIP Int. Conf. on Dependable Systems and Networks*: IEEE CS.

Sargent, R. G. 2008. Verification and validation of simulation models. In *Proceedings of the 2008 Winter Simulation Conference*, eds. S. J. Mason, R. R. Hill, L. Mönch, O. Rose, T. Jefferson, J. W. Fowler, 157–169. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Schmidt, N. J., and P. Kemper. 2008. Phrase based browsing for simulation traces of network protocols. In *Proceedings of the 2008 Winter Simulation Conference*, eds. S. J. Mason, R. R. Hill, L. Mönch, O. Rose, T. Jefferson, J. W. Fowler, 2811–2819. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Silva, M., E. Teruel, and J. Colom. 1998. Linear algebraic and linear programming techniques for the analysis of place/transition net systems. In *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets*, Number 1491 in LNCS. Berlin: Springer.

Traviando. 2009. <http://www.cs.wm.edu/~kemper/traviando.html>, <www.cs.wm.edu/~kemper/traviando/examples.html>, accessed June 1, 2009.

AUTHOR BIOGRAPHY

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