An Automated Technique to Support the Verification and Validation of Simulation Models

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

Abstract

Simulation modeling requires model validation and verification to ensure that computed results are worth being considered. While we cannot expect a magic solution to the general problem, automated techniques for particular aspects of validation and verification are feasible. In this paper, we propose a technique to deduce model properties automatically from simulation runs performed for verification and validation and to use those properties for runtime monitoring during production runs. Properties are represented as formulas in linear temporal logic and are limited to functional properties. We demonstrate the applicability of the approach with using an extended version of a stochastic Botnet model originally developed by Van Ruitenbeek and Sanders.

1. Introduction

Modern modeling frameworks help us to create and analyze detailed models of great complexity. In stochastic modeling for the dependability and performance assessment of systems, substantial research has explored the development of versatile formalisms, ways to develop accurate system models using these formalisms, and methods to efficiently analyze such models. In practice, model analysis usually boils down to a series of simulation experiments. In a simulation study, verification and validation (V&V) are necessary steps to justify confidence in the results of the study. Validation relates to the question "Did we build the right system?" while verification addresses the transformation of a conceptual model into an implemented, executable model with the question "Did we build the system right?" See for instance [1, 22]. A valid and verified model is expected to provide reasonably accurate values for measures of interest that are consistent with what would be observed in the system under study.

Verification directly assesses the correctness of an implementation, where "implementation" can mean many things – for example, creating a model in a graphical modeling environment, crafting and combining models from different simulation environments for an executable overall model, or implementing a simulation model in a common programming language with or without the support of a simulation library. Many graphical user interfaces in simulation modeling enable the inclusion of code snippets, illustrating the close proximity of modeling to programming. In summary, implementing a conceptual model comes with many challenges that are seen in software engineering in general.

Related to V&V, testing is broadly applied in software engineering with substantial support for automated testing and regression testing. One challenge is to provide the correct and expected results for non-trivial tests. A recent approach to this problem is the automatic generation of tests that characterize and document the behavior of a program in an executable, testable form as, for example, Agitar's Agitator software does. For V&V of simulation models, we will draw from this idea but instead consider the production of a formal specification in a modal logic. Such a formal specification can in turn be evaluated on-the-fly at runtime. The corresponding technique is known as *runtime verification* [17]. It is applied in software engineering, for example, in the work by Sokolsky [16] or Rosu [19].

Linear temporal logic (LTL) can form the basis of formal specifications for verification, including runtime verification. In [11], Dinesh et al. extend LTL to describe regulatory texts for business operations and offer a trace-checking algorithm to evaluate conformance of systems to regulations by trace analysis. Bauer et al. discuss how to conduct LTL model checking on traces in [5]. We will follow the latter approach to achieve runtime monitoring of a distributed set of simulation runs. Performing monitoring and runtime verification in a distributed environment has been discussed in [23], where Zhou et al. describe an extension to their mon-

itoring and checking framework MaC to evaluate network protocols that are specified with declarative networking. In our case, the simulation experiments are independent and communication among clients is not considered.

Given these approaches to performing runtime verification, the chief remaining challenge is obtaining a formal specification that accurately describes the system's correct behavior. In [18], Mondragon et al. describe how to clarify natural language specifications for program behavior and generate formulas in future interval logic using the Prospec tool in such a way that those formulas can then be compiled down to safety properties monitored with the MaC framework for running Java programs. In this paper, we investigate ways to derive a formal specification in an automated manner on the basis of the routine V&V work an analyst typically performs.

In software development, generating a formal description of program behavior is supported to a certain extent by tools that generate so-called *likely invariants*. The term *likely* refers to the fact that the invariant properties are inferred statistically from a finite sample of observations, so their correctness is accordingly not guaranteed. A well-established tool in this category is Daikon [12], which derives such invariants as safety properties that hold at method calls and returns. For readability, we drop the word "likely" from our discussion and use the term "invariant" with the understanding that all invariants deduced from finite trace information are only likely invariants. Very simple invariants, namely ranges of values, are used in [21] to monitor hardware and to detect permanent hardware faults.

In this paper, we propose a way to enhance the usual process of conducting a simulation study of the performance or dependability of a system in the following way. V&V requires an analyst to simulate a model for configurations that a) exercise all aspects of the model (according to a simple coverage criterion stipulating that relevant pieces of code that are not executed by tests cannot be considered sufficiently tested) and that b) enable validation of a model (i.e., show that the model is able to reproduce known performance or dependability results). If a simulation model succeeds on those V&V experiments, we can make use of those V&V experiments to infer correct behavior of a model and derive a formal specification of LTL formulas.

The class of properties we address in this paper focuses on purely functional properties of a stochastic model. We assume that timing requirements are covered sufficiently by the performance/dependability measures an analyst defines for a study. The formal specification is derived in an automated manner, along with a report in a less formal, easier to read format; we will not discuss details of the latter and refer instead to [14]. An analyst can review both of these outputs to recognize incorrect descriptions of parts of the system and accordingly add, change, or remove formulas from the specification. The set of formulas can be used later on to check further simulation experiments, either to regression-test a modified model or to monitor production runs. The main point is that an analyst can learn easily and promptly if the model is simulated in a way that leaves its range of verified behavior. Since we focus on untimed behavior, the set of formulas leaves sufficient room for a performance or dependability study to be meaningful.

The contribution of this paper is to pave the way for automated support of V&V testing by means of a formal specification in LTL that can be generated and monitored in a completely automated manner. We evaluate the proposed approach using an implementation that extends and integrates Möbius[10], Traviando [15], and Daikon [12]. In this paper, we derive various types of invariants, including several types using an adaptation of Daikon for the analysis of simulation traces. We formalize all invariants as formulas using a state/event linear temporal logic (SE-LTL) and transform them into finite automata in a three-valued variant of LTL for subsequent runtime verification.

The rest of the paper is structured as follows. In Section 2, we introduce necessary notation for simulation traces and SE-LTL. In Section 3, we describe different types of invariants, which we derive via Daikon and Traviando for given simulation traces. Section 4 describes our tool architecture, and Section 5 shares an application of our approach to an extended version of a Botnet model developed originally by Van Ruitenbeek and Sanders [20].

2. Definitions

We consider a discrete-event system that has a finite set $\mathcal{V}$ of m state variables and a finite set of actions $\mathcal{A}$. A state s is an m-dimensional vector with a value for each of the state variables $v_1, \ldots, v_m$ in $\mathcal{V}$. State variables are numerical variables with a domain in $\mathbb{R}$, $\mathbb{Z}$, or $\mathbb{N}$. The occurrence of actions that cause a change in value for state variables yields dynamic behavior. The occurrence of an action is denoted as an event. An event e is labeled with a corresponding action $a \in \mathcal{A}$ denoted as $a(e)$ and optionally also with a time stamp $t(e) \in \mathbb{R}$; it changes the state of a model from a predecessor or starting state (denoted s) to a successor state (s').

We are mainly interested in observing the behavior of a stochastic model 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 $n \in \mathbb{N}$, the length of σ. For those actions that are present in σ, we can observe circumstances (states) where they can occur (enabling conditions) and what changes an action makes to which state variables (firing effect or state transformation).

Note that the trace neither reveals any reasons why cer-

tain events occur nor why certain others do not. In any event, it is unclear if choices exist and why the one observed has been chosen. 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 $\mathcal{V}$ and $\mathcal{A}$ are known, and a finite trace σ is given such that all variables are assigned an initial value in s_0. For model-checking purposes, traces are extended with a state-labeling that indicates which elementary propositions hold in a state. We consider a set of state-based atomic propositions AP_s and a set of event-based propositions AP_e. A state-based atomic proposition is an (in)equality over functions with state variables in $\mathcal{V}$ as their parameters. A concrete example of a set of atomic propositions is comprised of (in)equalities relating arithmetic expressions that are constructed from state variables, constants (e.g., reals), and arithmetic operations (such as $\{+, -, \cdot, \div\}$). A state-based atomic proposition evaluates to true or false for any given state s_i, $0 \leq i \leq n$ in a trace σ. Let $L(\sigma_s(i)) \subseteq AP_s$ denote the subset of atomic propositions that are true at state s_i.

An event-based atomic proposition is an (in)equality that evaluates on a triple (s, e, s') where s denotes the starting state of an event e that yields s' as its successor state. A corresponding set of atomic propositions AP_e comprises functions over event labels in $\mathcal{A}$ and state variables in $\mathcal{V}$, where $'$ denotes that a variable refers to the successor state. Event labels are used to recognize if a considered event corresponds to an action of interest. For example, we can formulate that an action inc increments a state variable with the following implication $(a(e) = inc) \Rightarrow (v' = v + 1)$ where the left-hand side is an atomic proposition that is true if the considered event has a label inc and the right-hand side is an atomic proposition that is true if the value of v in the successor state s' is an increment of the value of v in the predecessor state. For readability, we simply use v instead of $v(s)$ and v' for $v(s')$ in formulas. If an event-based atomic proposition holds for an event e_i then a corresponding label is most naturally associated with that event; however, for notational simplicity it is associated with s_i in the logic below. Let $L(\sigma_e(i)) \subseteq AP_e$ denote the subset of atomic propositions that are true for (s_i, e_{i+1}, s_{i+1}) and $L(\sigma_e(n)) = \emptyset$ by definition.

Linear Temporal Logic. *State/event linear temporal logic* (SE-LTL) [6, 9] comprises a set of operators that can be used to describe what is expected to occur along a state-event sequence. SE-LTL expressions take the form of propositions comprised of the usual Boolean operators ($\vee, \wedge, \Rightarrow, \Leftrightarrow, \neg$), along with a set of temporal operators (**G**, **F**, **U**, **R**, **X**) and a set of atomic propositions. SE-LTL propositions evaluate to $\top$ (true) or $\bot$ (false).

Definition 1 *(SE-LTL). Let $AP = AP_e \cup AP_s$ be a set of atomic propositions with two disjoint subsets. The syntax of SE-LTL formulae is defined inductively as:*
$\phi ::= p | \neg\phi | \phi_1 \wedge \phi_2 | \phi_1 U \phi_2 | X\phi$
where p is an element of AP.

Atomic propositions AP_s are associated with states as is typical for LTL. For SE-LTL, we follow [6, 9] and associate action labels with the starting state of a corresponding event (with the marginal adjustment that action labels are taken from a set AP_e). We recall the semantics of LTL for *infinite* traces. Let σ^* denote an infinite trace with finite prefix σ and let $\sigma^*_k = s_k, e_{k+1}, s_{k+1} \ldots$ denote the suffix of σ^* that starts at state s_k. So, for a given trace σ^* and SE-LTL propositions ϕ and ψ, we can define the semantics of SE-LTL formulas evaluated at σ^*_k in the following way:

$$
\begin{array}{lcll}
\sigma^*_k & \models & p & \iff p \in L(\sigma^*_s(k)) \vee p \in L(\sigma^*_e(k)) \\
\sigma^*_k & \models & \neg\phi & \iff \sigma^*_k \not\models \phi \\
\sigma^*_k & \models & \phi \wedge \psi & \iff \sigma^*_k \models \phi \text{ and } \sigma^*_k \models \psi \\
\sigma^*_k & \models & \phi \mathbf{U} \psi & \iff \exists l \geq k : \sigma^*_l \models \psi \\
 & & & \quad \text{and } \forall i, k \leq i < l : \sigma^*_i \models \phi \\
\sigma^*_k & \models & \mathbf{X}\phi & \iff \sigma^*_{k+1} \models \phi
\end{array}
$$

Note that event-based atomic propositions are essentially associated with the starting state of the event to which they belong. With these basic operators, we can derive others to obtain the full set of common Boolean operators ($\vee$, $\wedge$, $\Rightarrow$, $\Leftrightarrow$, $\neg$), along with a set of temporal operators (**G**, **F**, **U**, **R**, **X**). Globally **G** ϕ specifies that ϕ is true at the current and all subsequent states. Finally **F** ϕ specifies that ϕ is eventually true; that is, that ϕ is true at the current or some subsequent state. Until ϕ **U** ψ specifies that ϕ is true *at least* until ψ is true: ϕ is true at this state and every subsequent state until ψ is true, at which point ϕ may be false. Release ϕ **R** ψ specifies that ψ is true at the current and all subsequent states until ϕ is true, at which point ψ is false. Next **X** ϕ specifies that ϕ is true in the state immediately following the current state. Formally:

$$
\begin{array}{lcll}
\sigma^*_k & \models & \mathbf{G}\phi & \iff \sigma^*_k \not\models \top \mathbf{U} \neg\phi \\
\sigma^*_k & \models & \mathbf{F}\phi & \iff \sigma^*_k \models \top \mathbf{U} \phi \\
\sigma^*_k & \models & \phi \mathbf{R} \psi & \iff \sigma^*_k \not\models (\neg\phi \mathbf{U} \psi))
\end{array}
$$

Of course, a verification of ϕ at runtime operates on a finite prefix σ of any σ^* such that this definition requires further adjustments. We briefly recall concepts of runtime verification from [17]. Runtime verification is a lightweight verification technique that focuses on checking whether a run of a system satisfies or violates a given correctness property. From a conceptual point of view, runtime verification answers the word problem (i.e., whether a given trace is an element of the language of traces that a correct model would generate). This technique is implemented using the con-

cept of a monitor. A monitor is a device that reads a finite trace and yields a verdict. A verdict is a truth value from some truth domain, which can be simply Boolean or enriched with further values such as *indeterminate*. A monitor corresponds to a particular correctness property and should adhere to two maxims, namely those of impartiality and anticipation. Impartiality requires the monitor to refuse a definite answer (true or false) if there are possible continuations of the given finite trace that may lead to contrary verdicts. Anticipation requires the monitor to give a definite answer for the given finite trace if all possible continuations would yield the same verdict. Since LTL is a well-accepted logic used for specifying properties with respect to *infinite* traces, it is natural to seek an adjustment for the case of runtime verification where only *finite* traces are considered. There are different ways to do so: Bauer et al. derive a three-valued LTL in [3], a four-valued LTL in [4], and an extension towards time in [3, 5]. As a first step, we follow [3].

Three-valued LTL. *Three-valued LTL* (LTL_3) is an adaptation of LTL to permit an on-the-fly evaluation on state-event sequences of finite length. It uses the same syntax definition of LTL, resp. SE-LTL in our case, and we just need to adjust the semantics. A direct implication of the impartiality maxim is that there is a need for a third truth-value, namely *indeterminate* (denoted **?**). For a finite sequence σ of length n, let σ^* denote an infinite trace with prefix σ and let ϕ be an SE-LTL proposition. We describe the semantics of LTL_3 in the following way:

$$\phi(\sigma) = \begin{cases} \top & \text{if and only if } \nexists \sigma^*, \sigma^* \not\models \phi \\ \bot & \text{if and only if } \nexists \sigma^*, \sigma^* \models \phi \\ ? & \text{otherwise} \end{cases}$$

In other words, ϕ evaluates to true if there is no way for ϕ to become false in states following s_n. Similarly, ϕ evaluates to false if there is no way for ϕ to become true in states following s_n. ϕ evaluates to indeterminate if neither of the two above cases apply.

For a given LTL_3 formula, we need to obtain a monitor that we can then execute to evaluate a simulation run. In [3, 5], Bauer et al. describe how to transform an LTL_3 formula into a deterministic finite state machine with three output symbols. We follow this approach to obtain a representation of a formula ϕ that allows us to evaluate a trace by reading its content in a single pass, in a forward manner, and without a need for storing all input data σ. The monitor classifies any prefix trace as either as good ($\top$), bad ($\bot$), or neither good nor bad (?). The monitor fulfills the anticipation maxim and gives a definite verdict as early as possible, i.e., based on the shortest informative prefix of σ.

For runtime verification, mainly safety properties are monitorable in the sense that a monitor is able to obtain a definite evaluation $\top$ or $\bot$. Following [5], the set of formulas that are monitorable is in fact larger than just safety properties, but many interesting properties are not monitorable, including the send/receive property we discuss in Section 3. Bauer et al. propose a four-valued logic that refines ? further to obtain more specific results; however, we will focus on the simpler logic LTL_3, which suffices for the set of formulas we can generate in an automated manner.

3. Derivation of Properties

This section discusses different types of properties and how to derive them in an automated manner. We assume that we are given a set of traces Σ as input, and we expect Σ to correspond to verification runs that cover pathological cases and exercise all actions of a model. The properties we identify are intended to help us distinguish faulty runs from those that are performed correctly. Σ provides us with an incomplete set of runs that the modeler considers correct. In general, we have far too little information to draw the line between the superset of correct runs that includes Σ and all other possible traces that are considered faulty. What we can do is identify certain properties that are based on the fact that performance and dependability models are not arbitrary programs but have certain common characteristics. We classify these properties into the following types.

Type 1: Finite Set of Values. We first consider a single state variable v for which Σ indicates a distribution of values with a finite range $l_v \leq v_v \leq u_v$, with lower bound l_v, upper bound u_v, and data on sequences of values of v as projections from elements of Σ. If the number of different values is reasonably small and fragmented, an explicit enumeration ($v \in \{c_1, \ldots, c_k\}$) can be useful. However, an explicit enumeration, like other statistical characteristics of the distribution of values like mean, mode, and median, seems to be very sensitive to changes in a model configuration. Furthermore, the length of the formula grows with the number of different values. Therefore, we expect bounds to be a coarser but more applicable representation. In [21], bounds on variables are the only property considered for runtime monitoring of hardware faults.

Type 2: Monotonicity. From the sequences of values, we can distinguish variables that fluctuate in value versus variables that are monotonously non-decreasing/non-increasing. The latter are used in models to count the number of times a particular event has happened – for example, a variable that is used to measure an impulse reward or the number of operational components in a transient simulation study for the mean time to failure. Monotonicity can be expressed as $v \bowtie v'$ with $\bowtie \in \{<, \leq, >, \geq\}$. If monotonicity is not an artifact of a too short trace σ, then we can expect this to hold throughout a series of experiments.

Type 3: Constant Weighted Sums. Considering sets of variables, a natural question concerns which subsets belong together. In many dependability models, we find a finite automaton with c states being encoded with c state variables, such that the sum of their corresponding values is always 1, since the automaton has to reside in one of its states, (i.e., to describe the different operational stages of a component). In performance models of closed systems, we have a finite number of customers switch positions between a finite set of c locations being encoded with c state variables, such that the sum of their values is always the number of customers. The particular number of customers may vary in a series of experiments but is constant in each individual experiment. In [13], we describe a method to calculate state variable invariants from a set of state transformation vectors $\Delta(\sigma) = \{s_{i+1} - s_i | 0 \leq i < n\}$. The resulting set of vectors $W \in \mathbb{R}^m$ has the property that $\sum_{i=1}^{m} w_i \cdot v_i = const$ for $w \in W, s \in \{s_0, \ldots, s_n\}$. Since the constant value depends on s_0 or, respectively, the model configuration, a formulation like the formula $\sum_{i=1}^{m} w_i \cdot v_i = \sum_{i=1}^{m} w_i \cdot v'_i$ is advisable. W is derived in a manner that obtains a minimal generating set of vectors, which implies that no relevant invariant is missed.

Type 4, 5: Pre- and Postconditions of an Action. Considering events (s, e, s') and individual actions, it is interesting to determine what the preconditions $P \subset AP_s$ are that hold for any state s to enable an action $a \in \mathcal{A}$, what the postconditions $Q \subset AP_s$ are that hold in s', and what transformation functions Δ action a performs to transform s into s'. We formulate a precondition as particular ranges of values $\bigwedge_{v \in \mathcal{V}} (l_v \leq v \leq u_v)$ or as relations among variables, e.g. as $v_1 \leq v_2$. In order to make a precondition hold only for a particular action a, the overall formula is $\mathbf{G}((a(e) = a) \implies P)$ where $P \subseteq AP_s$ is an atomic proposition of the starting state s. Analogously, we can define postconditions by considering v' instead of v. Postconditions hold for variable settings after the occurrence of a particular action.

Type 6: State Transformation Function of an Action. An action a that occurs as event e_i, i.e., $a(e_i) = a$, performs a state transformation $\delta(e_i) = s_{i+1} - s_i$. The state transformations a particular action a performs in Σ create a set of vectors $\Delta(a) = \bigcup_{\sigma \in \Sigma} \{\delta(e_i) | a(e_i) = a, 0 \leq i < n\}$. We can build formulas that describe $\Delta(a)$ in at least two ways: $\mathbf{G}((a(e) = a) \implies D)$ where $D = \bigvee_{d \in \Delta(a)} \bigwedge_{v \in \mathcal{V}} (v' - v = d(v))$ and simply enumerate all entries of $\Delta(a)$, or we can aggregate the possible ranges of values we observe for $v' - v$ by setting $D = \bigwedge_{v \in \mathcal{V}} (l_v \leq v' - v \leq u_v)$ where $l_v = min\{d(v), d \in \Delta(a)\}$ and $u_v = max\{d(v), d \in \Delta(a)\}$. The latter formula has a length that is bounded by $\mathcal{V}$, while the length of the former depends on $\mathcal{V}$ and $\Delta(a)$. We can expect that each action manipulates only very few variables and leaves most variables unchanged. A formal integration of an inertia rule that requires that variables remain unchanged if changes are not explicitly mentioned substantially shortens the length of most formulas in practice. Formulas for state transformation functions correspond to a code coverage criterion for testing program code: the given set of tests Σ must cover each action and the derived set of formulas covers all actions for subsequent monitoring (testing) simulation runs.

Type 7: Successor Events to an Action An action a that occurs will enable a particular set of possible actions $succ(a)$ to occur next. From Σ, we can derive $succ(a) = \bigcup_{\sigma \in \Sigma} \{a(e_{i+1}) | a(e_i) = a\}$. Since simulation models often contain concurrency, the actions that are observed following action a in $succ(a)$ need not be causally dependent on a at all. Nevertheless, a model will imply that certain combinations may not legally occur. So the following formula aims at characterizing legal successor events for any given action a as $\mathbf{G}((a(e) = a) \implies \mathbf{X}B)$ where $B = \bigvee_{b \in succ(a)} (a(e) = b)$. Alternatively one can evaluate the complementary set $\mathcal{A} \backslash succ(a)$ with $B = \bigwedge_{b \in \mathcal{A} \backslash succ(a)} (a(e) \neq b)$. For an implementation, one may select the representation that is shorter in length. If these formulas are generated for all actions in $\mathcal{A}$, then they check all possible bigrams (words of two literals) in the language generated by the simulation runs of a given model. This type of formula relates to a coverage criterion on branch coverage in software testing in that the formulas consider all possible successors to an action that may create a choice of several successors.

Type	Formula	where
1	$\mathbf{G}(v \in \{c_1, \ldots, c_k\})$	$\{c_1, \ldots, c_k\} \subset \mathbb{R}$
1	$\mathbf{G}(l_v \leq v \leq u_v)$	$l_v, u_v \in \mathbb{R}$
2	$\mathbf{G}(v \bowtie v')$	$\bowtie \in \{<, \leq, >, \geq\}$
3	$\mathbf{G}(\sum_{i=1}^{m} w_i \cdot v_i = \sum_{i=1}^{m} w_i \cdot v'_i)$	$w \in \mathbb{R}^m$
4	$\mathbf{G}((a(e) = a) \implies P)$	P precondition
5	$\mathbf{G}((a(e) = a) \implies Q)$	Q postcondition
6	$\mathbf{G}((a(e) = a) \implies D)$	D change
7	$\mathbf{G}((a(e) = a) \implies \mathbf{X}B)$	B successors

Table 1. Types of generated formulas.

More complex formulas. Table 1 summarizes the formulas considered so far, which are mainly safety properties that can be evaluated at any state. Of course, SE-LTL allows us to define much more complex formulas with the help of the $\mathbf{U}$ operator and other derived operators. The $\mathbf{U}$ operator allows us to recognize particular sequences of

actions or states. For instance, a formulation that a message that is sent is finally received can be formulated as $\mathbf{G}((a(e) = a) \implies \mathbf{X}(\phi \mathbf{U} \psi))$ where a is the send operation, ϕ describes the intransmission part, and ψ describes the receiving action. In existing models, we often find that individual messages are not distinguished, which implies that any receiving action would make the formula valid even though messages do not correspond. In case of concurrent messages, a single receive action would make all formulas valid that are initiated by a sequence of send actions before the receive action. As such, formulas of this kind are difficult to construct from scratch. In the example model we consider in Section 5, we will add a separate submodel that models an individual object such that these formulas can be created. However, we believe that these kinds of formulas are too model-specific for an automated derivation; we instead enable modelers to manually specify SE-LTL formulas.

Automated Derivation of Formulas of Type 1-7. We briefly discuss automated techniques and tools for the derivation of formulas of type 1-7. In particular, we discuss approaches implemented by Daikon and Traviando.

Daikon [12] is a tool designed to produce likely invariants from running programs. These invariants are intended to comprise a specification of the program's behavior, which can then be compared against expected properties or implemented as assertions that are checkable at runtime. While Daikon is primarily intended for use in conventional computer programs (with native support supplied for C/C++, Java, and other languages), it has been applied elsewhere; see, for example, [2] (using Daikon-generated invariants to detect rootkits).

We developed a transformer to translate simulation traces into formatted files of the sort produced by instrumented programs for Daikon. Since Daikon searches for invariants centered around program points (i.e., objects, method entries, method exits, etc.), we had to adapt the entities in simulation traces into types that Daikon understands. In the simulation trace format we use, variables are partitioned into *processes* corresponding to components of the model. We treat these processes as objects and actions as methods. This view provides a suitable means of enabling Daikon to search for invariants at the process level (i.e., invariants that apply to a component of the model) and for actions (pre- and post-conditions; properties of state transformations). Daikon-generated invariants can be used to produce formulas of types 1-2 and 4-6, but it does not provide support for formulas of type 3 (it does occasionally output sums of variables, but only occasionally and in an insufficient way) and formulas of type 7.

Algorithmically, Daikon starts with a very large set of candidate formulas that is reduced while scanning through its input data. Candidates are removed if they turn out to be false. The final set of invariants is sanitized of formulas that other valid formulas imply or for which there is insufficient statistical evidence to justify claiming that they hold.

In Traviando [15], we implemented an automated formula generator that explicitly targets types 1-7. Recognizing the corresponding properties from trace files is algorithmically straightforward for all types but for type 3. The derivation of a generating set of weight vectors w such that $\sum_{i=1}^{m} w_i \cdot v_i = const$ is performed by computing solutions for $w\mathbf{M} = 0$ where $\mathbf{M}$ is an integer or real-valued matrix of columns taken from sets $\Delta(a)$ for all $a \in \mathcal{A}$. So $\mathbf{M}$ has dimensions $\mathcal{V} \times k$ where $k = |\bigcup_{a \in \mathcal{A}} \Delta(a)|$. The corresponding approach has been published in [13], and example results demonstrate that it is helpful for recovering common invariants on states of a system component (in dependability models) and finite sets of customers (in performance models of closed systems).

4. Tool Support

We implemented our approach by extending and combining three different tools, namely Möbius [10] for modeling and simulation, Traviando [15] for the generation of formulas and for runtime monitoring, and Daikon [12] for the generation of formulas. The approach consists of three separate steps: 1) the generation of Σ with Möbius, 2) the offline derivation of formulas from Σ with Daikon and Traviando, and 3) the online, on-the-fly runtime verification of formulas for a running Möbius simulation by Traviando.

Modeling and Simulation. Möbius is a modeling framework for the dependability and performance assessment of systems developed by W. H. Sanders et al. at the University of Illinois at Urbana-Champaign. It supports multi-paradigm modeling where an overall model is composed of submodels formulated in one of several formalisms, most notably Stochastic Activity Networks (SANs). The Möbius simulator supports the generation of trace files for simulation runs that include information on state variables and actions as well as detailed sequential data on which action takes place, when, and what changes it makes to state variables. The simulation engine is able to distribute a series of individual simulation experiments over a network, which is tremendously helpful for conducting an industrial-strength simulation study in practice. We extended the Möbius simulator to communicate trace data with our runtime verifier software via sockets.

Generation of Formulas. We generate formulas from a set of simulation traces obtained from Möbius simulation runs. We implemented a transformer that translates a trace file into the expected input format for Daikon. Daikon gen-

erates formulas that are then reformatted to work with Traviando, a trace analyzer that we use for runtime monitoring. In addition to Daikon, we extended Traviando to derive SE-LTL formulas of types 1-7 on its own. The types of formulas generated that way are focused on functional properties of simulation models as discussed in Section 3. In addition to the automated generation of formulas, Traviando supports manual specification of formulas, which we use to obtain more specific and more complicated formulas for a given model. All these formulas can then be evaluated at runtime.

Runtime Monitoring. Figure 1 illustrates how a distributed set of simulation experiments conducted by Möbius is monitored by Traviando and a distributed set of Traviando client evaluators that digest trace data generated by Möbius simulators. We extended Traviando with a module that transforms SE-LTL formulas into LTL_3 formulas and corresponding finite automata, the monitors. On each host that runs a Möbius simulation experiment, we launch a Traviando runtime verifier client that communicates via sockets with Möbius and evaluates a Möbius simulation trace on-the-fly and with respect to a set of monitors that the central Traviando instance has sent to it. If a monitor verdict is $\perp$, a short witness trace of the last $k \approx 150$ events is sent to the central Traviando instance (the value of k is configured when clients are launched). Traviando makes these short witness traces available for further visualization and inspection by a human analyst.

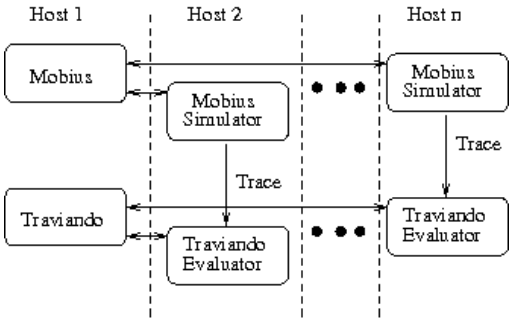

Figure 1. Architecture for Runtime Verification of Distributed Simulation Runs

5. Evaluation of Peer-to-Peer Botnet Model

In this section, we discuss an extension of a peer-to-peer botnet model [20]. Van Ruitenbeek and Sanders developed a stochastic model to examine how different factors impact the growth of a botnet which in turn provides guidance for defense tactics and for the design of future anti-malware systems. The model itself can be understood as an open system with an arrival stream of attacks that go through different stages when attacking a network. A single successful attack transforms a network node either into a propagation bot that will initiate further attacks or into a working bot that will perform work for the overall botnet. Propagation bots and working bots alternate between an active and an inactive mode. The model also includes the effects of defense mechanisms: during the infection phase, an attack may be unsuccessful to move from one stage to the next, which means it is destroyed and removed. Similar for the propagation and working bots, they can be identified and removed. Van Ruitenbeek and Sanders study the effect of varying infection probabilities and of varying bot removal rates. They show how prevention measures as well as detection and disinfection methods can effectively fight back botnet growth.

When running the corresponding Möbius model, we were interested in the average lifespan of a bot. Since the original model accounts for the number of bots being in a particular state, we need to extend the model. Our extension uses the concept of a tagged customer, such that we can measure the lifespan of a particular tagged bot. We also realized that the simulations were computationally expensive to run. So we generated a web report for a sample simulation trace as described in [14] and recognized that the simulator is mostly simulating events for an unsuccessful initial infection. So in what follows, we discuss two model modifications: a) an extension with a tagged customer, i.e., a distinct submodel that models the behavior of a single bot after an initially successful infection, and b) a modification of the arrival model such that we generate the same stream of successful bot infections but omit all those infections that are unsuccessful in reaching even the first stage of a bot infection. In addition to these conceptual changes, we also reorganize the model into a composed model with separate submodels for the arrival process, the botnet and the tagged bot. Figure 2 shows the overall composed model in Möbius. We also add a state variable to the botnet submodel that enforces an upper bound on the maximum number of bots. We use a bound of $2^{31} - 1$ to avoid the otherwise possible overflow of integer state variables. Finally, we introduce three global boolean variables, one for each submodel, and add one to the enabling condition of each activity such that we can disable a complete submodel for test runs.

Remodeling the arrival stream. In [20], initial bot infections occur with an exponentially distributed arrival rate whose value is a function of the number of active propagation bots. An initial bot infection when it occurs has a fixed probability p to successfully enter the system. With probability $1 - p$, it is unsuccessful and has no further effect. So, whenever the number of active propagation bots is set to a particular value, we will see a sequence of expo-

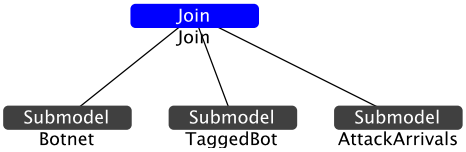

Figure 2. Extended P2P Botnet Model

nentially distributed interarrival times till the first successful infection. A new sequence starts if the number of active propagation bots is changed or an initial infection succeeds. So the number of attempts for a successful initial bot infection is a geometric random variable K with parameter p and $P\{K = n\} = (1-p)^{n-1}p$ is its distribution. The distribution of the time from a restart with a newly assigned number of active propagation bots to a first successful initial bot infection is the same as the distribution of the time between two successful initial bot infections, namely an Erlang distribution with rate λ (which depends on the number of active propagation bots) and K as the number of stages.

Modeling this in Möbius is possible. The delay of the activity that corresponds to the initial bot infection is defined as an Erlang distribution with parameters settings of M as `return(K->Mark()) ;`
and Beta as `return(K->Mark()/RateOfAttack*`
`(ActivePropagationBot->Count->Mark());`
This activity has an input gate with predicate
`ActivePropagationBot->Count->Mark()`
`== ErlangMemory->Count->Mark()`
and input function
`K->Mark() = TheDistribution->Geometric`
`(ProbInstallInitialInfection)+1 ;`
The input predicate checks if the number of active propagation bots equals a memorized value currently in use for the Erlang distribution. If those two differ, another activity updates the ErlangMemory variable, which in turn restarts the Erlang arrival activity. The input function is evaluated before the activity performs, which we use to update K, the number of stages used when sampling from an Erlang distribution to determine the overall delay. We checked the consistency of the original arrival stream and our new version by running copies of both in a separate test model and measured the generated arrival rates for various configurations.

Adding a Tagged Bot. When one wants to measure performance characteristics of an individual object, while the overall model only considers totals of those objects, the tagged customer pattern applies. The approach itself is a classic in performance modeling, in particular for an ergodic queueing system that can be analyzed by studying a single customer (the tagged customer) that enters the system according to an initial distribution representing the state of the system just after the arrival of a customer. For a recent publication, see [8]. In our case, we are mainly interesting in the modeling approach. We reorganize the existing botnet model of Van Ruitenbeek and Sanders into a composed Möbius model that consists of three atomic models as in Fig. 2: one that captures the dynamics of the arrival stream of successful initial bot infections as described above, one that describes the botnet SAN model as in [20] but with an upper bound to the number of bots, and finally a copy of the botnet SAN for the tagged bot, where the number of bots is set to be at most one. The arrival stream delivers a new bot either to submodel *Botnet* or *TaggedBot* with a preference for the latter. We measure the lifespan of an individual bot by measuring the time the tagged bot SAN is not idle and how often a new tagged bot is created. The model contains six ways to remove an existing bot in *TaggedBot* (and same for *Botnet*).

Generation of Sample Traces Σ. Sample traces should be representative for the functional behavior. So they should cover the behavior of all activities and the range of values for all state variables. Since timing is not reflected by our LTL formulas, we can adjust rates to reduce the effect of different time scales that are present in the model. In order to observe all activities, we consider two particular model configurations, one exercises only the arrival and tagged bot submodel, the other exercises only the arrival and botnet submodel. We obtain two sample traces, which in combination cover all activities of the overall model. In order to exercise ranges of values, we consider an additional set of six configurations in which all submodels are enabled, but the initial number of bots differs: one configuration has a minimal initial number of a single active propagation bot to start with, one has a medium initial total number of 500 bots and the remaining three configurations have a maximum initial number of $2^{31} - 1$ bots and differ only in the specific initial state of those bots. This set up is inspired by the general recommendations in software testing on how to test a finite range of values. In total, we obtain eight sample traces of length $1640 \leq n \leq 376878$. We use Traviando's reporting functionality [14] to check that all activities occur and that ranges of values are sufficiently covered.

Derivation of Formulas Running Daikon and Traviando on this set Σ of traces, we obtain 1815 formulas with Daikon, 169 with Traviando. We will go through the different types of formulas we described in Section 3 and highlight specific properties to illustrate our experiences.

Type 1. Daikon rightly produces a discrete set of values for all state variables of the tagged bot submodel, which

only take values in $\{0, 1\}$. Daikon also produces lower bounds for 15 variables as well as relative upper bounds for 3 variables, e.g., that the total number of active propagation bots is an upper bound for the number of active propagation bots in the botnet submodel. It does not produce any absolute upper bounds. Traviando produces lower and upper bounds for all 33 variables. Since our set of test runs Σ was selected to cover borderline cases for ranges of values so the computed upper bounds (even if not exactly at the maximum legal value $2^{31} - 1$ for all state variables) are high enough to cover experiments of interest. In this way, a formal specification generated from Σ will let an analyst know during a runtime verification if the exercised range of values in Σ is insufficient and thus the testing phase was not rigid enough.

Type 2. Monotonicity is not observed for any variables in this model and hence Σ does not rise to such formulas. However, both Daikon and Traviando would be able to recognize these properties. A combination of monotonicity and bounds of type 1 raises the question of what the model does if a bound is reached. If both observations are valid, then the only legal behavior for the model is to keep the value of that variable constant for the rest of the simulation run. When generating these formulas, feedback to an analyst must be given to consider this case.

Type 3. Daikon only obtains very simple equations for weighted sums, e.g., that the total number of initial bot infections is the sum of those at *Botnet* and *TaggedBot* or special cases such as $x = y$. Traviando recognizes invariants from an invariant computation that solves a linear equation system. For the botnet model, we obtain invariants for the tagged bot submodel as well as for the botnet submodel since both models are closed subsystems (closed due to the *idle* respectively *capacity* variable which limit the number of tagged bots to one and the number of bots in the botnet submodel to $2^{31} - 1$). Since we added state variables that give the total number of bots in a particular state (by adding those in the botnet submodel and the tagged bot subodel), the model set of weight vectors W that is computed contains eight different vectors due to the possibility to replace a particular variable by a sum of two others. The number of invariants as well as some of Daikon's equations may alert an analyst to clean up the model description to obtain a parsimonious model if desired. There is no such equation for variables of the submodel that models the arrivals, because variables represent data values that are changed in an irregular manner. The activities that modify these variables are challenging for an invariant calculation, e.g., Σ yields $|\Delta(initialbotinfection)| = 379$ for the activity that models the Erlang distribution for arrivals.

Type 4, 5, and 6. The characterization of when an action occurs, what it does and what the resulting state is, this is the strength of Daikon. Daikon generates hundreds of equations, inequalities and value sets for combinations of variables, be it with values of variables referring to the starting state for preconditions, referring to the successor state for postconditions or mixtures of both for state transformation functions. Daikon recognizes relationships among different variables, e.g., $x \leq y$, $x' = y$, or $x' > y'$ for the occurrence of some action a. Traviando generates formulas for each action, but it is not checking relations between different variables. It solely recognizes preconditions as a conjunction of ranges of values of individual variables. For state transformation functions, it creates two formulas for each action a. One lists all observed cases in $\Delta(a)$. The other consists only of a conjunction over all variables $v \in \mathcal{V}$ that gives bounds for the observed difference $v' - v$. The latter is particularly useful for the action of the initial bot infection that assigns a random value to state variable K. We observe 379 vectors in $\Delta(initialbotinfection)$ which makes the formula that enumerates those cases impractical. The alternative formula that uses ranges simply states that $-115 \leq K' - K \leq 132$ with respect to variable K.

Type 7. Daikon does not contribute to this type of formulas. Traviando generates a formula for each action with an average of 8.3 possible successor actions (out of 28). An interesting case results from the action that reacts on a change on the number of active propagation bots and forces a restart of the arrival process. Since this action reacts almost immediately, there are three actions that have only this action as its only observed immediate successor, namely the actions that model that a propagation bot 1) goes into sleep mode, 2) returns to active mode, or 3) is identified and removed. This is a relevant property for a correct model.

Runtime Verification. A series of experiments that varies the probabilities of attacks being successful or defense mechanisms to be successful does not modify functional properties of the model. A runtime verification of the generated sets of formulas reveals as expected that almost all of the derived formulas hold. The ones that may fail and depend on the executed simulation model are some statistical ones obtained with Daikon that are not backed by the model and the incomplete representation of the state transformation functions in the arrival model. All types of formulas considered result in rather simple and small deterministic finite automata. The socket communication between Möbius simulators and Traviando monitoring clients slows down the Möbius simulation. We performed a runtime verification for a simulation study of 18 model configurations where Möbius runs a distributed simulation across five machines. Monitoring 1815 Daikon-generated formulas, we

observed a slowdown by a factor of 41. Monitoring 168 Traviando-generated formulas, which are less numerous but longer, imposed a slowdown by a factor of 25, and monitoring all 1983 formulas resulted in a slowdown by a factor of 62. Unless future improvements for the communication of trace data and the evaluation of formulas reduces this overhead substantially, the runtime verification can be applied only to simulation experiments that are short enough to keep the overall running times in an acceptable range.

6. Conclusion

We presented an approach to derive a formal specification of the functional behavior of a simulation model from a set of finite simulation traces Σ. Set Σ is obtained from running simulations of a stochastic discrete event systems model for verification and validation purposes such that it is reasonable to assume that Σ represents correct model behavior. The set of properties that we derive are mainly safety properties formulated in LTL. This set of formulas contributes to the long-standing challenge of having appropriate documentation of a simulation model as well as achieving its verification and validation. In addition to documentation purposes, the generated set of formulas (possibly enriched with manually generated ones) can be used for runtime verification of a simulation model for production runs or for consistency evaluations of a modified model much like regression testing in software development. We implemented our approach with the help of Möbius, Daikon, and Traviando in order to obtain a fully automated framework for the generation of formulas as well as its runtime verification.

Acknowledgements. We thank the reviewers who helped to improve the paper. We also thankfully acknowledge that the James Monroe Scholars program and the Chappell Fellowship program of the College of William and Mary provided funding for this undergraduate research project.

References

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

[2] A. Baliga, V. Ganapathy, and L. Iftode. Automatic inference and enforcement of kernel data structure invariants. *Computer Security Applications Conf.*, 0:77–86, 2008.

[3] A. Bauer, M. Leucker, and C. Schallhart. Monitoring of real-time properties. In S. Arun-Kumar and N. Garg, editors, *FSTTCS*, volume 4337 of *LNCS*, pages 260–272. Springer, 2006.

[4] A. Bauer, M. Leucker, and C. Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In O. Sokolsky and S. Tasiran, editors, *RV*, volume 4839 of *LNCS*, pages 126–138. Springer, 2007.

[5] A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. Technical report, TUM-I0724, TU München, 2007.

[6] N. Benes, L. Brim, I. Cerna, J. Sochor, P. Verkova, and B. Zimmerova. Partial order reduction for state/event LTL. In M. Leuschel and H. Wehrheim, editors, *IFM 2009*, volume 5423 of *LNCS*, pages 307–321. Springer, 2009.

[7] S. Bensalem and D. Peled, editors. *Runtime Verification, 9th Int. Workshop, RV 2009, France, 2009.*, volume 5779 of *LNCS*. Springer, 2009.

[8] L. Bodrog, G. Horváth, S. Rácz, and M. Telek. A tool support for automatic analysis based on the tagged customer approach. In *QEST*, pages 323–332. IEEE CS, 2006.

[9] S. Chaki, E. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. State/event based software model checking. In E. Boiten, J. Derrick, and G. Smith, editors, *IFM 2004*, volume 2999 of *LNCS*, pages 128–147. Springer, 2004.

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

[11] N. Dinesh, A. K. Joshi, I. Lee, and O. Sokolsky. Checking traces for regulatory conformance. In M. Leucker, editor, *RV*, volume 5289 of *LNCS*, pages 86–103. Springer, 2008.

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

[13] P. Kemper. Recovering model invariants from simulation traces with Petri net analysis techniques. In *Winter Simulation Conference*. ACM, 2009.

[14] P. Kemper. Report generation for simulation traces with Traviando. In *DSN*, pages 347–352. IEEE CS, 2009.

[15] P. Kemper and C. Tepper. Automated trace analysis of discrete event system models. *IEEE TSE*, 35,2:195–208, 2009.

[16] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: A run-time assurance approach for java programs. *Formal Methods in System Design*, 24(2):129–155, 2004.

[17] M. Leucker and C. Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009.

[18] O. Mondragon, A. Q. Gates, S. Roach, H. Mendoza, and O. Sokolsky. Generating properties for runtime monitoring from software specification patterns. *Int J Software Engineering and Knowledge Engineering*, 17(1):107–126, 2007.

[19] G. Rosu, W. Schulte, and T.-F. Serbanuta. Runtime verification of c memory safety. In Bensalem and Peled [7], pages 132–151.

[20] E. V. Ruitenbeek and W. H. Sanders. Modeling peer-to-peer botnets. In *QEST*, pages 307–316. IEEE CS, 2008.

[21] S. Sahoo, M.-L. Li, P. Ramachandran, S. Adve, V. Adve, and Y. Zhou. Using likely program invariants to detect hardware errors. In *DSN*. IEEE CS, 2008.

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

[23] W. Zhou, O. Sokolsky, B. T. Loo, and I. Lee. DMaC: Distributed monitoring and checking. In Bensalem and Peled [7], pages 184–201.