An Expressive Verification Framework for State/Event Systems

Sagar Chaki, Edmund Clarke, Orna Grumberg,
Joël Ouaknine, Natasha Sharygina,
Tayssir Touili, Helmut Veith

June 2004
CMU-CS-04-145

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research was sponsored by the Semiconductor Research Corporation (SRC) under contract no. 99-TJ-684, the National Science Foundation (NSF) under grants no. CCR-9803774 and CCR-0121547, the Office of Naval Research (ONR) and the Naval Research Laboratory (NRL) under contract no. N00014-01-1-0796, the Army Research Office (ARO) under contract no. DAAD19-01-1-0485, the Austrian Science Fund Project N-Z29 N04, the EU Networks GAMES and ECRYPT, and was conducted as part of the PACC project at the Software Engineering Institute (SEI). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of SRC, NSF, ONR, NRL, ARO, the U.S. Government or any other entity.

Keywords: Temporal logic, compositional verification, software model checking

Abstract

Specification languages for concurrent software systems need to combine practical algorithmic efficiency with high expressive power and the ability to reason about both states and events. We address this question by defining a new branching-time temporal logic SE-AΩ which integrates both state-based and action-based properties. SE-AΩ is universal, i.e., preserved by the simulation relation, and thus amenable to counterexample-guided abstraction refinement. We provide a model-checking algorithm for this logic, and describe a compositional abstraction-refinement loop which exploits the natural decomposition of the concurrent system; the abstraction and refinement steps are performed over each component separately, and only the model checking step requires an explicit composition of the abstracted components. For experimental evaluation, we have integrated the presented algorithms in the software verification tool MAGIC, and determined a previously unknown race condition error in a piece of an industrial robot control software.

1 Introduction

The practical effectiveness of model checking is characterized by a trade-off between the expressive power of the specification formalism and the complexity of the corresponding model checking algorithm. For software verification, this problem is even more acute, since software is harder to specify, and state explosion is exacerbated by the concurrent execution of multiple components. The expressive power of temporal logics such as CTL or LTL is quite limited when it comes to specifying, e.g., the periodicity of events. The last decade has seen several attempts at extending the expressiveness of temporal logics [8, 32, 30, 31, 29, 13]. Recently, Clarke et al. [11] have investigated a family of universal branching logics, called $A\Omega$, which are extensions of ACTL by sets Ω of ω-regular path operators. A subtle property of $A\Omega$ is the *monotonicity* of the path operators: the semantics guarantees that the extended path operators cannot be used to implicitly define negation. While this property comes for free with the standard temporal path operators, its presence is crucial for obtaining extended *universal* branching logics. Such logics are preserved by simulation, and are therefore amenable to existential abstraction [9, 11].

Another shortcoming of standard temporal logics stems from the fact that for the verification of concurrent software conducted at the source code level, one needs to specify both *state* information (program counter location, memory contents) and *communication* among components. For example, the Bluetooth L2CAP specification [14] asserts that "when an `L2CAP_ConnectRsp` event is received in a `W4_L2CAP_CONNECT_RSP` state, within one time unit, an L2CAP process may send out an `L2CA_ConnectInd` event, disable the RTX timer, and move to state `CONFIG`." As this example shows, both states (`W4_L2CAP_CONNECT_RSP` and `CONFIG`) and events (`L2CAP_ConnectRsp` and `L2CA_ConnectInd`) are required to properly capture the desired L2CAP behavior.

Generally, in concurrent programs, communication among modules proceeds via actions (events) which can represent function calls, requests and acknowledgments, etc. These communications can be data dependent and carry data on its channels. Existing model checking techniques typically use either *state-based* or *event-based* formalisms to represent finite-state models of programs. In principle, both frameworks are interchangeable: an action can be encoded as a change in state variables, and likewise one can equip a state with different actions to reflect different values of its internal variables. Neither approach on its own is practical, however, when it comes to the specification of data-dependent communication claims: considerable domain expertise is then required to annotate the program and to specify proper specifications in temporal logic.

In this paper, we define the specification logic SE-AΩ which combines the high expressive power of $A\Omega$ with the ability to specify states and events simultaneously. The hybrid state/event-based semantics of SE-AΩ allows us to represent both software implementations and specifications directly without program annotations or privileged insights into program execution. Note that, for example, there is no natural generic extension of standard operators such as **U** (*until*) to a state/event based framework (see, e.g., [18]); SE-AΩ, however, enables us to employ different variants of CTL operators for actions and data valuations simultaneously at no additional expense. Notwithstanding its high expressive power and versatility, SE-AΩ lends itself naturally to an efficient verification strategy which combines counterexample-guided abstraction refinement (CEGAR [20, 7]) and compositional reasoning: starting with a coarse initial abstraction, our CEGAR scheme computes increasingly precise abstractions of the target system by analyzing spurious counterexamples until either a real counterexample is obtained or the system is found to be correct. More precisely, given a system M composed of n concurrent components $M_1, \ldots, M_n$, and a SE-AΩ specification φ, the verification of $M \models \varphi$ proceeds as follows:

1. **Abstract.** Create an abstraction $\widehat{M}$ such that all behaviors of $\widehat{M}$ are preserved by M. This is done component-wise without constructing the full state space of M.
2. **Verify.** Verify whether $\widehat{M} \models \varphi$. If so, report success and exit. Otherwise, extract an abstract counterexample $\widehat{C}$ that indicates in which way φ fails in $\widehat{M}$.
3. **Refine.** Check whether $\widehat{C}$ gives rise to a real counterexample over M. If $\widehat{C}$ corresponds to a genuine behavior of M then report a failure along with a fragment of each M_i that illustrates why $M \not\models \varphi$. If $\widehat{C}$ is spurious, on the other hand, refine $\widehat{M}$ using $\widehat{C}$ to obtain a more precise abstraction and repeat from step 1. This refinement step, like the initial abstraction, is performed component-wise.

Of the three steps in this abstract-verify-refine process only the verification stage of our technique requires the explicit composition of a system. The other stages can be performed one component at a time. Since verification is performed only on abstractions (which are usually much smaller than the corresponding concrete systems), our verification approach is able to significantly reduce the state space explosion problem. Another key characteristic of our algorithm is that the verification step handles both states and events *directly*, i.e., without conversion into either a pure state-based or a pure event-based framework. The model checking is therefore significantly more efficient than alternative conversion-based approaches, since it has been observed that conversion can lead to a quadratic blowup in both time and space even for reachability properties [2].

Note that the universality of SE-AΩ is crucial for the correctness of our approach, and that the verification step uses automata theoretic methods to evaluate the ω-regular path operators.

To the best of our knowledge, this is the first counterexample-guided, compositional abstraction refinement scheme to perform verification of branching-time specifications. We have implemented our approach in our C verification tool MAGIC [22] which extracts state/event finite-state models from C programs automatically via predicate abstraction [28, 3]. Our experiments with a piece of robot controller software resulted in the detection of a complicated race condition error.

The rest of this article is organized as follows. In Section 2 we summarize related work. This is followed by some preliminary definitions defined in Section 3. In Section 4 we present the SE-AΩ logic, followed by model checking, counterexample validation and abstraction refinement procedures described in Section 5. Finally, we give a brief overview of the application of our techniques in Section 6.

2 Related Work

Extensions of temporal logics to increase the expressiveness of temporal operators have been proposed by various authors [8, 32, 30, 31, 29, 13]. Wolper [32] and Vardi and Wolper [31] extended LTL by regular expressions and Büchi automata respectively. Vardi and Wolper [30] and Thomas [29] have proposed extended branching-time logics, but have not addressed model checking. Clarke et al. [8] describe the logic ECTL that similarly to our work considers ω-regular automata in the context of branching-time logic. However, this work does not deal with abstraction refinement or compositional methods. Clarke et al. [11] define a class $A\Omega$ of universal branching logics (cf. Section 1) for a systematic study of the complexity and completeness of counterexamples in model checking. The work of [11], however, does not define a model checking algorithm for $A\Omega$. Our work extends the $A\Omega$ logic with the combined state/event expressiveness and provides a model checking algorithm for SE-AΩ which also applies to $A\Omega$.

State/event-based notations have been explored by a number of authors [25, 18, 17, 2]. The novelty of our approach lies in the way in which we efficiently integrate an expressive state/event formalism with powerful state space reduction techniques, namely CEGAR and compositional reasoning. In this respect, not only do we substantially extend the expressiveness of the state/event linear temporal logic SE-LTL presented in [2], but we also show how to validate *branching (tree-like)* counterexamples in a *compositional* manner, based on new results relating simulation and weak simulation relations for parallel processes (see Theorem 4 in Section 5).

The formalization of a general notion of abstraction first appeared in [12]. The abstractions used in our approach are *conservative*. They are guaranteed to preserve ‘undesirable’ properties of the system (e.g., [19, 9]). Conservative abstractions usually lead to significant reductions in the state space but in general require an iterated abstraction refinement mechanism (such as CEGAR) in order to establish specification satisfaction. CEGAR has been used, among others, in [24] (in non-automated form), and [1, 26, 21, 15, 6, 10]. In particular, CEGAR-based schemes have been used for the verification of safety properties [1, 7, 15, 3] as well as liveness [2] properties.

Compositionality and abstraction have been extensively studied in process algebra (e.g., [16, 23, 27]). Abstraction and compositional reasoning have been combined [4] within a single CEGAR scheme to verify safety properties of concurrent C programs.

3 Preliminaries

Definition 1 (Labeled Kripke Structure). *A labeled Kripke structure (LKS) is a 6-tuple* $(S, init, AP, \mathcal{L}, \Sigma, T)$ *where (i)* S *is a finite non-empty set of states, (ii)* $init \in S$ *is an initial state, (iii)* AP *is a finite set of atomic state propositions, (iv)* $\mathcal{L} : S \to 2^{AP}$ *is a state-labeling function, (v)* Σ *is a finite set of actions (alphabet) and (vi)* $T \subseteq S \times \Sigma \times S$ *is a transition relation.*

Given an LKS $M = (S, init, AP, \mathcal{L}, \Sigma, T)$, we write $S(M)$, $init(M)$, $AP(M)$, $\mathcal{L}(M)$, $\Sigma(M)$ and $T(M)$ to mean S, $init$, AP, $\mathcal{L}$, Σ and T respectively. Given $s, s' \in$

S and $a \in \Sigma$ we write $s \xrightarrow{a} s'$ to mean $(s, a, s') \in T$. Also, let $Succ(s, a) = \{s' \in S \mid s \xrightarrow{a} s'\}$ and $Enabled(s) = \{a \in \Sigma \mid Succ(s, a) \neq \emptyset\}$. Finally, a *path* of M is an infinite sequence of consecutive transitions $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \ldots$. Note that we do not require paths to begin with $init$.

Definition 2 (Parallel Composition). *Let M_1 and M_2 be two LKSs such that $AP(M_1) \cap AP(M_2) = \emptyset$. Then the parallel composition of M_1 and M_2, denoted by $M_1 \| M_2$, is an LKS obeying the following conditions: (i) $S(M_1 \| M_2) = S(M_1) \times S(M_2)$, (ii) $init(M_1 \| M_2) = (init(M_1), init(M_2))$, (iii) $AP(M_1 \| M_2) = AP(M_1) \cup AP(M_2)$, and (iv) $\Sigma(M_1 \| M_2) = \Sigma(M_1) \cup \Sigma(M_2)$. Moreover, for all $s_1, s_1' \in S(M_1)$, $s_2, s_2' \in S(M_2)$, and $a \in \Sigma(M_1 \| M_2)$, the labeling function $\mathcal{L}(M_1 \| M_2)$ and the transition relation $T(M_1 \| M_2)$ are defined as follows:*

- $\mathcal{L}(M_1 \| M_2)((s_1, s_2)) = \mathcal{L}(M_1)(s_1) \cup \mathcal{L}(M_2)(s_2)$.
- *If $s_1 \xrightarrow{a} s_1'$ and $s_2 \xrightarrow{a} s_2'$ then $(s_1, s_2) \xrightarrow{a} (s_1', s_2')$.*
- *If $s_1 \xrightarrow{a} s_1'$ and $a \notin \Sigma(M_2)$ then $(s_1, s_2) \xrightarrow{a} (s_1', s_2)$.*
- *If $s_2 \xrightarrow{a} s_2'$ and $a \notin \Sigma(M_1)$ then $(s_1, s_2) \xrightarrow{a} (s_1, s_2')$.*

This notion of parallel composition is derived from CSP [16, 27]; it is commutative and associative, so that no parentheses are needed when composing more than two LKSs together.

Definition 3 (Simulation). *Let M_1 and M_2 be LKSs with $\Sigma(M_1) = \Sigma(M_2) = \Sigma$, and $AP(M_2) = AP(M_1)$. A relation $R \subseteq S(M_1) \times S(M_2)$ is said to be a simulation relation iff it satisfies the following conditions:*

1. *If $(s_1, s_2) \in R$ then $\mathcal{L}(M_1)(s_1) = \mathcal{L}(M_2)(s_2)$.*
2. *For any $s_1, s_1' \in S(M_1)$, $s_2 \in S(M_2)$, and $a \in \Sigma$, if $(s_1, s_2) \in R$ and $s_1 \xrightarrow{a} s_1'$ then there exists $s_2' \in S(M_2)$ such that $s_2 \xrightarrow{a} s_2'$ and $(s_1', s_2') \in R$.*

For two LKSs M_1 and M_2, if there exists a simulation relation R such that $(init(M_1), init(M_2)) \in R$ then we say that M_1 is simulated by M_2 and denote this by $M_1 \leqslant M_2$. The following is well-known [23]:

Theorem 1. *Let $M_1, \ldots, M_n$, $N_1, \ldots, N_n$ be LKSs such that $N_i \leqslant M_i$ for $1 \leq i \leq n$. Then $(N_1 \| \ldots \| N_n) \leqslant (M_1 \| \ldots \| M_n)$.*

In our framework, (existential) abstractions are obtained by 'lumping' together states of a concrete LKSs: abstract states are disjoint sets of concrete states; cf. [9]. In the remainder of this paper, we often use the letter M to denote a concrete LKS and its hatted counterpart $\widehat{M}$ to denote an abstract LKS. Note that an abstraction $\widehat{M}$ of M is entirely determined by an equivalence relation $R \subseteq S(M) \times S(M)$. We only consider *admissible* equivalence relations, i.e., we require that for all $s, s' \in S(M)$, whenever $(s, s') \in R$ then $\mathcal{L}(M)(s) = \mathcal{L}(M)(s')$. Given a state $s \in S(M)$, we denote its corresponding equivalence class by $[s]^R$ (or simply $[s]$ when R is clear from context.)

Definition 4 (Abstraction). *Let M be an LKS and R be an admissible equivalence relation on $S(M)$. Then M^R is the abstract quotient LKS induced by R such that (i) $S(M^R) = \{[s] \mid s \in S(M)\}$, (ii) $init(M^R) = [init(M)]$, (iii) $AP(M^R) = AP(M)$, (iv) for all $[s] \in S(M^R)$, $\mathcal{L}(M^R)([s]) = \mathcal{L}(M)(s)$ (well-defined since R is admissible), (v) $\Sigma(M^R) = \Sigma(M)$, and (vi) $T(M^R) = \{([s], a, [s']) \mid (s, a, s') \in T(M)\}$.*

For $s \in S(M)$ and $a \in \Sigma(M)$, the set of *abstract successors* of s along a is defined to be $AbsSucc(s, a) = \{[s'] \in M^R \mid (s, a, s') \in T(M)\}$.

It is easy to see that for any M and R, $M \preccurlyeq M^R$. Combining this with Theorem 1 we get the following result.

Lemma 1. *Let $M_1, \ldots, M_n$ be LKSs and $R_1, \ldots, R_n$ be equivalence relations. Then $(M_1 \| \ldots \| M_n) \preccurlyeq (M_1^{R_1} \| \ldots \| M_n^{R_n})$.*

4 The Logic SE-AΩ

Following [11], we define a universal branching-time logic called *State-Event Universal Logic* (SE-AΩ). The logic is interpreted over LKSs and can be used to specify properties involving both data and actions in a natural manner. SE-AΩ is defined in negation normal form, i.e., negations are only applied to atomic propositions. Unlike ACTL or ACTL*, it does not have a fixed set of operators. Rather, any ω-regular language can serve as a temporal operator. Since the logic is universal, every such operator is preceded by a universal path quantifier **A**.

Similarly to usual temporal operators, the new operators are applied to other formulas in the logic. Syntactically, this is done by defining an ω-regular language O over a set of markers that serve as placeholders for the formulas to which O is applied. Since SE-AΩ is aimed at specifying both actions and data, its operators can be applied to subsets of actions as well as formulas over atomic propositions.

Formally, let $Mark = \{m_1, m_2, \ldots\}$ be a denumerable set of *markers* and let $\overline{m} = \{m_1, \ldots, m_n\}$ be a finite subset of $Mark$. Let O be an ω-regular language over the alphabet $2^{\overline{m}}$. The corresponding n-ary temporal operator will be denoted by **O**. Let AP be a set of atomic propositions and Σ be a set of actions. Then the syntax of SE-AΩ is defined inductively as follows.

- If $p \in AP$ then p and $\neg p$ are formulas.
- If φ_1 and φ_2 are formulas then so are $\varphi_1 \vee \varphi_2$ and $\varphi_1 \wedge \varphi_2$.
- Let **O** be an n-ary temporal operator and for $1 \leq i \leq n$, φ_i be either a formula or a subset of Σ. Then $\mathbf{AO}(\varphi_1, \ldots, \varphi_n)$ is a formula.

The semantics of SE-AΩ is defined over LKSs. More precisely, given an SE-AΩ formula φ, an LKS M, and $s \in S(M)$ we write $M, s \models \varphi$ to mean that s satisfies φ, defined inductively as follows:

- For $p \in AP$, $M, s \models p$ iff $p \in \mathcal{L}(s)$ and $M, s \models \neg p$ iff $p \notin \mathcal{L}(s)$.
- $M, s \models \varphi_1 \vee \varphi_2$ iff $M, s \models \varphi_1$ or $M, s \models \varphi_2$.
- $M, s \models \varphi_1 \wedge \varphi_2$ iff $M, s \models \varphi_1$ and $M, s \models \varphi_2$.

- $M, s \models \mathbf{AO}(\varphi_1, \ldots, \varphi_n)$ iff for every path π starting from s, we have $M, \pi \models \mathbf{O}(\varphi_1, \ldots, \varphi_n)$ [as defined below].

Let $\pi = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \ldots$ be a path of M and π^i be its suffix starting from s_i. We first define when π satisfies an argument φ_k of the operator $\mathbf{O}$. $M, \pi \models \varphi_k$ iff either $\varphi_k \subseteq \Sigma$ and $a_0 \in \varphi_k$, or φ_k is a formula and $M, s_0 \models \varphi_k$.

Let $\mathbf{O}(\varphi_1, \ldots, \varphi_n)$ be as above, and O be the ω-regular language corresponding to $\mathbf{O}$. Recall that the alphabet of O is $2^{\overline{m}}$ where $\overline{m} = \{m_1, \ldots, m_n\}$. Then $M, \pi \models \mathbf{O}(\varphi_1, \ldots, \varphi_k)$ iff there is a word $o = o_1 o_2 \cdots \in O$ such that for all $i \geq 0$ and for all $m_k \in o_i$, $M, \pi^i \models \varphi_k$.

Lastly, we write $M \models \varphi$ to mean $M, \mathit{init}(M) \models \varphi$.

As an example, let $O = \{m_1, m_2\}^* \{m_1, m_3\}\{m_4\}\{\}^\omega$ be an ω-regular expression. Then $\mathbf{O}(\varphi, \{a\}, \{b\}, \psi)$ represents an ‘until’ operator that captures paths in which $\varphi \mathbf{U} \psi$ holds along a sequence of a actions ending with the action b. This example demonstrates that in addition to formulas φ_k that should hold, the logic SE-AΩ allows us to restrict the actions that can be performed, by using $\varphi_k \subseteq \Sigma$.

An important property of the logic SE-AΩ is that it is preserved by the simulation relation. This is formalized by the following lemma.

Lemma 2. *Given two LKSs M_1 and M_2 and an SE-AΩ formula φ, if $M_2 \models \varphi$ and $M_1 \leqslant M_2$, then $M_1 \models \varphi$.*

5 Compositional CEGAR Verification for SE-AΩ

Let $M_1, \ldots, M_n$ be LKSs and let φ be an SE-AΩ formula. In seeking to determine whether $M = M_1 \| \ldots \| M_n \models \varphi$, we wish to avoid constructing the full LKS M, since the size of its state space increases exponentially with the number of its components. We therefore first compute a (typically much smaller) abstraction $\widehat{M_i}$ of each component M_i, and only then check whether $\widehat{M} = \widehat{M_1} \| \ldots \| \widehat{M_n} \models \varphi$. If this holds, we conclude that $M \models \varphi$ as well. Otherwise, we extract from $\widehat{M}$ a counterexample $\widehat{C}$ violating φ, and check whether this counterexample is valid, i.e., whether it corresponds to a real execution of M. In the affirmative, we conclude that $M \not\models \varphi$. Otherwise, we use this spurious counterexample to refine our abstractions, and repeat the process until either a real counterexample is found or the property is shown to hold. The main strength of our approach is the fact that the abstraction, counterexample-validation, and refinement steps are all carried out one component at a time, so that it is never necessary to construct the full state space of the concrete system M.

5.1 Model Checking

Let $\widehat{M}$ be an LKS[1], $s \in S(\widehat{M})$, and φ be an SE-AΩ formula. We give a model-checking algorithm to determine whether $\widehat{M}, s \models \varphi$. We proceed by structural induction on φ,

[1] In the interests of consistency and clarity, we present our approach in both this section and the next in terms of the abstract LKS $\widehat{M}$, although it naturally applies to concrete systems as well.

starting with the case in which φ is of the form $\mathbf{AO}(\varphi_1, \ldots, \varphi_n)$. Let O be the ω-regular language over $\overline{m} = \{m_1, \ldots, m_n\}$ corresponding to $\mathbf{O}$. The algorithm consists of the following steps: (i) compute from $\widehat{M}$ and s the 'smallest' ω-regular language O_s over the alphabet $2^{\overline{m}}$ such that $\widehat{M}, s \models \mathbf{AO}_s(\varphi_1, \ldots, \varphi_n)$, and (ii) check whether O_s is 'subsumed' by O.

Intuitively, the idea is to interpret each path π in $\widehat{M}$ as a sequence of maximal subsets of formulas (among $\varphi_1, \ldots, \varphi_n$) that hold along π. We then check whether replacing each φ_j with the corresponding marker m_j results in a sequence belonging to O.

In order to do so we build an automaton B_s obtained from $\widehat{M}$ by replacing every action a, in transitions of the form (q, a, q'), with the subset of markers corresponding to the formulas that hold for the transition. More precisely, if φ_j is an SE-AΩ formula, we include the corresponding marker m_j provided that $\widehat{M}, q \models \varphi_j$, and if $\varphi_j \subseteq \Sigma(\widehat{M})$, we include m_j if $a \in \varphi_j$.

To make this more rigorous, we first recall the notion of *Büchi automata*:

Definition 5 (Büchi Automaton). *A Büchi automaton is a 5-tuple $B = (S, I, \Sigma, T, Acc)$ where (i) S is a finite non-empty set of states, (ii) $I \subseteq S$ is a set of initial states, (iii) Σ is a finite alphabet, (iv) $T \subseteq S \times \Sigma \times S$ is a transition relation, and (v) $Acc \subseteq S$ is a set of accepting states.*

A path of B is an infinite sequence $\pi = q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} \ldots$ such that $q_0 \in I$, and for every i, $(q_i, a_i, q_{i+1}) \in T$. π is accepting if it visits the set Acc infinitely often.

The language O_s is represented by a Büchi automaton B_s, which is derived from $\widehat{M}$ as follows: $B_s = (S_s, I_s, \Sigma_s, T_s, Acc_s)$, where (i) $S_s = S(\widehat{M})$, (ii) $I_s = \{s\}$, (iii) $\Sigma_s = 2^{\overline{m}}$, (iv) $Acc_s = S(\widehat{M})$, and (v) T_s is the set of transitions such that for each $(q, a, q') \in T(\widehat{M})$, T_s includes a transition $(q, \overline{m}', q')$ such that $\overline{m}' \subset \overline{m}$ and the following condition holds: for $0 \leq j \leq n$, $m_j \in \overline{m}'$ iff either $\varphi_j \subseteq \Sigma(\widehat{M})$ and $a \in \varphi_j$ or φ_j is a formula and $\widehat{M}, q \models \varphi_j$.

Note that in order to construct B_s we need to know whether $\widehat{M}, q \models \varphi_i$ for every $q \in S(\widehat{M})$ and every $i \in \{1, \ldots, n\}$. This is achieved by invoking the model checking algorithm recursively.

In the second step, we must check whether O_s is *subsumed* by O. Observe first that it is not enough to simply check whether $O_s \subseteq O$. That is because O and O_s are defined over the alphabet $2^{\overline{m}}$, and SE-AΩ is 'monotonic' (cf. [11]). In order to define monotonicity of SE-AΩ we consider two ω-regular languages O and O' over $\overline{m}$ that satisfy: for every $w = w_1 w_2 \cdots \in O$ there exists $w' = w'_1 w'_2 \cdots \in O'$ such that for every $i \geq 1$, $w_i \subseteq w'_i$. Then for every model $\widehat{M}$, if $M \models \mathbf{AO}'(\varphi_1, \ldots, \varphi_k)$ then $M \models \mathbf{AO}(\varphi_1, \ldots, \varphi_k)$. For example, let $\overline{m} = \{m_1, m_2, m_3\}$, and suppose that $O = \{m_2\}^\omega$ and that $O_s = \{m_1, m_2\}^\omega$. Then $\widehat{M}, s \models \mathbf{AO}_s(\varphi_1, \varphi_2, \varphi_3)$ and, thanks to monotonicity, $\widehat{M}, s \models \mathbf{AO}(\varphi_1, \varphi_2, \varphi_3)$ as well, even though $O_s \not\subseteq O$. To overcome this problem, we check whether $O_s \subseteq {\uparrow}O$, where ${\uparrow}O = (\{m_2\} + \{m_1, m_2\} + \{m_2, m_3\} + \{m_1, m_2, m_3\})^\omega$. The language ${\uparrow}O$ is called the *monotonic closure* of O and, intuitively, is obtained by replacing in O every occurrence of a set of markers $\overline{m}' \subseteq \overline{m}$ by the sum of all the sets of markers $\overline{m}''$ such that $\overline{m}' \subseteq \overline{m}'' \subseteq \overline{m}$. Formally:

Definition 6 (Monotonic Closure). *Let $B = (S_B, I_B, 2^{\overline{m}}, T_B, Acc_B)$ be a Büchi automaton accepting some ω-regular language O. The monotonic closure of O is the ω-regular language $\uparrow O$ accepted by the Büchi automaton $\uparrow B = (S_{\uparrow B}, I_{\uparrow B}, 2^{\overline{m}}, T_{\uparrow B}, Acc_{\uparrow B})$ constructed from B as follows: $S_{\uparrow B} = S_B$, $I_{\uparrow B} = I_B$, $Acc_{\uparrow B} = Acc_B$, and $T_{\uparrow B} = \{(q, \overline{m}'', q') \mid \exists \overline{m}' \subseteq \overline{m}'' \centerdot (q, \overline{m}', q') \in T_B\}$.*

The correctness of our two-step procedure is encapsulated by the following:

Theorem 2. *$\widehat{M}, s \models \mathbf{AO}(\varphi_1, \ldots, \varphi_n)$ iff $O_s \subseteq \uparrow O$.*

The other cases (in which φ is not an ω-regular operator) are straightforward. To summarize, $\widehat{M}, s \models \varphi$ iff:

- $p \in \mathcal{L}(s)$ if $\varphi = p$ and $p \notin \mathcal{L}(s)$ if $\varphi = \neg p$, where $p \in AP$.
- $\widehat{M}, s \models \varphi_1$ and $\widehat{M}, s \models \varphi_2$ if $\varphi = \varphi_1 \wedge \varphi_2$.
- $\widehat{M}, s \models \varphi_1$ or $\widehat{M}, s \models \varphi_2$ if $\varphi = \varphi_1 \vee \varphi_2$.
- $O_s \subseteq \uparrow O$ if $\varphi = \mathbf{AO}(\varphi_1, \ldots, \varphi_n)$, where O_s and $\uparrow O$ are defined as above.

5.2 Counterexample Generation

Let $\widehat{M}$ be an LKS, $s \in S(\widehat{M})$, and φ be an SE-AΩ formula. Suppose that $\widehat{M}, s \not\models \varphi$. In this section, we show how to compute a counterexample to φ, i.e., a fragment of $\widehat{M}$ beginning at state s that violates φ. As for the model-checking algorithm of SE-AΩ, we give a recursive procedure:

- If $\varphi = \varphi_1 \vee \varphi_2$, then compute counterexamples $\widehat{C}_1$ and $\widehat{C}_2$ to φ_1 and φ_2 respectively, and glue $\widehat{C}_1$ and $\widehat{C}_2$ at their initial states. Indeed, $\widehat{M}, s \not\models \varphi_1 \vee \varphi_2$ iff $\widehat{M}, s \not\models \varphi_1$ *and* $\widehat{M}, s \not\models \varphi_2$.
- If $\varphi = \varphi_1 \wedge \varphi_2$, then compute a counterexample either to φ_1 or to φ_2. Indeed, $\widehat{M}, s \not\models \varphi_1 \wedge \varphi_2$ iff $\widehat{M}, s \not\models \varphi_1$ *or* $\widehat{M}, s \not\models \varphi_2$.
- If $\varphi = \mathbf{AO}(\varphi_1, \ldots, \varphi_n)$, proceed as follows. Since $\widehat{M}, s \not\models \varphi$, there exists a pattern in O_s that is not in $\uparrow O$. Let $\pi = s_0 \xrightarrow{\overline{m}_0} s_1 \xrightarrow{\overline{m}_1} \ldots$ (where $s_0 = s$) be an accepting path of B_s such that the ω-word $\overline{m}_0 \overline{m}_1 \ldots$ does not belong to $\uparrow O$. Recall that by the definition of the automaton B_s, each transition $s_i \xrightarrow{\overline{m}_i} s'_i$ in T_{B_s} corresponds to a transition $s_i \xrightarrow{a_i} s'_i$ in $T(\widehat{M})$. Let therefore $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots$ be the corresponding path of π in $\widehat{M}$. This path then clearly violates $\mathbf{O}(\varphi_1, \ldots, \varphi_n)$. To compute a counterexample to φ, it suffices to take this path and to glue to each state s_i counterexamples to all formulas φ_j such that $\widehat{M}, s_i \not\models \varphi_j$. (Note that, while the path is infinite, it comprises of only finitely many distinct states.)

Owing to the direct manner in which a counterexample $\widehat{C}$ is extracted from an LKS $\widehat{M}$, there is a canonical mapping $\rho : S(\widehat{C}) \rightarrow S(\widehat{M})$ which satisfies the following conditions: (i) $\rho(init(\widehat{C})) = init(\widehat{M})$, (ii) for all $q \in S(\widehat{C})$, $\mathcal{L}(\widehat{C})(q) = \mathcal{L}(\widehat{M})(\rho(q))$, and (iii) if $(q, a, q') \in T(\widehat{C})$, then $(\rho(q), a, \rho(q')) \in T(\widehat{M})$. We shall make use of ρ later on in the refinement step.

Example 1. Figure 1 (a) shows an LKS M with $AP(M) = \{p, q\}$, $\Sigma(M) = \{a, b\}$, and initial state $S1$. (b) shows the abstract quotient LKS M^R induced by the equivalence relation R having equivalence classes $\{S1, S2\}$ and $\{S3, S4\}$. Let φ be the formula (in CTL^*-like notation) $\mathbf{AG}(\{a\} \Rightarrow \mathbf{A}(p \vee \mathbf{X}p \vee \mathbf{XX}p))$. φ asserts that on all paths, whenever the action a occurs from a state s, then the atomic proposition p either holds at s or, along any path starting at s, in one of the next two states. It is not hard to see that $M^R \not\models \varphi$, and indeed (c) shows a counterexample $\widehat{C}$ illustrating this. The dotted arrows from $\widehat{C}$ to M^R represent the canonical mapping ρ.

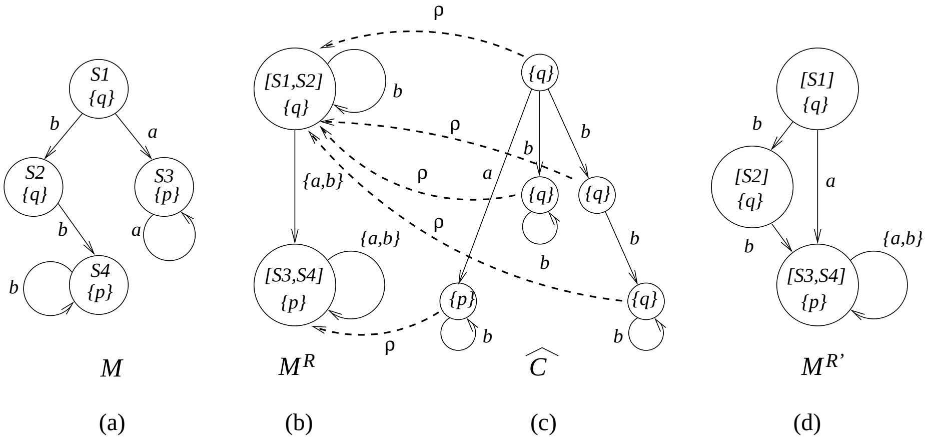

Fig. 1. (a) concrete LKS M; (b) Abstract LKS M^R; (c) counterexample $\widehat{C}$; (d) refined abstract LKS $M^{R'}$.

Observe, however, that the counterexample is in fact spurious. Indeed, the abstract LKS $M^{R'}$ pictured in (d) is a refinement of M^R induced by the equivalence relation R' having equivalence classes $\{S1\}$, $\{S2\}$, and $\{S3, S4\}$. Since $M^{R'} \models \varphi$, we conclude that $M \models \varphi$ as well.

5.3 Counterexample Validation

Suppose that $\widehat{M}, s \not\models \varphi$ for some SE-AΩ formula φ, and let $\widehat{C}$ be a counterexample to φ. Recall that $\widehat{M}$ is an abstraction of a concrete LKS M. We say that $\widehat{C}$ is a *valid* counterexample iff $\widehat{C} \leqslant M$. Indeed, from Lemma 2 we get:

Theorem 3. *Let φ be an SE-AΩ formula. If $\widehat{C} \leqslant M$ and $\widehat{C} \not\models \varphi$, then $M \not\models \varphi$.*

Intuitively, this holds because SE-AΩ formulas describe properties that are quantified over all possible paths of the structure.

This result suggests a way to formally check whether a counterexample $\widehat{C}$ is valid for a concrete system M or not. However, as mentioned earlier, when M is a concurrent C program built of components $M_1, \ldots, M_n$, we are faced with the problem that even if each component M_i has a finite state space, constructing the state space of M might be prohibitive in practice due to exponential blowup. To overcome this problem, we propose to check if the concrete system M simulates the counterexample $\widehat{C}$ in a compositional way by checking whether for every $i \in \{1, \ldots, n\}$, M_i weakly simulates the i^{th} projection of $\widehat{C}$.

Definition 7 (i^{th} Projection). *Let $M = M_1 \| \ldots \| M_n$ be a parallel composition of LKSs, and let $\widehat{C}$ be a further LKS. For any $i \in \{1, \ldots, n\}$, $\widehat{C}\upharpoonright_i$ is the LKS defined by: (i) $S(\widehat{C}\upharpoonright_i) = S(\widehat{C})$, (ii) $\mathit{init}(\widehat{C}\upharpoonright_i) = \mathit{init}(\widehat{C})$, (iii) $AP(\widehat{C}\upharpoonright_i) = AP(M_i)$, (iv) for any $s \in S(\widehat{C}\upharpoonright_i)$, $\mathcal{L}(\widehat{C}\upharpoonright_i)(s) = \mathcal{L}(\widehat{C})(s) \cap \mathcal{L}(M_i)$, (v) $\Sigma(\widehat{C}\upharpoonright_i) = \Sigma(M_i) \cup \{\tau\}$[2], and (vi) $T(\widehat{C}\upharpoonright_i)$ is defined as follows:*

- *If $(s, a, s') \in T(\widehat{C})$ and $a \in \Sigma(M_i)$ then $(s, a, s') \in T(\widehat{C}\upharpoonright_i)$.*
- *If $(s, a, s') \in T(\widehat{C})$ and $a \notin \Sigma(M_i)$ then $(s, \tau, s') \in T(\widehat{C}\upharpoonright_i)$.*

The introduction of τ actions also naturally leads to a *weak* version of simulation, which we define next specialized to the case in which only the system being simulated is capable of performing τ's.

Definition 8 (Weak Simulation). *Let $\widehat{C}$ and M be LKSs such that $\Sigma(\widehat{C}) = \Sigma(M) \cup \{\tau\}$ and $AP(\widehat{C}) = AP(M)$. A relation $R \subseteq S(\widehat{C}) \times S(M)$ is said to be a weak simulation relation iff R satisfies the following conditions:*

1. *If $(s_1, s_2) \in R$ then $\mathcal{L}(\widehat{C})(s_1) = \mathcal{L}(M)(s_2)$.*
2. *For any $s_1, s_1' \in S(\widehat{C})$, $s_2 \in S(M)$, and $a \in \Sigma(\widehat{C}) \setminus \{\tau\}$, if $(s_1, s_2) \in R$ and $s_1 \xrightarrow{a} s_1'$ then there exists $s_2' \in S(M)$ such that $s_2 \xrightarrow{a} s_2'$ and $(s_1', s_2') \in R$.*
3. *For any $s_1, s_1' \in S(\widehat{C})$ and $s_2 \in S(M)$, if $(s_1, s_2) \in R$ and $s_1 \xrightarrow{\tau} s_1'$ then $(s_1', s_2) \in R$.*

For two LKSs $\widehat{C}$ and M, if there exists a weak simulation relation R such that $(\mathit{init}(\widehat{C}), \mathit{init}(M)) \in R$ then we say that $\widehat{C}$ is weakly simulated by M and denote this by $\widehat{C} \preccurlyeq M$.

The following key result forms the basis of our compositional approach to counterexample validation.

Theorem 4 (Compositionality). *Let $M_1, \ldots, M_n$ be LKSs and let $\widehat{C}$ be a further LKS. Then $\widehat{C} \leqslant (M_1 \| \ldots \| M_n)$* iff *$\widehat{C}\upharpoonright_i \preccurlyeq M_i$* for $1 \leq i \leq n$.

Proof. (Sketch.) Consider the case $n = 2$; the general case is handled in a similar manner. Suppose first that $\widehat{C} \leqslant M_1 \| M_2$. Let $R \subseteq S(\widehat{C}) \times S(M_1 \| M_2)$ be a corresponding simulation relation. Define $R_1 = \{(s, s_1) \mid \exists s_2 \centerdot \big(s, (s_1, s_2)\big) \in R\}$, and $R_2 = \{(s, s_2) \mid \exists s_1 \centerdot \big(s, (s_1, s_2)\big) \in R\}$. It is readily verified that R_1 (resp. R_2) is a weak simulation relation between $\widehat{C}\upharpoonright_1$ and M_1 (resp. $\widehat{C}\upharpoonright_2$ and M_2). Therefore $\widehat{C}\upharpoonright_1 \preceq M_1$ and $\widehat{C}\upharpoonright_2 \preceq M_2$.

[2] We assume that τ is a fresh action not otherwise present in the alphabet of LKSs.

In the other direction, let R_1 and R_2 be two weak simulation relations witnessing $\widehat{C}\!\restriction_1 \preceq M_1$ and $\widehat{C}\!\restriction_2 \preceq M_2$ respectively. Let $R = \{(s,(s_1,s_2)) \mid (s,s_1) \in R_1 \wedge (s,s_2) \in R_2\}$. It is easy to check that R is a simulation relation between $\widehat{C}$ and $M_1 \| M_2$, as required. □

Putting everything together, we get:

Corollary 1. *Let $M_1, \ldots, M_n$ be LKSs, φ an SE-AΩ formula, and $\widehat{C}$ an abstract counterexample to $M_1 \| \ldots \| M_n \models \varphi$. Then $\widehat{C}$ is a valid counterexample iff $\widehat{C}\!\restriction_i \preccurlyeq M_i$ for every $i \in \{1, \ldots, n\}$.*

Checking whether $\widehat{C}\!\restriction_i \preccurlyeq M_i$ is done in a standard manner by a fixpoint computation of the maximal weak simulation relation between $\widehat{C}\!\restriction_i$ and M_i.

5.4 Abstraction Refinement

We now describe our counterexample-guided refinement procedure. Suppose that $\widehat{C} \not\preccurlyeq M$; then the counterexample $\widehat{C}$ is spurious, and we need to refine our abstraction $\widehat{M} = \widehat{M_1} \| \ldots \| \widehat{M_n}$. We achieve this by examining each of the abstractions $\widehat{M_i}$ individually: for $i \in \{1, \ldots, n\}$, we refine $\widehat{M_i}$ if $\widehat{C}\!\restriction_i \not\preccurlyeq M_i$. To this end, fix j an index in $\{1, \ldots, n\}$ such that $\widehat{C}\!\restriction_j \not\preccurlyeq M_j$. Recall that $\widehat{M_j}$ is a quotient LKS of the form $M_j^{R_j}$, where R_j is an equivalence relation on $S(M_j)$. Our refinement step consists in producing a strictly finer equivalence relation than R_j.

Recall the canonical mapping $\rho : S(\widehat{C}) \to S(\widehat{M})$ defined in Section 5.2, and let $\rho_j : S(\widehat{C}) \to S(\widehat{M_j})$ be its corresponding j^{th} projection. We can show that:

Lemma 3. *Suppose that for any $s \in S(\widehat{C})$, any $a \in \mathit{Enabled}(s)$, and any $s_1, s_2 \in \rho_j(s)$, we have that $\mathit{AbsSucc}(s_1,a) = \mathit{AbsSucc}(s_2,a)$. Then $\widehat{C}\!\restriction_j \preccurlyeq M_j$.*

Since, by assumption, $\widehat{C}\!\restriction_j \not\preccurlyeq M_j$, it follows from Lemma 3 that there exist a state $s \in S(\widehat{C})$, an action $a \in \mathit{Enabled}(s)$, and two states $s_1, s_2 \in \rho_j(s)$ such that $\mathit{AbsSucc}(s_1,a) \neq \mathit{AbsSucc}(s_2,a)$. Let R_j' be a new equivalence relation derived from R_j by sub-partitioning the equivalence class $\rho_j(s)$ as follows: q, q' belong to the same sub-partition iff $\mathit{AbsSucc}(q,a) = \mathit{AbsSucc}(q',a)$. R_j' is clearly a *proper* refinement of R_j (i.e. the number of R_j' will be strictly greater than that of R_j), and is moreover admissible since R_j was admissible. It should be noted that the refined abstract LKS $M_j^{R_j'}$ is however *not* guaranteed to refute the (projected) counterexample $\widehat{C}\!\restriction_j$. For example, Figure 1 shows the abstract LKS M^R and its refinement $M^{R'}$ which, in this case, refutes the spurious counterexample $\widehat{C}$.

Since the refinement procedure always yields a proper refinement and since each LKS is finite, the CEGAR-based SE-AΩ verification algorithm always terminates. In particular, spurious counterexamples are always eventually refuted.

6 Applications and Future Work

We implemented our compositional approach for verification of branching-time logics in the MAGIC tool, developed at Carnegie Mellon [5, 22]. MAGIC extracts finite LKS models from C programs. We applied the SE-AΩ model checking compositional loop for verification of a set of benchmarks whose abstract models were automatically extracted by MAGIC. We verified code provided by our industrial partner, one of the market leading robot manufacturers worldwide. We analyzed the IPC (InterProcess Communication) protocol used to mediate communication in a multi-threaded robot controller software. We model checked the synchronous communication portion of the IPC protocol which was implemented in terms of messages passed between queues owned by different threads. We specified a set of more than 20 SE-AΩ properties most of which were expressed using both states and events. That was required to make proper assertions on the communication actions carrying data.

We found a **bug** in the provided version of the IPC code and reported it to our industrial partner. The bug was a race condition in which a writer mistakenly blocks while trying to write to a queue that is not full. That bug violated the property that no communications timeout when they could be safely delivered. It had been undetected despite seven years of industrial use of the IPC, including a substantial testing phase.

We are currently examining other case studies. For future work, we would also like to carry out a systematic evaluation of the expressiveness of the SE-AΩ logic in comparison to other universal logics, estimating the complexity of our algorithms and improving the methods presented in this paper.

References

[1] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proc. of SPIN*. LNCS 2057, 2001.

[2] S. Chaki, E. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. State/event-based software model checking. In *Proc. of IFM*. LNCS, 2004. To appear.

[3] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *Proc. of ICSE*. IEEE Computer Society, 2003.

[4] S. Chaki, J. Ouaknine, K. Yorav, and E. Clarke. Automated compositional abstraction refinement for concurrent C programs: A two-level approach. In *Proc. of SoftMC*. ENTCS 89(3), 2003.

[5] Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. In *Proc. of ICSE*. IEEE Computer Society, 2003.

[6] P. Chauhan, E. M. Clarke, J. H. Kukula, S. Sapra, H. Veith, and D. Wang. Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis. In *Proc. of FMCAD*, 2002.

[7] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. of CAV*. LNCS 1855, 2000.

[8] E. M. Clarke, O. Grumberg, and R. P. Kurshan. A synthesis of two approaches for verifying finite state concurrent systems. *Logic Computat.*, 2(5):606–618, 1992.

[9] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *Proc. of TOPLAS*, 1994.

[10] E. M. Clarke, A. Gupta, J. H. Kukula, and O. Strichman. SAT based abstraction-refinement using ILP and machine learning techniques. In *Proc. of CAV*, 2002.

[11] Edmund M Clarke, Somesh Jha, Yuan Lu, and Helmut Veith. Tree-like counterexamples in model checking. In *Proc. of LICS*. IEEE Computer Society, 2002.

[12] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the SIGPLAN Conference on Programming Languages*, 1977.

[13] M. Dam. CTL* and ECTL* as fragments of the modal μ-calculus. *Theoretical Computer Science*, 126:77–96, 1994.

[14] J. Haartsen, Bluetooth Baseband Specification, version 1.0. `http://www.bluetooth.com`.

[15] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. of POPL*, 2002.

[16] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[17] M. Huth, R. Jagadeesan, and D. Schimidt. Modal transition systems: A foundation for three-valued program analysis. In *LNCS*, volume 2028, page 155. Springer, 2001.

[18] E. Kindler and T. Vesper. ESTL: A temporal logic for events and states. *Lecture Notes in Computer Science*, 1420:365–383, 1998.

[19] R. P. Kurshan. Analysis of discrete event coordination. In *Proc. REX Workshop 89*, volume 430. Springer LNCS, 1989.

[20] R. P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, 1994.

[21] Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental verification by abstraction. In *Proc. of TACAS*. LNCS 2031, 2001.

[22] MAGIC website. `http://www.cs.cmu.edu/∼chaki/magic`.

[23] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[24] G. Naumovich, L. A. Clarke, L. J. Osterweil, and M. B. Dwyer. Verification of concurrent software with FLAVERS. In *Proc. of ICSE*. ACM Press, 1997.

[25] R. De Nicola and F. Vaandrager. Three logics for branching bisimulation. *Journal of the ACM (JACM)*, 42(2):458–487, 1995.

[26] C. S. Păsăreanu, M. B. Dwyer, and W. Visser. Finding feasible counter-examples when model checking abstracted Java programs. In *Proc. of TACAS*, 2001.

[27] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.

[28] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Proc. of CAV*, volume 1254. Springer LNCS, 1997.

[29] W. Thomas. Computation tree logic and regular ω-languages. In *Proc. of Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*. LNCS 354, 1989.

[30] M.Y. Vardi and P. Wolper. Yet another process logic. In *Proc. of FOCS*, 1983.

[31] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.

[32] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56:72–99, 1983.