State/Event-based Software Model Checking*

Sagar Chaki   Edmund M. Clarke   Joël Ouaknine
Natasha Sharygina   Nishant Sinha

Computer Science Department, Carnegie Mellon University
5000 Forbes Ave., Pittsburgh PA 15213, USA

Abstract. We present a framework for model checking concurrent software systems which incorporates both states and events. Contrary to other state/event approaches, our work also integrates two powerful verification techniques, counterexample-guided abstraction refinement and compositional reasoning. Our specification language is a state/event extension of linear temporal logic, and allows us to express many properties of software in a concise and intuitive manner. We show how standard automata-theoretic LTL model checking algorithms can be ported to our framework at no extra cost, enabling us to directly benefit from the large body of research on efficient LTL verification.
We have implemented this work within our concurrent C model checker, MAGIC, and checked a number of properties of OpenSSL-0.9.6c (an open-source implementation of the SSL protocol) and Micro-C OS version 2 (a real-time operating system for embedded applications). Our experiments show that this new approach not only eases the writing of specifications, but also yields important gains both in space and in time during verification. In certain cases, we even encountered specifications that could not be verified using traditional pure event-based or state-based approaches, but became tractable within our state/event framework. We report a bug in the source code of Micro-C OS version 2, which was found during our experiments.

1 Introduction

Control systems ranging from smart cards to automated flight controllers are increasingly being incorporated within complex software systems. In many instances, errors in such systems can have dramatic consequences, hence the urgent need to be able to ensure and guarantee their correctness.

* 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, and was conducted as part of the Predictable Assembly from Certifiable Components (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.

In this endeavor, the well-known methodology of *model checking* [CE81, CES86, QS81, CGP99] holds much promise. Although most of its early applications dealt with hardware and communication protocols, model checking is increasingly used to verify software systems [SLA, BR01, BMMR01, BLA, HJMS02, HJMQ03, CDH+00, PDV01, Sto02, MAG, CCG+03, COYC03]. Unfortunately, applying model checking to software is complicated by several factors, ranging from the difficulty to model computer programs—due to the complexity of programming languages as compared to hardware description languages—to difficulties in specifying meaningful properties of software using the usual temporal logical formalisms of model checking. A third reason is the perennial state space explosion problem, whereby the complexity of verifying an implementation against a specification becomes prohibitive.

The most common instantiations of model checking to date have focused on finite-state models and either branching-time (CTL [CE81]) or linear-time (LTL [LP85]) temporal logics. To apply model checking to software, it is necessary to specify (often complex) properties on the finite-state abstracted models of computer programs. The difficulties in doing so are even more pronounced when reasoning about *modular* software, such as concurrent or component-based sequential programs. Indeed, in modular programs, communication among modules proceeds via actions (or events), which can represent function calls, requests and acknowledgments, etc. Moreover, such communication is commonly data-dependent. Software behavioral claims, therefore, are often specifications defined over combinations of program actions and data valuations.

Existing modeling techniques usually represent finite-state machines as finite annotated directed graphs, using either *state-based* or *event-based* formalisms. Although 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), converting from one representation to the other often leads to a significant enlargement of the state space. Moreover, neither approach on its own is practical when it comes to modular software, in which actions are often data-dependent: considerable domain expertise is then required to annotate the program and to specify proper claims.

This work, therefore, proposes a framework in which both state-based and action-based properties can be expressed, combined, and verified. The modeling framework consists of *labeled Kripke structures* (LKS), which are directed graphs in which states are labeled with atomic propositions and transitions are labeled with actions. The specification logic is a *state/event* derivative of LTL. This allows us to represent both software implementations and specifications directly without any program annotations or privileged insights into program execution. We further show that standard efficient LTL model checking algorithms can be applied, *at no extra cost in space or time*, to help reason about state/event-based systems. We have implemented our approach within the concurrent C verification tool MAGIC [MAG, CCG+03, COYC03], and report promising results in the examples which we have tackled.

The state/event-based formalism presented in this paper is suitable for both sequential and concurrent systems. One of the benefits of restricting ourselves to linear-time logic (as opposed to a more expressive logic such as CTL^* or the modal mu-calculus) is the ability to invoke the MAGIC compositional abstraction refinement procedures developed for the efficient verification of concurrent software [COYC03]. These procedures are embedded within a counterexample-guided abstraction refinement framework (CEGAR for short) [CGJ$^+$00], one of the core features of MAGIC. CEGAR lets us investigate the validity of a given specification through a sequence of increasingly refined abstractions of our system, until the property is either established or a real counterexample is found. Moreover, thanks to compositionality, the abstraction, counterexample validation, and refinement steps can all be carried out component-wise, thereby alleviating the need to build the full state space of the distributed system.

We illustrate our state/event paradigm with a current surge protector example, and conduct further experiments with the source code for OpenSSL-0.9.6c (an open-source implementation of the SSL protocol) and Micro-C OS version 2 (a real-time operating system for embedded applications). In the case of the latter, we discovered a bug, which it turns out was already known to the implementors of Micro-C OS. We contrast our approach with equivalent pure state-based and event-based alternatives, and show that the state/event methodology yields significant gains in human effort (ease of expressiveness), state space, and verification time, at no discernible cost.

The paper is organized as follows. In Section 2, we review and discuss related work. Section 3 defines our state/event implementation formalism, labeled Kripke structures. We also lay the basic definitions and results needed for the presentation of our compositional CEGAR verification algorithm. In Section 4, we present our state/event specification formalism, based on linear temporal logic. We review standard automata-theoretic model checking techniques, and show how these can be adapted to the verification task at hand. In Section 5, we illustrate these ideas by modeling a simple surge protector. We also contrast our approach with pure state-based and event-based alternatives, and show that both the resulting implementations and specifications are significantly more cumbersome. We then use MAGIC to check these specifications, and discover that the non-state/event formalisms incur important time and space penalties during verification.[1] Section 6 details our compositional CEGAR algorithm. In Section 7, we report on case studies in which we checked specifications on the source code for OpenSSL-0.9.6c and Micro-C OS version 2, which led us to the discovery of a bug in the latter. Finally, Section 8 summarizes the contributions of the paper and outlines several avenues for future work.

[1] In order to invoke MAGIC, we code the LKSs as simple C programs; the algorithm used by MAGIC implements the techniques described in the paper. Lack of space prevents us from discussing *predicate abstraction*, whereby MAGIC transforms a (potentially infinite-state) C program into a finite-state machine. We refer the reader to [CCG$^+$03] for a detailed exposition of this point.

2 Related Work

Counterexample-guided abstraction refinement [CGJ+00, Kur94], or CEGAR, is an iterative procedure whereby spurious counterexamples to a specification are repeatedly eliminated through incremental refinements of a conservative abstraction of the system. CEGAR has been used, among others, in [NCOD97] (in non-automated form), and [BR01, PDV01, LBBO01, HJMS02, CCK+02, CGKS02, COYC03].

Compositionality, which features centrally in our work, is broadly concerned with the preservation of properties under substitution of components in concurrent systems. It has been extensively studied, among others, in process algebra (e.g., [Hoa85, Mil89, Ros97]), in temporal logic model checking [GL94], and in the form of assume-guarantee reasoning [McM97, HQR00, CGP03].

The combination of CEGAR and compositional reasoning is a relatively new approach. In [BLO98], a compositional framework for (non-automated) CEGAR over data-based abstractions is presented. This approach differs from ours in that communication takes place through shared variables (rather than blocking message-passing), and abstractions are refined by eliminating spurious transitions, rather than by splitting abstract states.

The idea of combining state-based and event-based formalisms is certainly not new. De Nicola and Vaandrager [NV95], for instance, introduce 'doubly labeled transition systems', which are very similar to our LKSs. From the specification point of view, our state/event version of LTL is also subsumed by the modal mu-calculus [Koz83, Pnu86, BS01], via a translation of LTL formulas into Büchi automata. The novelty of our approach, however, is the way in which we efficiently integrate an expressive state/event formalism with powerful verification techniques, namely CEGAR and compositional reasoning. We are able to achieve this precisely because we have adequately restricted the expressiveness of our framework. To our knowledge, our work is the first to combine these three features within a single setup.

Kindler and Vesper [KV98] propose a state/event-based temporal logic for Petri nets. They motivate their approach by arguing, as we do, that pure state-based or event-based formalisms lack expressiveness in important respects.

Huth *et al.* [HJS01] also propose a state/event framework, and define rich notions of abstraction and refinement. In addition, they provide 'may' and 'must' modalities for transitions, and show how to perform efficient three-valued verification on such structures. They do not, however, provide an automated CEGAR framework, and it is not clear whether they have implemented and tested their approach.

Giannakopoulou and Magee [GM03] define 'fluent' propositions within a labeled transition systems context to express action-based linear-time properties. A fluent proposition is a property that holds after it is initiated by an action and ceases to hold when terminated by another action. This work exploits partial-order reduction techniques and has been implemented in the LTSA tool.

In a comparatively early paper, De Nicola *et al.* [NFGR93] propose a process algebraic framework with an action-based version of CTL as specification formalism. Verification then proceeds by first translating the underlying labeled transition systems (LTSs) of processes into Kripke structures and the action-based CTL specifications into equivalent state-based CTL formulas. At that point, a model checker is used to establish or refute the property.

Dill [Dil88] defines 'trace structures' as algebraic objects to model both hardware circuits and their specifications. Trace structures can handle equally well states or events, although usually not both at the same time. Dill's approach to verification is based on abstractions and compositional reasoning, albeit without an iterative counterexample-driven refinement loop.

In general, events (input signals) in circuits can be encoded via changes in state variables. Browne makes use of this idea in [Bro89], which features a CTL* specification formalism. Browne's framework also features abstractions and compositional reasoning, in a manner similar to Dill's.

Finally, Burch [Bur92] extends the idea of trace structures into a full-blown theory of 'trace algebra'. The focus here however is the modeling of discrete and continuous time, and the relationship between these two paradigms. This work also exploits abstractions and compositionality, however once again without automated counterexample-guided refinements.

3 Labeled Kripke Structures

A labeled Kripke structure (LKS for short) is a 7-tuple $(S, \mathit{Init}, P, \mathcal{L}, T, \Sigma, \mathcal{E})$ with S a finite set of *states*, $\mathit{Init} \subseteq S$ a set of initial states, P a finite set of *atomic state propositions*, $\mathcal{L} : S \to 2^P$ a *state-labeling function*, $T \subseteq S \times S$ a transition relation, Σ a finite set (*alphabet*) of *events* (or *actions*), and $\mathcal{E} : T \to (2^\Sigma \setminus \{\emptyset\})$ a *transition-labeling function*. We often write $s \xrightarrow{A} s'$ to mean that $(s, s') \in T$ and $A \subseteq \mathcal{E}(s, s')$.[2] In case A is a singleton set $\{a\}$ we write $s \xrightarrow{a} s'$ rather than $s \xrightarrow{\{a\}} s'$. Note that both states and transitions are 'labeled', the former with sets of atomic propositions, and the latter with non-empty sets of events. We further assume that our transition relation is *total* (every state has some successor), so that deadlock does not arise.

A *path* $\pi = \langle s_1, a_1, s_2, a_2, \dots \rangle$ of an LKS is an alternating infinite sequence of states and events subject to the following: for each $i \geqslant 1$, $s_i \in S$, $a_i \in \Sigma$, and $s_i \xrightarrow{a_i} s_{i+1}$.

The *language* of an LKS M, denoted $L(M)$, consists of the set of maximal paths of M whose first state lies in the set Init of initial states of M.

3.1 Abstraction

Let $M = (S, \mathit{Init}, P, \mathcal{L}, T, \Sigma, \mathcal{E})$ and $A = (S_A, \mathit{Init}_A, P_A, \mathcal{L}_A, T_A, \Sigma_A, \mathcal{E}_A)$ be two LKSs. We say that A is an *abstraction* of M, written $M \sqsubseteq A$, iff

[2] In keeping with standard mathematical practice, we write $\mathcal{E}(s, s')$ rather than the more cumbersome $\mathcal{E}((s, s'))$.

1. $P_A \subseteq P$,
2. $\Sigma_A = \Sigma$, and
3. For every path $\pi = \langle s_1, a_1, \ldots \rangle \in L(M)$ there exists a path $\pi' = \langle s'_1, a'_1, \ldots \rangle \in L(A)$ such that, for each $i \geqslant 1$, $a'_i = a_i$ and $\mathcal{L}_A(s'_i) = \mathcal{L}(s_i) \cap P_A$.

In other words, A is an abstraction of M if the 'propositional' language accepted by A contains the 'propositional' language of M, when restricted to the atomic propositions of A. This is similar to the well-known notion of 'existential abstraction' for Kripke structures in which certain variables are hidden [CGJ+00].

Two-way abstraction defines an equivalence relation $\sim$ on LKSs: $M \sim M'$ iff $M \sqsubseteq M'$ and $M' \sqsubseteq M$. We shall only be interested in LKSs up to $\sim$-equivalence.

3.2 Parallel Composition

The notion of parallel composition we consider in this paper allows for communication through shared actions only; in particular, we forbid the sharing of variables. This restriction facilitates the use of compositional reasoning in verifying specifications.

Let $M_1 = (S_1, \mathit{Init}_1, P_1, \mathcal{L}_1, T_1, \Sigma_1, \mathcal{E}_1)$ and $M_2 = (S_2, \mathit{Init}_2, P_2, \mathcal{L}_2, T_2, \Sigma_2, \mathcal{E}_2)$ be two LKSs. M_1 and M_2 are said to be *compatible* if (i) they do not share variables: $S_1 \cap S_2 = P_1 \cap P_2 = \emptyset$, and (ii) their parallel composition (as defined below) yields a total transition relation (so that no deadlock can occur). The parallel composition of M_1 and M_2 (assumed to be compatible)[3] is given by $M_1 \parallel M_2 = (S_1 \times S_2, \mathit{Init}_1 \times \mathit{Init}_2, P_1 \cup P_2, \mathcal{L}_1 \cup \mathcal{L}_2, T, \Sigma_1 \cup \Sigma_2, \mathcal{E})$, where $(\mathcal{L}_1 \cup \mathcal{L}_2)(s_1, s_2) = \mathcal{L}_1(s_1) \cup \mathcal{L}_2(s_2)$, and T and $\mathcal{E}$ are such that $(s_1, s_2) \xrightarrow{A} (s'_1, s'_2)$ iff $A \neq \emptyset$ and one of the following holds:

1. $A \subseteq \Sigma_1 \setminus \Sigma_2$ and $s_1 \xrightarrow{A} s'_1$ and $s'_2 = s_2$,
2. $A \subseteq \Sigma_2 \setminus \Sigma_1$ and $s_2 \xrightarrow{A} s'_2$ and $s'_1 = s_1$, and
3. $A \subseteq \Sigma_1 \cap \Sigma_2$ and $s_1 \xrightarrow{A} s'_1$ and $s_2 \xrightarrow{A} s'_2$.

In other words, components must synchronize on shared actions and proceed independently on local actions. Moreover, local variables are preserved by the respective states of each component. This notion of parallel composition is derived from CSP; see also [ACFM85].

Let M_1 and M_2 be as above, and let $\pi = \langle (s^1_1, s^2_1), a_1, \ldots \rangle$ be an alternating infinite sequence of states and events of $M_1 \parallel M_2$. The *projection* $\pi \lceil M_i$ of π on M_i consists of the (possibly finite) subsequence of $\langle s^i_1, a_1, \ldots \rangle$ obtained by simply removing all pairs $\langle a_j, s^i_{j+1} \rangle$ for which $a_j \notin \Sigma_i$. In other words, we keep from π only those states that belong to M_i, and excise any transition labeled with an event not in M_i's alphabet.

We now record the following theorem, which extends similar standard results for the process algebra CSP (for related proofs, we refer the reader to [Ros97]).

[3] The assumption of deadlock-freedom greatly simplifies our exposition, and also enables us to use a wider class of abstractions. At the moment, the onus is on the user to ensure that all LKSs to be composed in parallel are compatible. In the future, we plan to incorporate an optional deadlock-freedom checker within MAGIC.

Theorem 1.

1. *Parallel composition is (well-defined and) associative and commutative up to $\sim$-equivalence. Thus, in particular, no bracketing is required when combining more than two LKSs.*
2. *Let $M_1, \ldots, M_n$ be compatible LKSs, and let $A_1, \ldots, A_n$ be respective abstractions of the M_i: for each i, $M_i \sqsubseteq A_i$. Then $M_1 \parallel \ldots \parallel M_n \sqsubseteq A_1 \parallel \ldots \parallel A_n$. In other words, parallel composition preserves the abstraction relation.*
3. *Let $M_1, \ldots, M_n$ be compatible LKSs with respective alphabets $\Sigma_1, \ldots, \Sigma_n$, and let π be an infinite alternating sequence of states and events of the composition $M_1 \parallel \ldots \parallel M_n$. Then $\pi \in L(M_1 \parallel \ldots \parallel M_n)$ iff, for each i, there exists $\pi'_i \in L(M_i)$ such that $\pi \lceil M_i$ is a prefix[4] of π'_i. In other words, whether a path belongs to the language of a parallel composition of LKSs can be checked by projecting and examining the path on each individual component separately.*

Theorem 1 forms the basis of our compositional approach to verification: abstraction, counterexample validation, and refinement can all be done componentwise.

4 State/Event Linear Temporal Logic

We now present a logic enabling us to refer easily to both states and events when constructing specifications.

Given an LKS $M = (S, Init, P, \mathcal{L}, T, \Sigma, \mathcal{E})$, we consider linear temporal logic *state/event formulas* over the sets P and Σ (here p ranges over P and a ranges over Σ):

$$\phi ::= p \mid a \mid \neg\phi \mid \phi \wedge \phi \mid \mathbf{X}\phi \mid \mathbf{G}\phi \mid \mathbf{F}\phi \mid \phi\ \mathbf{U}\ \phi.$$

We write SE-LTL to denote the resulting logic, and in particular to distinguish it from (standard) LTL.

Let $\pi = \langle s_1, a_1, s_2, a_2, \ldots \rangle$ be a path. π^i stands for the suffix of π starting in state s_i. We then inductively define path-satisfaction of SE-LTL formulas as follows:

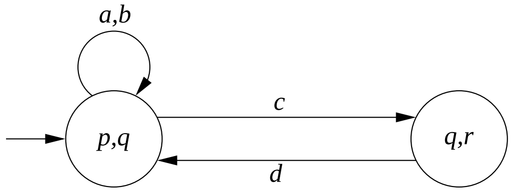
1. $\pi \models p$ iff s_1 is the first state of π and $p \in \mathcal{L}(s_1)$,
2. $\pi \models a$ iff a is the first event of π,
3. $\pi \models \neg\phi$ iff $\pi \not\models \phi$,
4. $\pi \models \phi_1 \wedge \phi_2$ iff $\pi \models \phi_1$ and $\pi \models \phi_2$,
5. $\pi \models \mathbf{X}\phi$ iff $\pi^2 \models \phi$,
6. $\pi \models \mathbf{G}\phi$ iff, for all $i \geqslant 1$, $\pi^i \models \phi$,
7. $\pi \models \mathbf{F}\phi$ iff, for some $i \geqslant 1$, $\pi^i \models \phi$, and
8. $\pi \models \phi_1\ \mathbf{U}\ \phi_2$ iff there is some $i \geqslant 1$ such that $\pi^i \models \phi_2$ and, for all $1 \leqslant j \leqslant i-1$, $\pi^j \models \phi_1$.

[4] By convention, an infinite sequence is prefix of another one iff they are the same.

We then let $M \models \phi$ iff, for every path $\pi \in L(M)$, $\pi \models \phi$.

We also use the derived $\mathbf{W}$ operator: $\phi_1 \mathbf{W} \phi_2$ iff $(\mathbf{G}\phi_1) \vee (\phi_1 \mathbf{U} \phi_2)$, as well as standard boolean connectives such as $\rightarrow$, etc.

As a simple example, consider the following LKS M. It has two states, the leftmost of which is the sole initial state. Its set of atomic state propositions is $\{p, q, r\}$; the first state is labeled with $\{p, q\}$ and the second with $\{q, r\}$. M's transitions are similarly labeled with sets of events drawn from the alphabet $\{a, b, c, d\}$.



As the reader may easily verify, $M \models \mathbf{G}(c \rightarrow \mathbf{F}r)$ but $M \not\models \mathbf{G}(b \rightarrow \mathbf{F}r)$. Note also that $M \models \mathbf{G}(d \rightarrow \mathbf{F}r)$, but $M \not\models \mathbf{G}(d \rightarrow \mathbf{XF}r)$.

4.1 Automata-based Verification

We aim to reduce SE-LTL verification problems to standard automata-theoretic techniques for LTL. Note that a standard—but unsatisfactory—way of achieving this is to explicitly encode actions through changes in (additional) state variables, and then proceed with LTL verification. Unfortunately, this trick usually leads to a significant blow-up in the state space, and consequently yields much larger verification times. The approach we present here, on the other hand, does *not* alter the size of the LKS, and is therefore considerably more efficient.

We first recall some basic results about LTL, Kripke structures, and automata-based verification.

A Kripke structure is simply an LKS minus the alphabet and the transition-labeling function; as for LKSs, the transition relation of a Kripke structure is required to be total. An LTL formula is an SE-LTL formula which makes no use of events as atomic propositions.

For P a set of atomic propositions, let $\mathbf{B}_P \cong 2^{2^P}$ denote the set of boolean combinations of atomic propositions over P.

A Büchi automaton is a 6-tuple $B = (S_B, Init_B, P, \mathcal{L}_B, T_B, Acc)$ with S_B a finite set of states, $Init \subseteq S_B$ a set of initial states, P a finite set of atomic state propositions, $\mathcal{L}_B : S_B \rightarrow \mathbf{B}_P$ a state-labeling function, $T \subseteq S_B \times S_B$ a transition relation, and $Acc \subseteq S_B$ a set of *accepting* states.

Note that the transition relation is not required to be total, and is moreover unlabeled. Note also that the states of a Büchi automaton are labeled with arbitrary boolean combinations of atomic propositions.

For π an infinite sequence of states of a Büchi automaton, let $\mathsf{inf}(\pi) \subseteq S_B$ be the set of states which occur infinitely often in π. π is *accepted* by the Büchi automaton B if $\mathsf{inf}(\pi) \cap Acc \neq \emptyset$. The set of all such accepted paths is written $L(B)$.

Let $M = (S, Init, P, \mathcal{L}, T)$ be a Kripke structure. The state-labeling function $\mathcal{L} : S \to 2^P$ indicates, for each state $s \in S$, exactly which atomic propositions hold at s; such labeling is equivalent to asserting that the compound proposition $\bigwedge \mathcal{L}(s) \wedge \bigwedge \{\neg p | p \in P \setminus \mathcal{L}(s)\}$ holds at s. Let us denote this compound proposition by $\widetilde{\mathcal{L}(s)}$. Every Kripke structure can therefore be viewed as a Büchi automaton, where we consider every state to be accepting.

Let $B = (S_B, Init_B, P, \mathcal{L}_B, T_B, Acc)$ be a Büchi automaton over the same set of atomic propositions as M. We can define the 'standard' product $M \times B = (S', Init', -, -, T', Acc')$ as a product of Büchi automata. More precisely,

1. $S' = \{(s,b) \in S \times S_B \mid \widetilde{\mathcal{L}(s)}$ implies $\mathcal{L}_B(b)\}$,
2. $(s,b) \longrightarrow (s',b')$ iff $s \longrightarrow s'$ and $b \longrightarrow b'$,
3. $(s,b) \in Init'$ iff $s \in Init$ and $b \in Init'$, and
4. $(s,b) \in Acc'$ iff $b \in Acc$.

The non-symmetrical standard product $M \times B$ accepts exactly those paths of M which are 'consistent' with B. Its main technical use lies in the following result of Gerth *et al.* [GPVW95]:

Theorem 2. *Given a Kripke structure M and LTL formula ϕ, there is a Büchi automaton $B_{\neg\phi}$ such that*

$$M \models \phi \text{ iff } L(M \times B_{\neg\phi}) = \emptyset.$$

An efficient tool to convert LTL formulas into optimized Büchi automata with the above property is Somenzi and Bloem's Wring [Wri, SB00].

We now turn to labeled Kripke structures. Let $M = (S, Init, P, \mathcal{L}, T, \Sigma, \mathcal{E})$ be an LKS. Recall that SE-LTL formulas allow events in Σ to stand for atomic propositions. For $x \in \Sigma$, let us therefore write $\widetilde{x}$ to denote the (formal) compound proposition $x \wedge \bigwedge\{\neg y \mid y \in \Sigma \setminus \{x\}\}$. We can also, given an SE-LTL formula ϕ over P and Σ, interpret ϕ as an *LTL* formula over $P \cup \Sigma$ (viewed as atomic *state* propositions); let us denote the latter formula by $\phi^\flat$. $\phi^\flat$ is therefore syntactically identical to ϕ, but differs from ϕ in its semantic interpretation.

We now define the *state/event product* of a labeled Kripke structure with a Büchi automaton. Let M be as above, and let $B = (S_B, Init_B, P \cup \Sigma, \mathcal{L}_B, T_B, Acc)$ be a Büchi automaton over the set of atomic state propositions $P \cup \Sigma$. The state/event product $M \otimes B = (S', Init', -, -, T', Acc')$ is a Büchi automaton that satisfies

1. $S' = \{(s,b) \in S \times S_B \mid \widetilde{\mathcal{L}(s)}$ implies $\exists \Sigma \centerdot \mathcal{L}_B(b)\}$,[5]

[5] The term $\exists \Sigma \centerdot \mathcal{L}_B(b)$ denotes the formula $\mathcal{L}_B(b)$ in which all atomic Σ-propositions have been existentially quantified out; in practice, however, the output of Wring is presented in a format which renders this operation trivial (and computationally inexpensive).

2. $(s, b) \longrightarrow (s', b')$ iff there exists $x \in \Sigma$ such that $s \xrightarrow{x} s'$ and $b \longrightarrow b'$ and $(\widetilde{\mathcal{L}(s)} \wedge \widetilde{x})$ implies $\mathcal{L}_B(b)$,
3. $(s, b) \in \mathit{Init}'$ iff $s \in \mathit{Init}$ and $b \in \mathit{Init}_B$, and
4. $(s, b) \in \mathit{Acc}'$ iff $b \in \mathit{Acc}$.

Finally, we have:

Theorem 3. *For any LKS M and SE-LTL formula ϕ,*

$$M \vDash \phi \text{ iff } L(M \otimes B_{\neg\phi^\flat}) = \emptyset.$$

Note that the state/event product does *not* require an enlargement of the LKS M (although we consider below just such an enlargement in the course of the proof of the theorem).

Proof. Observe that a state of M can have several differently-labeled transitions emanating from it. However, by duplicating states (and transitions) as necessary, we can transform M into a $\sim$-equivalent LKS M' having the following property: for every state s of M', the transitions emanating from s are all labeled with the same (single) event. As a result, the validity of an SE-LTL atomic event proposition a in a given state of M' does not depend on the particular path to be taken from that state, and can therefore be recorded as a propositional state variable of the state itself. Formally, this gives rise to a Kripke structure M'' over atomic state propositions $P \cup \Sigma$.

We now claim that

$$L(M \otimes B_{\neg\phi^\flat}) = \emptyset \text{ iff } L(M'' \times B_{\neg\phi^\flat}) = \emptyset. \tag{1}$$

To see this, notice first that there is a bijection between $L(M)$ and $L(M'')$ (which we denote $\pi \mapsto \pi''$). Next, observe that any path in $L(M \otimes B_{\neg\phi^\flat})$ can be decomposed as a pair (π, β), where $\pi \in L(M)$ and $\beta \in L(B_{\neg\phi^\flat})$; likewise, any path in $L(M'' \times B_{\neg\phi^\flat})$ can be decomposed as a pair (π'', β), where $\pi'' \in L(M'')$ and $\beta \in L(B_{\neg\phi^\flat})$. A straightforward inspection of the relevant definitions then reveals that $(\pi, \beta) \in L(M \otimes B_{\neg\phi^\flat})$ iff $(\pi'', \beta) \in L(M'' \times B_{\neg\phi^\flat})$, which establishes our claim.

Finally, we clearly have $M \vDash \phi$ iff $M' \vDash \phi$ iff $M'' \vDash \phi^\flat$. Combining this with Theorem 2 and Equation 1 above, we get $M \vDash \phi$ iff $L(M \otimes B_{\neg\phi^\flat}) = \emptyset$, as required. $\square$

The significance of Theorem 3 is that it enables us to make use of the highly optimized algorithms and tools available for verifying LTL formulas on Kripke structures to verify *SE-LTL* specifications on *labeled* Kripke structures, at no additional cost.

5 A Surge Protector

We describe a safety-critical current surge protector in order to illustrate the advantages of state/event-based implementations and specifications over both the pure state-based and the pure event-based approaches.

The surge protector is meant at all times to disallow changes in current beyond a varying threshold. The labeled Kripke structure in Figure 1 captures the main functional aspects of such a protector in which the possible values of the current and threshold are 0, 1, and 2. The threshold value is stored in the variable m, and changes in threshold and current are respectively communicated via the events $m0$, $m1$, $m2$, and $c0$, $c1$, $c2$.[6] Note, for instance, that when $m = 1$ the protector accepts changes in current to values 0 and 1, but not 2 (in practice, an attempt to hike the current up to 2 should trigger, say, a fuse and a jump to an emergency state, behaviors which are here abstracted away).

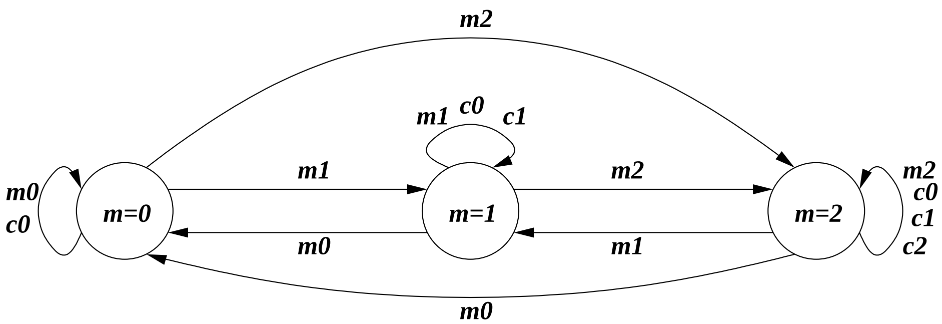


Fig. 1. The LKS of a surge protector

The required specification is neatly captured as the following SE-LTL formula:

$$\phi_{\mathrm{se}} = \mathbf{G}((c2 \rightarrow m = 2) \wedge (c1 \rightarrow (m = 1 \vee m = 2))).$$

By way of comparison, Figure 2 represents the (event-free) Kripke structure that captures the same behavior as the LKS of Figure 1. In this pure state-based formalism, nine states are required to capture all the reachable combinations of threshold ($m = i$) and last current changes ($c = j$) values.

The data (9 states and 39 transitions) compares unfavorably with that of the LKS in Figure 1 (3 states and 9 transitions). Moreover, as the allowable current ranges increase, the number of states of the LKS will grow linearly, as opposed to quadratically for the Kripke structure. The number of transitions of both will grow quadratically, but with a roughly four-fold larger factor for the Kripke structure. These observations highlight the advantages of a state/event approach, which of course will be more or less pronounced depending on the type of system under consideration.

[6] The reader may object that we have only allowed for *boolean* variables in our definition of labeled Kripke structures; it is however trivial to implement more complex types, such as bounded integers, as boolean encodings, and we have therefore elided such details here.

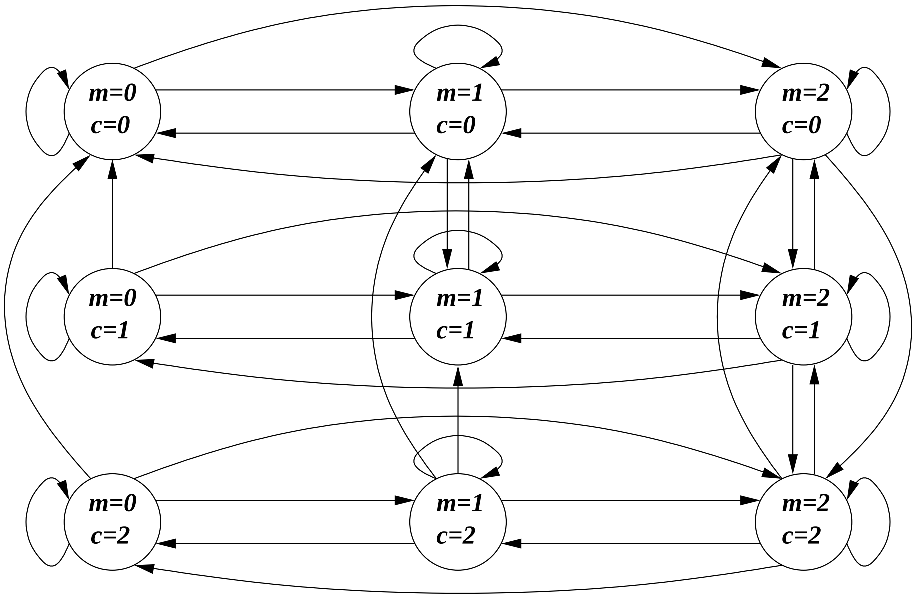


Fig. 2. The Kripke structure of a surge protector

Another advantage of the state/event approach is witnessed when one tries to write down specifications. In this instance, the specification we require is

$$\begin{aligned}\phi_{\mathrm{s}} = {} & \mathbf{G}(((c = 0 \vee c = 2) \wedge \mathbf{X}(c = 1)) \rightarrow (m = 1 \vee m = 2)) \wedge \\ & \mathbf{G}(((c = 0 \vee c = 1) \wedge \mathbf{X}(c = 2)) \rightarrow m = 2),\end{aligned}$$

which is arguably significantly more complex than ϕ_{se}.

The pure event-based specification ϕ_{e} capturing the same requirement is also clearly more complex than ϕ_{se}:

$$\begin{aligned}\phi_{\mathrm{e}} = {} & \mathbf{G}(m0 \rightarrow ((\neg c1)\ \mathbf{W}\ (m1 \vee m2))) \wedge \\ & \mathbf{G}(m0 \rightarrow ((\neg c2)\ \mathbf{W}\ m2)) \wedge \\ & \mathbf{G}(m1 \rightarrow ((\neg c2)\ \mathbf{W}\ m2)).\end{aligned}$$

The greater simplicity of the implementation and specification associated with the state/event formalism is not purely a matter of aesthetics, or even a safeguard against subtle mistakes; experiments also suggest that the state/event formulation yields significant gains in both time and memory during verification. We implemented three parameterized instances of the surge protector as simple C programs, in one case allowing message passing (representing the LKS), and in the other relying solely on local variables (representing the Kripke structure). We also wrote corresponding specifications respectively as SE-LTL and LTL formulas (as above) and converted these into Büchi automata using the tool Wring [Wri].

Figure 3 records the number of Büchi states and transitions associated with the specification, as well as the time taken by MAGIC to construct the Büchi automaton and confirm that the corresponding implementation indeed meets the specification.

Range	Pure State				Pure Event				State/Event			
	St	Tr	B-T	T-T	St	Tr	B-T	T-T	St	Tr	B-T	T-T
2	4	5	253	383	6	10	245	320	3	4	184	252
3	8	12	270	545	12	23	560	674	4	6	298	407
4	14	23	492	1141	20	41	1597	1770	5	8	243	391
5	22	38	1056	2326	30	64	3795	4104	6	10	306	497
6	32	57	2428	4818	42	92	12077	12660	7	12	614	962
7	44	80	6249	10358	56	125	54208	55064	8	14	930	1321
8	58	107	17503	24603	72	163	372784	374166	9	16	2622	3133
9	74	138	55950	67553	*	*	*	*	10	18	8750	9488
10	92	173	195718	213969	*	*	*	*	11	20	33556	34503
11	*	*	*	*	*	*	*	*	12	22	135252	136500
12	*	*	*	*	*	*	*	*	13	24	534914	536451
13	*	*	*	*	*	*	*	*	*	*	*	*

Fig. 3. Comparison of pure state-based, pure event-based and state/event-based formalisms. Values of c and m range between 0 and **Range**. **St** and **Tr** respectively denote the number of states and transitions of the Büchi automaton corresponding to the specification. **B-T** is the Büchi construction time and **T-T** is the total verification time. All times are reported in milliseconds. A $*$ indicates that the Büchi automaton construction did not terminate in 10 minutes.

A careful inspection of the table in Figure 3 reveals several consistent trends. First, the number of Büchi states increases quadratically with the value of *Range* for both the pure state-based and pure event-based formalisms. In contrast, the increase is only linear when both states and events are used. We notice a similar pattern among the number of transitions in the Büchi automata. The rapid increase in the sizes of Büchi automata will naturally contribute to increased model checking time. However, we notice that the major portion of the total verification time is required to construct the Büchi automaton. While this time increases rapidly in all three formalisms, the growth is observed to be most benign for the state/event scenario. The net result is clearly evident from Figure 3. Using both states and events allows us to push the limits of c and m beyond what is possible by using either states or events alone.

6 Compositional Counterexample-Guided Verification

We now discuss how our framework enables us to verify SE-LTL specifications on parallel compositions of labeled Kripke structures incrementally and compositionally.

When trying to determine whether an SE-LTL specification holds on a given LKS, the following result is the key ingredient needed to exploit abstractions in the verification process:

Theorem 4. *Let M and A be LKSs with $M \sqsubseteq A$. Then for any SE-LTL formula ϕ over M which mentions only propositions (and events) of A,*

$$\textit{if } A \vDash \phi \textit{ then } M \vDash \phi.$$

Proof. This follows easily from the fact that every path of M is matched by a corresponding property-preserving path of A. □

Let us now assume that we are given a collection $M_1, \ldots, M_n$ of LKSs, as well as an SE-LTL specification ϕ, with the task of determining whether $M_1 \parallel \ldots \parallel M_n \vDash \phi$. We first create initial abstractions $A_1 \sqsupseteq M_1, \ldots, A_n \sqsupseteq M_n$, in a manner to be discussed shortly. We then check whether $A_1 \parallel \ldots \parallel A_n \vDash \phi$. In the affirmative, we conclude (by Theorems 1 and 4) that $M_1 \parallel \ldots \parallel M_n \vDash \phi$ as well. In the negative, we are provided with a counterexample $\pi_A \in L(A_1 \parallel \ldots \parallel A_n)$ such that $\pi_A \nvDash \phi$. We must then determine whether this counterexample is real or spurious, i.e., whether it corresponds to a counterexample $\pi \in L(M_1 \parallel \ldots \parallel M_n)$.

This validation check can be performed compositionally, as follows. According to Theorem 1, the counterexample is real iff for each i, the projection $\pi_A \restriction A_i$ corresponds to (the prefix of) a valid behavior of M_i. To this end, we 'simulate' $\pi_A \restriction A_i$ on M_i. If M_i accepts the path, we go on to the next component. Otherwise, we *refine* our abstraction A_i, yielding a new abstraction A_i' with $M_i \sqsubseteq A_i' \sqsubseteq A_i$ and such that A_i' also rejects the projection $\pi_A \restriction A_i'$ of the spurious counterexample π_A.

This process is iterated until either the specification is proved, or a real counterexample is found. Termination follows from the fact that the LKSs involved are all finite, and therefore admit only finitely many distinct abstractions.[7]

The advantage of this approach is that all the abstractions that we consider in this paper are *existential abstraction quotients* of LKSs. In other words, abstractions are obtained by lumping together states of the original LKSs, and have therefore smaller state spaces. Formally, given an LKS $M = (S, \mathit{Init}, P, \mathcal{L}, T, \Sigma, \mathcal{E})$ and a partition $\approx$ of the state space S, an *existential abstraction quotient* of M is any LKS $A = (S_A, \mathit{Init}_A, P_A, \mathcal{L}_A, T_A, \Sigma_A, \mathcal{E}_A)$ such that

1. $S_A = S/\!\approx$,
2. $\mathit{Init}_A = \{[s] \in S/\!\approx \mid \exists s' \in [s] \centerdot s' \in \mathit{Init}\}$,
3. $P_A \subseteq P$, and for all $s, s' \in S$, if $s \approx s'$ then $\mathcal{L}(s) \cap P_A = \mathcal{L}(s') \cap P_A$,
4. for all $s \in S, \mathcal{L}_A([s]) = \mathcal{L}(s) \cap P_A$,
5. $\Sigma_A = \Sigma$, and

[7] When the LKSs $M_1, \ldots, M_n$ are generated from C programs via predicate abstraction, as is the case for MAGIC, termination will depend on whether MAGIC is eventually able to generate sufficiently strong predicates. Although this in general cannot be guaranteed, as a result of the undecidability of the halting problem, in practice it has not been observed to cause any problems.

6. for all $s, s' \in S$ and $a \in \Sigma$, $[s_1] \xrightarrow{a} [s_2]$ iff there exists $s_1' \in [s_1]$, $s_2' \in [s_2]$ such that $s_1' \xrightarrow{a} s_2'$.

We now have:

Theorem 5. *For M an LKS and $\approx$ a partition of S, any existential abstraction quotient A of M as defined above is a genuine abstraction of M in the sense of Section 3.1: $M \sqsubseteq A$.*

Note that an abstraction of M is entirely determined by the partition $\approx$ and the set P_A of atomic state propositions. In our case, given an SE-LTL formula ϕ, we shall fix P_A to be $\mathsf{stateprop}(\phi)$, the set of all atomic state propositions appearing in ϕ. Abstractions of M can therefore be identified with partitions $\approx$ of S that meet condition 3 above; we denote the corresponding abstraction by $M/\!\approx$.

Theorem 6. *Let M be an LKS and let $M/\!\approx$ be an abstraction of M. For any refinement $\approx'$ of the partition $\approx$, $M/\!\approx'$ is an abstraction of M that is also a refinement of $M/\!\approx$: $M \sqsubseteq M/\!\approx' \sqsubseteq M/\!\approx$.*

We leave the proofs of Theorems 5 and 6 to the reader.

To define the initial abstraction $M/\!\approx_1$, we let $s \approx_1 s'$ iff $\mathcal{L}(s) \cap \mathsf{stateprop}(\phi) = \mathcal{L}(s') \cap \mathsf{stateprop}(\phi)$ and $\mathsf{enabled}(s) = \mathsf{enabled}(s')$, where $\mathsf{enabled}(s)$ denotes the set of actions that appear in transitions originating from s, etc.

We must refine our abstraction whenever we encounter a spurious counterexample $\pi_{A_k} \in L(M/\!\approx_k) \setminus L(M)$. We achieve this, in fully automated fashion, by constructing a refinement $\approx_{k+1}$ of the partition $\approx_k$ which splits abstract states along the path π_{A_k}. The approach we take is very similar to that presented in [COYC03]; unfortunately, the details involved are too lengthy to reproduce here, and we refer the reader to that paper for a thorough account of the technique.

As discussed above, MAGIC iterates this abstraction-validation-refinement procedure component-wise until the property ϕ of interest is either established ($M_1 || \ldots || M_n \models \phi$) or refuted ($M_1 || \ldots || M_n \not\models \phi$).

7 Experimental Results

We experimented with two broad sets of benchmarks. All our experiments were performed on an AMD Athlon XP 1600+ machine with 900 MB RAM running RedHat Linux 7.1. The first set of our examples were based on OpenSSL-0.9.6c, an open-source implementation of the SSL protocol. This is a popular protocol used for secure exchange of sensitive information over untrusted networks. SSL involves an initial handshake between a client and a server that attempt to establish a secure channel between themselves. The target of our verification process was the implementation of this handshake, comprising of about 350 lines of ANSI C code each for the server and the client.

From the official SSL specification [SSL] we derived a set of nine properties that every correct SSL implementation should satisfy. The first five properties are

Name	St(B)	Tr(B)	St(Mdl)	T(BA)	T(Mdl)	T(Ver)	T(Total)	Mem
srvr-1-ss	4	5	5951	213	32195	1654	34090	-
srvr-1-se	3	4	4269	209	18116	1349	19674	-
srvr-2-ss	11	23	4941	292	31331	2479	34102	-
srvr-2-se	3	4	4269	196	17897	1317	19410	-
srvr-3-ss	37	149	5065	1147	26958	4031	32137	-
srvr-3-se	3	4	4269	462	17950	1908	20319	-
srvr-4-ss	16	41	5446	806	29809	7382	39341	28.6
srvr-4-se	7	14	4333	415	21453	3513	25906	24.1
srvr-5-ss	25	47	7951	690	48810	6842	56888	39.3
srvr-5-se	20	45	4331	497	18808	2925	22765	24.2
clnt-1-ss	16	41	4867	793	24488	1235	26953	25.8
clnt-1-se	7	14	3693	376	17250	583	18683	22.1
clnt-2-ss	25	47	7574	699	43592	1649	46444	38.1
clnt-2-se	18	40	3691	407	15304	1087	17269	21.2
ssl-1-ss	25	47	24799528	874	65585	*	*	850.5
ssl-1-se	20	45	13558984	655	33091	2172139	2206983	162.4
ssl-2-ss	25	47	32597042	836	66029	*	*	346.6
ssl-2-se	18	40	15911791	713	34641	4148550	4185068	320.7
UCOS-BUG	8	14	873	205	3409	261	3880	-
UCOS-1	8	14	873	194	3365	2797	6357	-
UCOS-2	5	8	873	123	3372	2630	6127	-

Fig. 4. Experimental results with OpenSSL and Micro-C OS. **St(B)** and **Tr(B)** = respectively the number of states and transitions in the Büchi automaton; **St(Mdl)** = number of states in the model; **T(Mdl)** = model construction time; **T(BA)** = Büchi construction time; **T(Ver)** = model checking time; **T(Total)** = total verification time. All reported times are in milliseconds. **Mem** is the total memory requirement in MB. A * indicates that the model checking did not terminate within 2 hours and was aborted. In such cases, other measurements were made at the point of forced termination. A - indicates that the corresponding measurement was not taken.

relevant only to the server, the next two apply only to the client, and the last two properties refer to both a server and a client executing concurrently. For instance, the first property states that whenever the server asks the client to terminate the handshake, it eventually either gets a correct response from the client or exits with an error code. The second property expresses the fact that whenever the server receives a handshake request from a client, it eventually acknowledges the request or returns with an error code. The third property states that a server never exchanges encryption keys with a client once the cipher scheme has been changed.

Each of these properties were then expressed in SE-LTL, once using only states and again using both states and events. Table 4 summarizes the results of our experiments with these benchmarks. The SSL benchmarks have names of the form x-y-z where x denotes the type of the property and can be either srvr, clnt or ssl, depending on whether the property refers respectively to only the server, only the client, or both server and client. y denotes the property number while z denotes the specification style and can be either ss (only states) or se (both states and events). We note that in each case the numbers for state/event properties are considerably better than those for the corresponding pure-state properties.

The second set of our benchmarks were obtained from the source code of version 2.00 of Micro-C OS. This is a popular, lightweight, real-time, multi-tasking operating system written in about 3000 lines of ANSI C. The OS uses a lock to ensure mutual exclusion for critical section code. Using SE-LTL we expressed two properties of the OS: (i) the lock is acquired and released alternately starting with an acquire and (ii) every time the lock is acquired it is eventually released. These properties were expressed using only events.

We found a **bug** in the OS that causes it to violate the first property. We informed the developers of the OS about this bug and were told that it has been detected and fixed. The developers also kindly supplied us with the latest source code for the OS, and we are currently attempting to find errors in it. The second property was found to be valid. In Figure 4 these experiments are named UCOS-BUG and UCOS-2 respectively. Next we fixed the bug and verified that the first property holds for the corrected OS. This experiment is called UCOS-1 in Figure 4.

8 Conclusion and Future Work

In this paper, we have presented an expressive framework for modeling and verifying linear-time temporal specifications on concurrent software systems. Our approach involves both states and events, and is predicated on a compositional counterexample-guided abstraction refinement scheme. We have also shown how standard automata-theoretic techniques for verifying linear temporal logic formulas can be ported to our framework at no extra cost, and have implemented these on our C model checker MAGIC. We have also carried out a number of experiments on the source code for OpenSSL-0.9.6c and Micro-C OS version 2, discovering a bug in the latter. These experiments have led us to conclude that not only does a state/event formalism facilitate the formulation of appropriate specifications (as compared to standard pure state-based or event-based frameworks), but also yield significant improvements in both verification time and memory usage.

There remain many avenues for further research. One is to consider alternative specification formalisms, such branching-time temporal logics. In our current framework, it may be possible to further optimize the automata-theoretic part of the verification, by directly transforming SE-LTL formulas into *labeled* Büchi automata. Doing so should yield more compact automata-based representations of specifications, resulting in a smaller overall state space. Another direction is to investigate other, more aggressive (and perhaps specification-dependent), notions of abstraction. We are also currently working on a compositional CEGAR-based algorithm to check deadlock-freedom.

MAGIC is at present an explicit model checking tool—it could be worthwhile to incorporate symbolic and partial order techniques to improve its efficiency further. Another interesting area of research is to develop mechanisms to handle shared variables. Other modifications under consideration include the modeling of fairness constraints. Lastly, we are currently attempting to model and verify the source code of a controller for a large industrial metal-casting plant.

References

[ACFM85] T. S. Anantharaman, E. M. Clarke, M. J. Foster, and B. Mishra. Compiling path expressions into VLSI circuits. In *Proceedings of POPL*, pages 191–204, 1985.

[BLA] BLAST website. `http://www-cad.eecs.berkeley.edu/~rupak/blast`.

[BLO98] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In *Proceedings of CAV*, volume 1427, pages 319–331. Springer LNCS, 1998.

[BMMR01] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 203–213, 2001.

[BR01] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of SPIN*, volume 2057, pages 103–122. Springer LNCS, 2001.

[Bro89] M. C. Browne. *Automatic verification of finite state machines using temporal logic.* PhD thesis, Carnegie Mellon University, 1989. Technical report no. CMU-CS-89-117.

[BS01] J. Bradfield and C. Stirling. *Modal Logics and Mu-Calculi : An Introduction*, pages 293–330. Handbook of Process Algebra. Elsevier, 2001.

[Bur92] J. Burch. *Trace algebra for automatic verification of real-time concurrent systems.* PhD thesis, Carnegie Mellon University, 1992. Technical report no. CMU-CS-92-179.

[CCG+03] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *Proceedings of ICSE 2003*, pages 385–395, 2003.

[CCK+02] 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 *Proceedings of FMCAD*, pages 33–51, 2002.

[CDH+00] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *Proceedings of ICSE*, pages 439–448. IEEE Computer Society, 2000.

[CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. *Lecture Notes in Computer Science*, 131, 1981.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications.

ACM Transactions on Programming Languages and Systems, 8(2):244–263, 1986.

[CGJ+00] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.

[CGKS02] E. M. Clarke, A. Gupta, J. H. Kukula, and O. Shrichman. SAT based abstraction-refinement using ILP and machine learning techniques. In *Proceedings of CAV*, pages 265–279, 2002.

[CGP99] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, December 1999.

[CGP03] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. Learning assumptions for compositional verification. In *Proceedings of TACAS*, volume 2619, pages 331–346. Springer LNCS, 2003.

[COYC03] S. Chaki, J. Ouaknine, K. Yorav, and E. M. Clarke. Automated compositional abstraction refinement for concurrent C programs: A two-level approach. In *Proceedings of SoftMC 03*. ENTCS 89(3), 2003.

[Dil88] D. L. Dill. *Trace theory for automatic hierarchical verification of speed-independent circuits*. PhD thesis, Carnegie Mellon University, 1988. Technical report no. CMU-CS-88-119.

[GL94] O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Trans. on Programming Languages and Systems*, 16(3):843–871, 1994.

[GM03] D. Giannakopoulou and J. Magee. Fluent model checking for event-based systems. In *Proceedings of FSE*. ACM Press, 2003.

[GPVW95] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.

[HJMQ03] T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *Proceedings of CAV*, volume 2725. Springer LNCS, 2003.

[HJMS02] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of POPL*, pages 58–70, 2002.

[HJS01] M. Huth, R. Jagadeesan, and D. Schmidt. Modal transition systems: A foundation for three-valued program analysis. In *Proceedings of ESOP 01*. LNCS 2028, 2001.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[HQR00] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Decomposing refinement proofs using assume-guarantee reasoning. In *Proceedings of ICCAD*, pages 245–252. IEEE Computer Society Press, 2000.

[Koz83] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[Kur94] R. P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, 1994.

[KV98] E. Kindler and T. Vesper. ESTL: A temporal logic for events and states. In *Proceedings of ATPN 98*, pages 365–383. LNCS 1420, 1998.

[LBBO01] Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental verification by abstraction. In *Proceedings of TACAS*, volume 2031, pages 98–112. Springer LNCS, 2001.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of POPL*, 1985.

[MAG] MAGIC website. `http://www.cs.cmu.edu/~chaki/magic`.

[McM97] K. L. McMillan. A compositional rule for hardware design refinement. In *Proceedings of CAV*, volume 1254, pages 24–35. Springer LNCS, 1997.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall International, London, 1989.

[NCOD97] G. Naumovich, L. A. Clarke, L. J. Osterweil, and M. B. Dwyer. Verification of concurrent software with FLAVERS. In *Proceedings of ICSE*, pages 594–595. ACM Press, 1997.

[NFGR93] R. De Nicola, A. Fantechi, S. Gnesi, and G. Ristori. An action-based framework for verifying logical and behavioural properties of concurrent systems. *Computer Networks and ISDN Systems*, 25(7):761–778, 1993.

[NV95] R. De Nicola and F. Vaandrager. Three logics for branching bisimulation. *Journal of the ACM (JACM)*, 42(2):458–487, 1995.

[PDV01] C. S. Păsăreanu, M. B. Dwyer, and W. Visser. Finding feasible counter-examples when model checking abstracted Java programs. In *Proceedings of TACAS*, volume 2031, pages 284–298. Springer LNCS, 2001.

[Pnu86] A. Pnueli. Application of temporal logic to the specification and verification of reactive systems: A survey of current trends. In J.W. de Bakker, W. P. de Roever, and G. Rozenburg, editors, *Current Trends in Concurrency*, volume 224 of *Lecture Notes in Computer Science*, pages 510–584. Springer, 1986.

[QS81] J.P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *proceedings of Fifth Intern. Symposium on Programming*, pages 337–350, 1981.

[Ros97] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall International, London, 1997.

[SB00] F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In *Computer-Aided Verification*, pages 248–263, 2000.

[SLA] SLAM website. `http://research.microsoft.com/slam`.

[SSL] OpenSSL. `http://wp.netscape.com/eng/ssl3/ssl-toc.html`.

[Sto02] S. D. Stoller. Model-checking multi-threaded distributed Java programs. *International Journal on Software Tools for Technology Transfer*, 4(1):71–91, 2002.

[Wri] Wring website. `http://vlsi.colorado.edu/~rbloem/wring.html`.