State/Event-based Software Model Checking

Sagar Chaki 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 very powerful verification techniques, counterexample-guided abstraction refinement, and compositional reasoning. Our specification language is a state/event extension of linear-time temporal logic, and allows us to express many properties of software in a very concise and intuitive manner. We also show how standard automata-theoretic LTL verification techniques can be ported to our framework, and have implemented these within our C model checker, MAGIC. Preliminary investigations suggest that this new approach not only reduces human effort, but also boasts important gains both in space and in time.

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 the correctness of software.

In this endeavor, the well-known methodology of *model checking* [CE81, CES86, QS81, CGP99] has so far had very little success. This state of affairs is at first sight surprising, given the many well-documented happy applications of model checking in the hardware world. However, 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 perrenial state space explosion problem, whereby the complexity of verifying an implementation against a specification becomes prohibitive.

It has often been observed that, in attempting to address these issues, tradeoffs are inevitable. More expressive specification formalisms, such as the mu-calculus, tend to have costlier verification complexity, and vice-versa. In this paper, we propose an original combination of several approaches and methodologies to engineer a pleasantly expressive framework which nonetheless leaves us free to exploit some powerful state-space explosion-fighting tools, such as counterexample-guided abstraction refinement [CGJ+00] and compositional reasoning.

The most common instantiations of model checking to date have focused on finite-state models and either branching-time (CTL) or linear-time (LTL) temporal logics. Initially, model checking was primarily used to reason about the correctness of hardware designs and communication protocols. One of the major difficulties in applying model checking to software is the complexity of specifying temporal logic properties over the finite-state abstracted models of computer programs. This problem is even more pronounced when reasoning about concurrent software. Indeed, in concurrent programs, communication among modules proceeds via actions (or events), which can represent function calls, requests and acknowledgements, 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 labeled transition systems (LTS) using either *state-based* or *event-based* formalisms. 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. This approach is not practical however for large-scale software, and further worsens when actions are 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 modelling framework consists of *labeled Kripke structures* (LKS), which are LTSs in which in addition transitions are labeled with actions. The specification logic is a *state/event* derivative of LTL. This allows one to represent both software implementations and specifications directly without any program annotations, or privileged insight into the program execution. We further show that standard efficient LTL model checking algorithms can be applied to help reasoning about state/event-based systems. We have implemented our approach within the concurrent C verification tool MAGIC [CCG+03, COYC03], and report promising results in the preliminary 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. These procedures are embedded within a counterexample-guided abstraction refinement framework (CEGAR for short), 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. We contrast our approach with equivalent purely 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.

1.1 Related Work

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 comprehensively subsumed by the modal mu-calculus [Pnu86, BS01]. The novelty of our approach, however, is the way in which we efficiently integrate an expressive state/event formalism with very powerful verification techniques, namely CEGAR as well as 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 (ESTL) for Petri nets. Interestingly, they argue that purely state-based or event-based formalisms lack expressiveness in important respects. However it is not straightforward to compare their work to ours since the underlying foundations differ in very significant ways.

Huth et al. [MH01] also propose a state/event framework, and also 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.

1.2 Outline

The paper is organized as follows. Section 2 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 3, 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 4, we illustrate these ideas by modelling a simple surge protector. We also contrast our approach with purely 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 an important time and space penalty during verification.[1] Section 5 details our compositional CEGAR loop. Finally, Section 6 summarizes the contribution of the paper and outlines several avenues for future work.

2 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 also point out that, in contrast with many like formalisms, we do *not* require our transition relation T to be total, in order to be able to account for deadlock.[3]

A *finite path* $\pi = \langle s_1, a_1, s_2, a_2, s_3, \ldots, s_k \rangle$ of an LKS is a valid alternating finite sequence of states and events: for each $1 \leqslant i \leqslant k$, $s_i \in S$, $a_i \in \Sigma$, and (for $i < k$) $s_i \xrightarrow{a_i} s_{i+1}$. An *infinite path* is likewise defined to be a valid alternating infinite sequence of states and events. Lastly, a *maximal path* is either an infinite path, or a finite path which cannot be properly extended.

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.

2.1 Abstraction

Let $M = (S_M, \mathit{Init}_M, P_M, \mathcal{L}_M, T_M, \Sigma_M, \mathcal{E}_M)$ 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

1. $P_A \subseteq P_M$,
2. $\Sigma_A = \Sigma_M$,
3. For every path $\pi = \langle s_1, a_1, \ldots, s_k \rangle \in L(M)$ there exists a path $\pi' = \langle s'_1, a'_1, \ldots, s'_k \rangle \in L(A)$ such that, for each i, $a'_i = a_i$ and $\mathcal{L}_A(s'_i) = \mathcal{L}_M(s_i) \cap P_A$, and
4. Same as (3) for infinite paths.

[1] In order to invoke MAGIC, we code the LKSs and LTSs 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] In keeping with standard mathematical practice, we write $\mathcal{E}(s, s')$ rather than the more cumbersome $\mathcal{E}((s, s'))$.

[3] Deadlock is an inevitable consequence of our decision to handle synchronous parallel composition.

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 notion of abstraction generalizes a well-known 'existential abstraction' for Kripke structures. It is important to also note that, as defined here, abstractions preserve deadlocking behavior (thanks to clause (3)).

Two-way abstraction defines an equivalence relation $\sim$ on LKSs: $M \sim M'$ iff $M \sqsubseteq M'$ and $M' \sqsubseteq M$. We shall in general not wish to distinguish between $\sim$-equivalent LKSs.

2.2 Parallel Composition

As mentioned in the Introduction, the notion of parallel composition we consider here allows for communication through shared actions only; in particular, we forbid the sharing of variables. This restriction enables us to use compositional reasoning to verify 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 they do not share variables: $S_1 \cap S_2 = P_1 \cap P_2 = \emptyset$. The parallel composition of M_1 and M_2, defined only for compatible LKSs, is given by $M_1 || M_2 = (S_1 \times S_2, \mathit{Init}_1 \times \mathit{Init}_2, P_1 \cup P_2, \mathcal{L}_1 \cup \mathcal{L}_2, T, \Sigma_1 \cup \Sigma_2, \mathcal{E})$, 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 $s_1' = s_1$
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.

Let M_1 and M_2 be as above, and let $\pi = \langle (s_1^1, s_1^2), a_1, \ldots, (s_k^1, s_k^2) \rangle$ be an alternating finite sequence of states and events of $M_1 || M_2$. The projection $\pi \restriction M_i$ of π on M_i consists of the subsequence of $\langle s_1^i, a_1, \ldots, s_k^i \rangle$ obtained by simply removing all pairs $\langle a_j, s_{j+1}^i \rangle$ 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. For π an infinite path, the definition of $\pi \restriction M_i$ is adapted in the obvious way.

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

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 $M_1 || \ldots || M_n \sqsubseteq A_1 || \ldots || A_n$. 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 a (finite of infinite) alternating sequence of states and events of $M_1 || \ldots || M_n$. Then $\pi \in L(M_1 || \ldots || M_n)$ iff, for each i, there exists $\pi'_i \in L(M_i)$ such that $\pi \restriction M_i$ is a prefix of π'_i, and in addition π cannot properly be extended as the parallel composition of the 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.

3 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 $M = (S, \mathit{Init}, P, \mathcal{L}, T, \Sigma, \mathcal{E})$, 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 $\pi = \langle s_1, a_1, s_2, a_2, \ldots, s_k \rangle$ or $\pi = \langle s_1, a_1, s_2, a_2, \ldots \rangle$ be a finite or infinite path. We define π^i stand for the suffix of π starting in state s_i, with the convention that π^i denotes the empty sequence whenever i exceeds the largest index of π. We then inductively define path-satisfaction of SE-LTL formulas as follows:

1. $\pi \vDash p$ iff s_1 is the first state of π and $p \in \mathcal{L}(s_1)$.
2. $\pi \vDash a$ iff a is the first event of π.
3. $\pi \vDash \neg\phi$ iff $\pi \nvDash \phi$.
4. $\pi \vDash \phi_1 \wedge \phi_2$ iff $\pi \vDash \phi_1$ and $\pi \vDash \phi_2$.
5. $\pi \vDash \mathbf{X}\phi$ iff $\pi^2 \neq \langle\rangle$ and $\pi^2 \vDash \phi$.
6. $\pi \vDash \mathbf{G}\phi$ iff, for all $i \geqslant 1$, $\pi^i \vDash \phi$ or $\pi^i = \langle\rangle$.
7. $\pi \vDash \mathbf{F}\phi$ iff, for some $i \geqslant 1$, $\pi^i \neq \langle\rangle$ and $\pi^i \vDash \phi$.
8. $\pi \vDash \phi_1 \ \mathbf{U}\ \phi_2$ iff there is some $i \geqslant 1$ such that $\pi^i \neq \langle\rangle$, $\pi^i \vDash \phi_2$ and, for all $1 \leqslant j \leqslant i-1$, $\pi^j \vDash \phi_1$.

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

As a simple illustrating example, the reader may wish to verify that the SE-LTL formula **GXtrue** expresses deadlock-freedom (here **true** can be defined to be, say, $p \vee \neg p$, for any $p \in P$).

3.1 Automata-based Verification

We aim to reduce SE-LTL verification problems to standard automata-theoretic techniques for LTL. Since the latter usually deal exclusively with infinite paths, we must first perform some preliminary transformations. Given an LKS M, we construct an LKS $M_\perp$, which is identical to M, except that every deadlocked state s of M is given in $M_\perp$ a new transition $s \xrightarrow{!} \perp$, where ! is a new event and $\perp$ is a new state. Moreover, we postulate in $M_\perp$ a !-labelled self-transition. Lastly, $M_\perp$ is endowed with a single extra atomic proposition, dk, which uniquely labels $\perp$.

Given an SE-LTL formula ϕ over (the atomic propositions and alphabet of) M, define inductively a new SE-LTL formula $\phi_\perp$ over $M_\perp$, as follows:

1. $p_\perp = p$.
2. $a_\perp = a$.
3. $(\neg\phi)_\perp = \neg(\phi_\perp)$.
4. $(\phi_1 \wedge \phi_2)_\perp = {\phi_1}_\perp \wedge {\phi_2}_\perp$.
5. $(\mathbf{X}\phi)_\perp = \mathbf{X}(\phi_\perp \wedge \neg dk)$.
6. $(\mathbf{G}\phi)_\perp = \mathbf{G}(\phi_\perp \vee dk)$.
7. $(\mathbf{F}\phi)_\perp = \mathbf{F}(\phi_\perp \wedge \neg dk)$.
8. $(\phi_1 \;\mathbf{U}\; \phi_2)_\perp = {\phi_1}_\perp \;\mathbf{U}\; ({\phi_2}_\perp \wedge \neg dk)$.

We now have:

Lemma 2. *For any LKS M and SE-LTL formula ϕ over M,*

$$M \models \phi \text{ iff } M_\perp \models \phi_\perp.$$

Proof. Straightforward structural induction. □

We now recall some basic results about (standard) LTL, Kripke structures, and automata-based verification.

A Kripke structure is simply an LKS minus the alphabet and the transition-labeling function; in addition, 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.

Let $M = (S, \mathit{Init}, P, \mathcal{L}, T)$ be a Kripke structure, and let $B = (S_B, \mathit{Init}_B, P, \mathcal{L}_B, T_B, \mathit{Acc})$ be a Büchi automaton. Note that $\mathcal{L}_B$ takes states in S_B to arbitrary boolean combinations of atomic propositions in P (rather than merely conjunctions of such).

The Kripke structure M can be viewed as a Büchi automaton in which every state is Büchi-accepting. We can therefore define the product $M \times B = (S', \mathit{Init}', -, -, T', \mathit{Acc}')$ as a product of Büchi automata. More precisely,

1. $S' = \{(s,b) \,|\, \mathcal{L}(s) \text{ implies } \mathcal{L}_B(b)\}$, where $\mathcal{L}(s)$ is interpreted as a conjunction of atomic propositions,
2. $(s,b) \longrightarrow (s',b')$ iff $s \longrightarrow s'$ and $b \longrightarrow b'$,
3. $(s,b) \in \mathit{Init}'$ iff $s \in \mathit{Init}$ and $b \in \mathit{Init}'$, and

4. $(s, b) \in Acc'$ iff $b \in Acc$.

The main technical tool is the following result of Gerth et al. [GPVW95]:

Theorem 3. *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 define the product of a labeled Kripke structure with a Büchi automaton. Let $M = (S, \mathit{Init}, P, \mathcal{L}, T, \Sigma, \mathcal{E})$ be an LKS, and let $(S_B, \mathit{Init}_B, P, \mathcal{L}_B, T_B, \mathit{Acc})$ be a Büchi automaton. Their product $M \otimes B = (S', \mathit{Init}', -, -, T', \mathit{Acc}')$ is a Büchi automaton that satisfies

1. $S' = \{(s, b) \mid \mathcal{L}(s) \text{ implies } \mathcal{L}_B(b)\}$, where $\mathcal{L}(s)$ is interpreted as a conjunction of atomic propositions,
2. $(s, b) \longrightarrow (s', b')$ iff there exists $x \in \Sigma$ such that $s \xrightarrow{x} s'$ and x implies $\mathcal{L}_B(b)$ (where the event x is viewed as an atomic SE-LTL proposition),
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 4. *For any LKS M and SE-LTL formula ϕ,*

$$M \models \phi \text{ iff } L(M_\perp \otimes B_{\neg(\phi_\perp)}) = \emptyset.$$

Proof. From Lemma 2 we know that $M \models \phi$ iff $M_\perp \models \phi_\perp$. Now observe that a state of $M_\perp$ can have several differently-labeled transitions emanating from it. However, by duplicating states (and transitions) as necessary, we can transform $M_\perp$ into a $\sim$-equivalent LKS $M'_\perp$ having the following property: for every state s of $M'_\perp$, 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 $M'_\perp$ 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. But this is precisely what occurs when one interprets the SE-LTL formula $\phi_\perp$ as an *LTL* formula (over a set of atomic variables which includes Σ). The claim then follows from Theorem 3. □

The significance of Theorem 4 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. (However, we argue later on that a direct algorithm would probably be appreciably more efficient.)

4 Example: 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 labelled 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 $c0$, $c1$, $c2$.[4] 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).

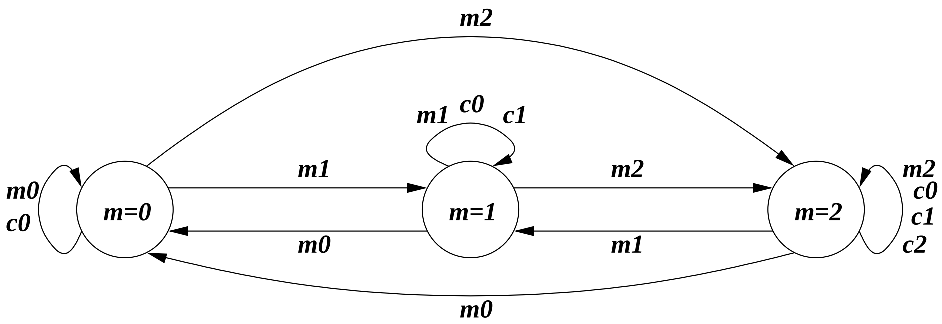

Fig. 1. The LKS of a surge protector

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

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

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

[4] 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 ellided such details here.

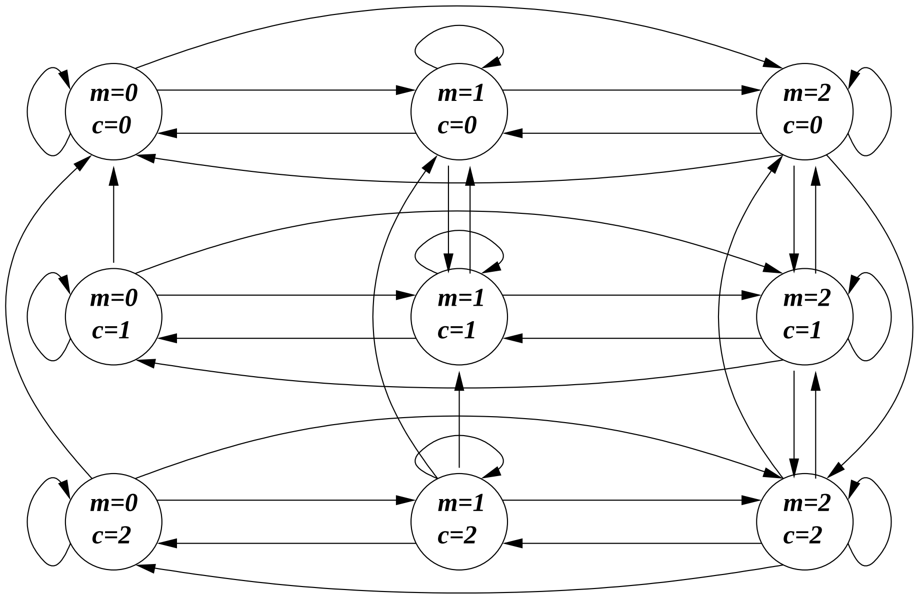

Fig. 2. The Kripke structure of a surge protector

current ranges increase, the number of states of the LKS will grow linearly, as opposed to quadratically for the Kripke structure. The 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 when modeling certain kinds of systems.

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

$$\begin{aligned}\phi_{\mathrm{s}} = {} & \mathbf{G}(((c=0 \lor c=2) \land \mathbf{X}(c=1)) \to (m=1 \lor m=2)) \land \\ & \mathbf{G}(((c=0 \lor c=1) \land \mathbf{X}(c=2)) \to m=2),\end{aligned}$$

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}:

$$\begin{aligned}\phi_{\mathrm{e}} = {} & \mathbf{G}(m0 \to ((\neg c1)\ \mathbf{U}\ (m1 \lor m2))) \land \\ & \mathbf{G}(m0 \to ((\neg c2)\ \mathbf{U}\ m2)) \land \\ & \mathbf{G}(m1 \to ((\neg c2)\ \mathbf{U}\ m2)).\end{aligned}$$

The greater simplicity of the implementation and specification associated with the state/event formalism is not purely a matter of esthetics, or even a safeguard against subtle mistakes; preliminary 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 very 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 (in seconds) taken by MAGIC to confirm that the corresponding implementation indeed meets the specification.

current range	state/event			pure state		
	B-states	B-trans.	time	B-states	B-trans.	time
$0 \leqslant m, c \leqslant 2$	4	6	20.4	8	12	71.4
$0 \leqslant m, c \leqslant 3$	5	8	35.9	14	23	204.6
$0 \leqslant m, c \leqslant 4$	6	10	51.0	22	38	558.2

Fig. 3. Comparison of state/event and pure state formalisms. B-states and B-trans. respectively denote the number of states and transitions of the Büchi automaton corresponding to the specification. Times are given in milliseconds.

5 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 to be able to exploit abstractions in the verification process:

Theorem 5. *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, if $A \models \phi$ then $M \models \phi$.*

Proof. This follows easily from the fact that every path of M is matched by a corresponding property-preserving path of A. Note that the fact that this correspondence preserves deadlock is crucial. □

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 $M_1 || \ldots || M_n \models \phi$. 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 $A_1 || \ldots || A_n \models \phi$. In the affirmative, we conclude (by Theorems 1 and 5) that $M_1 || \ldots || M_n \models \phi$ as well. In the negative, we are provided with a counterexample $\pi_A \in L(A_1 || \ldots || A_n)$ 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 $\pi \in L(M_1 || \ldots || M_n)$.

This validation check can be performed compositionally, as follows. According to Theorem 1, the counterexample is real iff for each i, the projection $\pi_A \upharpoonright A_i$ corresponds to a valid behavior of M_i. To this end, we 'simulate' $\pi_A \upharpoonright A_i$ 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 relevant projection 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.

It remains to explain how initial abstractions are generated, and how abstractions are refined. One can use the standard technique of hiding state variables and then increasingly restoring them to rule out spurious counterexamples as needed—details of this approach (in the context of Kripke structures) may be found in [CGJ+00]. However, for this technique to be sound it is necessary to ensure that the abstractions thus obtained preserve any deadlocking behavior of the concrete model (which in general may not be the case). In practice, we may either verify (through model checking[5] or otherwise) that the system $M_1 || \ldots || M_n$ is deadlock-free, or alternately subsequently validate the specification ϕ on all the deadlocking paths of $M_1 || \ldots || M_n$.

6 Conclusion and Future Work

In this paper, we have presented an attractive framework for modelling and verifying linear-time specifications on concurrent software systems. Our approach is based on both states and events, and supports compositional counterexample-guided abstraction refinement. We have also shown how standard automata-theoretic techniques for verifying linear temporal logic formulas can be ported to our framework, and have implemented these on our C model checker MAGIC. Preliminary investigations with MAGIC yield very promising results as compared to more standard state-only or event-only approaches.

There remain many avenues for further research. One is to try to further optimize the automata-theoretic part of the verification, by directly transforming an SE-LTL formula into a *labelled* Büchi automaton. An examination of a certain number of examples suggests that this could lead to a state space reduction of a factor of at most the cardinality of the alphabet set. Another direction is to investigate other, even less restrictive (and perhaps specification-dependent), notions of abstraction. We have also begun work on a compositional CEGAR-based algorithm to check deadlock-freedom. MAGIC is at present an explicit model checking tool—it could be extremely interesting to explore symbolic and partial order techniques to further tackle the state space explosion problem.

[5] We are currently working on developing an entirely different counterexample-guided compositional technique to check deadlock-freedom.

Another promising area would be to develop mechanisms to handle shared variables; at present, these can only be dealt with in clumsy fashion through shared actions. Other modifications to the basic setup could include the ability to add fairness constraints. Lastly, we are actively looking to model and verify much larger, industrial-size case studies.

References

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

[CCG+03] Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. In *Proceedings of ICSE 2003*, pages 385–395, 2003.

[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.

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

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

[GPVW95] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre 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.

[KV98] Ekkart Kindler and Tobias Vesper. ESTL: A temporal logic for events and states. *Lecture Notes in Computer Science*, 1420:365–383, 1998.

[MH01] David Schimidt Michael Huth, Radha Jagadeesan. Modal transition systems : A foundation for three-valued program analysis. In *Lecture Notes in Computer Science*, volume 2028, page 155. Springer-Verlag Heidelberg, 2001.

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

[Pnu86] Amir Pnueli. Application of temporal logic to the specification and verification of reactive systems: A survey of current trends. In J.W. de Bakker, Willem P. de Roever, and Grzegorz 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] Fabio Somenzi and Roderick Bloem. Efficient buchi automata from ltl formulae. In *Computer-Aided Verification*, pages 248–263, 2000.

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