Under consideration for publication in Formal Aspects of Computing

Concurrent Software Verification with States, Events, and Deadlocks[1,2]

Sagar Chaki[1], Edmund Clarke[2], Joël Ouaknine[3], Natasha Sharygina[1,2], Nishant Sinha[2]

[1]Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA
[2]Computer Science Department, Carnegie Mellon University, Pittsburgh, USA
[3]Oxford University Computing Laboratory, Oxford, UK

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 also present an algorithm to detect deadlocks in concurrent message-passing programs. Deadlock-freedom is not only an important and desirable property in its own right, but is also a prerequisite for the soundness of our model checking algorithm. Even though deadlock is inherently non-compositional and is not preserved by classical abstractions, our iterative algorithm employs both (non-standard) abstractions and compositional reasoning to alleviate the state-space explosion problem. The resulting framework differs in key respects from other instances of the counterexample-guided abstraction refinement paradigm found in the literature.

We have implemented this work in the MAGIC verification tool for concurrent C programs and performed tests on a broad set of benchmarks. 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 also recorded substantial reductions in time and memory consumption when performing deadlock-freedom checks with our new abstractions. Finally, we report two bugs (including a deadlock) in the source code of Micro-C/OS versions 2.0 and 2.7, which we discovered during our experiments.

Keywords: Concurrent software; Model checking; Temporal logic; States and events; Deadlock; Compositional reasoning; Counterexample-guided abstraction refinement

---

[1] This research was sponsored by 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).

[2] This article combines and builds upon the papers [CCO+04] and [CCOS04].

Correspondence and offprint requests to: Joël Ouaknine, `joel@comlab.ox.ac.uk`.

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 need to be able to ensure and guarantee their correctness.

In this endeavour, 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 [SLAM, BR01, BMMR01, BLAST, HJMS02, HJMQ03, CDH+00, PDV01, Sto02, MAGIC, CCG+03, COYC03, AQR+04]. 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 events (or actions), which can represent function calls, requests and acknowledgements, etc. Moreover, such communication is commonly data-dependent. Software behavioural claims, therefore, are often specifications defined over combinations of program events and data valuations.

Existing modelling 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 event can be encoded as a change in state variables, and likewise one can equip a state with different events 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 events are often data-dependent: considerable domain expertise is usually required to annotate the program and to specify proper claims.

In this work, we propose a framework in which both state-based and event-based properties can be expressed, combined, and verified. The modelling framework consists of *labelled Kripke structures* (LKS), which are directed graphs in which states are labelled with atomic propositions and transitions are labelled with events. 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 [MAGIC, CCG+03, COYC03], which extracts LKS models from C programs automatically via predicate abstraction [GS97]. We report promising results in the large set of benchmarks that we have tackled.

A core feature of MAGIC is the use of compositional abstraction refinement techniques developed for the efficient verification of concurrent software [COYC03]. These techniques are embedded within a counterexample-guided abstraction refinement framework (CEGAR for short) [CGJ+00]. 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.

The verification of state/event specifications on concurrent LKSs as described above requires, in order to be sound, that the global composed system be *deadlock-free*. When several components communicate via blocking message-passing, the possibility that two components might at some point have incompatible communication requirements can unfortunately in general not be discounted. Indeed, such an occurrence is termed deadlock and results in a situation in which no further progress can be made. Deadlock-freedom is often a vital specification in its own right, especially for safety-critical systems, such as embedded systems or plant controllers, that are expected always to service requests within a fixed time limit or be responsive to external stimuli.

Unfortunately, deadlock is inherently non-compositional and moreover is not preserved by classical existential or universal abstractions. One of the main contributions of this paper is the development of new, non-standard abstraction schemes that do preserve deadlock and are (in an appropriate sense) compositional. These abstractions are inspired from the notion of *failures* in the process algebra CSP [Hoa85, Ros97]. As a result, we have been able to devise a compositional CEGAR-based framework for deadlock detection, which has led to substantial time and space improvements over corresponding classical approaches found in the literature: our MAGIC implementation recorded speed-ups of up to 20 times along with up to four-fold reductions in memory usage in some of our benchmarks.

We carried out a large number of experiments to evaluate both our state/event model checking algorithm as well as our deadlock detection scheme. Our benchmarks were for the most part derived from (i) OpenSSL-0.9.6c, an open-source implementation of the SSL protocol; (ii) IPC-1.6, an inter-process communication protocol developed by ABB used to mediate information in a multi-threaded robotics control automation system; and (iii) Micro-C/OS 2.0 and 2.7, a real-time operating system for embedded applications. In the case of the latter, we discovered and report two bugs (including a deadlock), which have been acknowledged by the implementors of Micro-C/OS.

The rest of this article is organized as follows. In Section 2 we summarize related work. Section 3 introduces our state/event implementation formalism, labelled Kripke structures. We also discuss existential abstractions, parallel composition, and assorted results. This is followed by Section 4 in which we present our state/event temporal logic. We review standard automata-theoretic model checking techniques, and show how these can be adapted to this new framework. In Section 5, we illustrate these ideas by modelling a simple surge protector. We then 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. Section 6 details our compositional CEGAR algorithm for model checking state/event specifications on labelled Kripke structures, and integrates as a whole the various pieces presented in earlier sections.

Deadlock is formally introduced in Section 7. We then explain why classical abstractions are inadequate for handling deadlock in Section 8, and propose new abstractions that overcome the problems raised. Section 9 explains how these abstractions can be exploited within a compositional CEGAR framework, culminating in our deadlock detection algorithm. Finally, Section 10 is devoted to experiments and case studies, while Section 11 concludes and discusses future work.

2. Related Work

The formalization of a general notion of abstraction first appeared in [CC77]. The abstractions used in our approach are *conservative*: they are guaranteed to preserve 'undesirable' properties of the system, so that if the abstraction is bug-free then so is the original model, but may introduce spurious 'bad' behaviours not present in the original system—for more details on conservative abstractions, see, e.g., [Kur89, CGL94]. Conservative abstractions usually lead to significant reductions in the state space but in general require an iterated abstraction refinement mechanism (see below) in order to establish specification satisfaction.

Counterexample-guided abstraction refinement [Kur94, CGJ+00], 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 labelled 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 labelled 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. Their work exploits partial-order reduction techniques and has been implemented in the LTSA tool [LTSA].

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 labelled 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.

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

Deadlock detection has been widely studied in various contexts. One of the earliest deadlock-detection tools, for the process algebra CSP, was FDR [FSE]; see also [RD87, BR91, MJ97, Ros97, MH00]. Corbett has evaluated various deadlock-detection methods for concurrent systems [Cor96] while Demartini et al. have developed deadlock-detection tools for concurrent Java programs [DIS99]. However, to the best of our knowledge, none of these approaches involve abstraction refinement or compositionality in automated form.

Very recently, the notion of *stuck-freedom* [FHRR04], closely related to deadlock-freedom, has been developed for the process algebra CCS. Stuck-freedom is compositional and an algorithm for model checking stuck-free conformance for concurrent software has been implemented in the tool ZING [AQR$^+$04].

3. Labelled Kripke Structures

A *labelled 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-labelling 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-labelling function*. We often write $s \xrightarrow{A} s'$ to mean that $(s, s') \in T$ and $A \subseteq \mathcal{E}(s, s')$.[3] In case A is a singleton set $\{a\}$ we write $s \xrightarrow{a} s'$ rather than $s \xrightarrow{\{a\}} s'$. Note that T and $\mathcal{E}$ are entirely determined by the collection of transitions of the form $s \xrightarrow{a} s'$. Note also that both states and transitions are 'labelled', the former with sets of atomic propositions, and the latter with non-empty sets of events.

We do *not* assume that the transition relation is total, and indeed we will see that the possibility of *deadlock* (reaching a state with no successor) is an unavoidable consequence of the blocking message-passing

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

semantics we have chosen to model concurrency. We are, however, mainly interested in non-deadlocking behaviour, and an algorithm to detect deadlock is presented in Section 9.

A *path* $\pi = \langle s_1, a_1, s_2, a_2, \ldots \rangle$ of an LKS is an alternating sequence of states and events subject to the following: for each $i \geqslant 1$, $s_i \in S$, $a_i \in \Sigma$, and $s_i \xrightarrow{a_i} s_{i+1}$. Paths can be infinite or finite, however we require that a finite path end with a state (rather than with an event). We usually assume that a path is infinite unless specified otherwise. Note that finite paths are not required to be 'maximal'.

Given an LKS M, we write $Path(M)$ to denote the set of infinite paths of M whose first state lies in the set $Init$ of initial states of M. Later on, we will also write $FPath(M)$ to denote the collection of finite $Init$-rooted paths of M.

Intuitively, an LKS M represents a finite-state model of some program. States correspond to program statements, and atomic state propositions correspond to predicates on the local variables of the program, or to any other property we may wish to record of a given state. States are then labelled with precisely all of the atomic propositions that they satisfy. Upon moving from one state to another, we assume that the LKS performs some visible event, for potential synchronous communication purposes with another concurrent component (LKS). The LKS may offer several distinct events—corresponding to distinct offers of synchronization—only one of which will be performed at any one time; for this reason, transitions are labelled with non-empty sets of events.

We are mainly interested in modelling reactive systems and components intended to run indefinitely, such as file servers or operating systems, hence our primary focus on infinite paths. Of course, systems with a well-defined notion of termination, such as authentication protocols, can also be handled in our framework, by modelling termination as a transition to a special 'sink' state, from which only self-transitions are possible.

3.1. Abstraction

The notion of abstraction is central to our approach. We list below the properties that we require of any abstraction scheme to be usable in our framework, and then give a concrete method for constructing abstractions that meet these criteria in the next subsection.

In general, given an LKS M, we want an abstraction of M to be conservative, i.e., to over-approximate, in a controlled manner, the behaviours of M. We want the abstraction to adequately reflect the events that M can perform, but we may only be interested in a specific subset of the atomic state propositions of M, and consequently require the abstraction to faithfully reflect only those state propositions that belong to this subset.

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

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

The set P_A of atomic state propositions of the abstraction is the subset of state propositions of M that we are interested in. An important special case is when $P_A = \emptyset$, in which case we really are only interested in the sequences of *events* that M can perform—this will be useful when dealing with deadlock detection later on.

Note that this paper focuses on *linear-time* behaviours, and hence our abstractions are defined in terms of paths rather than trees. For a branching-time account of concurrent software verification, we refer the reader to [CCG+04].

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 (see Proposition 3 in Section 4).

3.2. Existential Quotients of Labelled Kripke Structures

We present a specific method for constructing abstractions that meet the criteria laid out above. An abstraction of an LKS M is obtained by quotienting the states of M by a suitable equivalence relation. The idea is very similar to the well-known notion of 'existential abstraction' for ordinary Kripke structures in which certain variables are hidden [CGJ+00].

More precisely, for $M = (S, \mathit{Init}, P, \mathcal{L}, T, \Sigma, \mathcal{E})$, let $P_A \subseteq P$ be a subset of atomic state propositions of M, and let $\approx$ be an equivalence relation on the states S of M that respects P_A: if $s \approx s'$, then $\mathcal{L}(s) \cap P_A = \mathcal{L}(s') \cap P_A$. The *existential quotient* of M (with respect to P_A and $\approx$) is the LKS $A = (S_A, \mathit{Init}_A, P_A, \mathcal{L}_A, T_A, \Sigma_A, \mathcal{E}_A)$ such that:

1. $S_A = S/{\approx}$, the collection of equivalence classes of S,
2. $\mathit{Init}_A = \{[s] \in S/{\approx} \mid \exists s' \in [s] \,\centerdot\, s' \in \mathit{Init}\}$,
3. for all $s \in S, \mathcal{L}_A([s]) = \mathcal{L}(s) \cap P_A$,
4. $\Sigma_A = \Sigma$, and
5. 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 write $M/(P_A, \approx)$ or even $M/{\approx}$ (when the set P_A is clearly understood from the context) to denote the abstraction A of M obtained in the above manner. The main advantage of working with $M/{\approx}$ is that in general it has a smaller state space than M.

We now have:

Proposition 1. For M an LKS, any existential quotient $M/(P_A, \approx)$ of M is a genuine abstraction of M in the sense of Subsection 3.1: $M \sqsubseteq M/(P_A, \approx)$.

In fact, one can show that any abstraction $M/(P_A, \approx)$ of M *simulates* M, so that existential quotients are *branching-time* abstractions as well as *linear-time* abstractions; we will not, however, make use of this fact in this paper.

Note that any P_A-respecting equivalence relation $\approx$ on the state space of M can be viewed as a partition of this state space. Moreover, since any refinement (sub-partition) of a P_A-respecting partition is again P_A-respecting, we have:

Proposition 2. Let M be an LKS and let $M/(P_A, \approx)$ be an abstraction of M. For any refinement $\approx'$ of the partition $\approx$, $M/(P_A, \approx')$ is an abstraction of M that is also a refinement of $M/(P_A, \approx)$: $M \sqsubseteq M/(P_A, \approx') \sqsubseteq M/(P_A, \approx)$.

We leave the straightforward proofs of Propositions 1 and 2 to the reader.

3.3. Parallel Composition

The notion of parallel composition that we consider in this paper allows for communication through shared events 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. Assume that M_1 and M_2 are *compatible*, i.e., that they do not share states or variables: $S_1 \cap S_2 = P_1 \cap P_2 = \emptyset$. The parallel composition of M_1 and M_2 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 $s'_1 = s_1$,
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 [Hoa85, Ros97]; see also [ACFM85].

Note that, because of our blocking semantics for parallel composition, it is possible for $M_1 \parallel M_2$ to exhibit deadlock—i.e., reach some state with no outgoing transition—even if both M_1 and M_2 are deadlock-free. Deadlock arises through incompatible communication requirements: for example, one component might be waiting for some input or acknowledgement that the other component is incapable of supplying. In that case the two components are stuck and cannot make any further progress. Deadlock is clearly an undesirable behaviour and is usually symptomatic of a bug in the program.

Let M_1 and M_2 be as above, and let $\pi = \langle (s_1^1, s_1^2), a_1, \ldots \rangle$ be an alternating infinite sequence of states and events of $M_1 \parallel M_2$. The *projection* $\pi \restriction M_i$ of π on M_i consists of the (possibly finite) subsequence of $\langle s_1^i, a_1, \ldots \rangle$ 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 labelled 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]).

Theorem 1.

1. Parallel composition is 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 $\pi \in \mathit{Path}(M_1 \parallel \ldots \parallel M_n)$ iff, for each i, there exists $\pi'_i \in \mathit{Path}(M_i)$ such that $\pi \restriction M_i$ 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 component-wise.

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 $M = (S, \mathit{Init}, P, \mathcal{L}, T, \Sigma, \mathcal{E})$, 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 $\pi = \langle s_1, a_1, s_2, a_2, \ldots \rangle$ be an infinite path. Let π^i stand for the suffix of π starting in state s_i. We then inductively define path-satisfaction of SE-LTL formulas as follows:

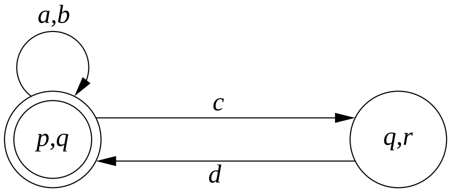
1. $\pi \vDash p$ iff s_1 is the first state of π and $p \in \mathcal{L}(s_1)$,
2. $\pi \vDash a$ iff a is the first event of π,
3. $\pi \vDash \neg\phi$ iff $\pi \nvDash \phi$,
4. $\pi \vDash \phi_1 \wedge \phi_2$ iff $\pi \vDash \phi_1$ and $\pi \vDash \phi_2$,
5. $\pi \vDash \mathbf{X}\phi$ iff $\pi^2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

We then let $M \vDash \phi$ iff, for every infinite path $\pi \in \mathit{Path}(M)$, $\pi \vDash \phi$.

Let us also introduce the derived $\mathbf{W}$ operator: $\phi_1 \;\mathbf{W}\; \phi_2$ iff $(\mathbf{G}\phi_1) \vee (\phi_1 \;\mathbf{U}\; \phi_2)$. We will also freely use standard Boolean connectives such as $\rightarrow$, etc.

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

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



As the reader may easily verify, $M \vDash \mathbf{G}(c \rightarrow \mathbf{F}r)$ but $M \nvDash \mathbf{G}(b \rightarrow \mathbf{F}r)$. Note also that $M \vDash \mathbf{G}(d \rightarrow \mathbf{F}r)$[5], but $M \nvDash \mathbf{G}(d \rightarrow \mathbf{XF}r)$.

We remark that SE-LTL formulas conform with our intuitive interpretation of specifications provided that the LKSs under consideration are deadlock-free: indeed, an LKS with no infinite path, for example, vacuously satisfies *any* SE-LTL formula! Deadlock checking is therefore a vital prerequisite for ensuring the soundness of our analysis. A system found to exhibit deadlock should be sent back to its designer for debugging; if it turns out the deadlocks somehow do not correspond to genuine bugs, then at the very least they should be managed via the 'soft' sink-state-based approach discussed in Section 3.

Let us record the following proposition, which validates our decision not to distinguish between $\sim$-equivalent LKSs:

Proposition 3. Let M and M' be LKSs with $M \sim M'$. Then, for any SE-LTL formula ϕ, $M \vDash \phi$ iff $M' \vDash \phi'$.

4.1. Automata-based Verification

We aim to reduce SE-LTL verification problems to standard automata-theoretic techniques for LTL. Note that a straightforward—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-labelling function. An LTL formula is an SE-LTL formula that 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 in P.

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

Note that the transition relation is unlabelled, and that the states of a Büchi automaton are labelled 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 it is Init_B-rooted, if it is consistent with the transition relation, and if $\mathsf{inf}(\pi) \cap \mathit{Acc} \neq \emptyset$. The set of all such accepted paths is written $\mathit{Path}(B)$.

Let $M = (S, \mathit{Init}, P, \mathcal{L}, T)$ be a Kripke structure. The state-labelling function $\mathcal{L} : S \rightarrow 2^P$ indicates, for each state $s \in S$, exactly which atomic propositions hold at s; such labelling 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 $B = (S_B, \mathit{Init}_B, P, \mathcal{L}_B, T_B, \mathit{Acc})$ be a Büchi automaton over the same set of atomic propositions as M. We can define the 'standard' product $M \times B = (S', \mathit{Init}', -, -, T', \mathit{Acc}')$ as a product of Büchi automata. More precisely,

[5] Indeed, according to the semantics, a path satisfies the formula d iff d is the first event following the initial state of the path. Observe now that the only state from which d is immediately possible if the right-hand one, in which r holds, and therefore in which *a fortiori* $\mathbf{F}r$ holds. This example highlights a somewhat counterintuitive feature of our semantics in which states and events are combined.

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

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

$M \vDash \phi$ iff $\mathit{Path}(M \times B_{\neg\phi}) = \emptyset$.

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

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

1. $S' = \{(s,b) \in S \times S_B \mid \widetilde{\mathcal{L}(s)} \text{ implies } \exists\Sigma \,\boldsymbol{.}\, \mathcal{L}_B(b)\}$,[6]
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,

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

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-labelled outgoing transitions. 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 labelled 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

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

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

[6] The term $\exists\Sigma \,\boldsymbol{.}\, \mathcal{L}_B(b)$ 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 for which this operation is trivial.

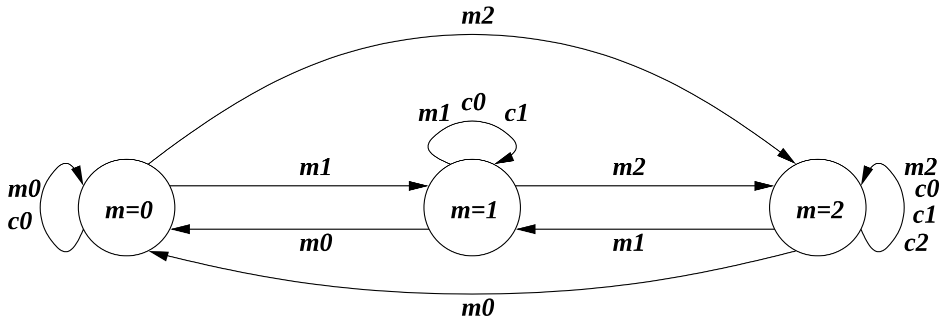


Fig. 1. The LKS of a surge protector.

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 *labelled* Kripke structures, at no additional space or time costs (no change in the sizes of the LKS M or the formula ϕ).

Note that a Büchi automaton has non-empty accepted language iff it accepts a *lasso*, i.e., a path consisting of a finite prefix leading to an infinite loop. Efficient algorithms for checking emptiness of Büchi automata are well-known, see e.g., [CGP99]. Moreover, these algorithms can always be required to produce an explicit witness in the case of non-emptiness. Since a lasso in $M \otimes B_{\neg\phi^\flat}$ clearly projects onto a lasso in M, we have the following:

Theorem 4. For an LKS M and an SE-LTL formula ϕ, if $M \nvDash \phi$, then one can extract a lasso $\pi \in \mathit{Path}(M)$ such that $\pi \nvDash \phi$.

Such counterexamples allow us to incrementally refine our abstractions during model checking.

5. A Surge Protector

We describe a toy model of 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 $c0$, $c1$, $c2$.[7] 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, behaviours which are here omitted for simplicity).

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 essentially the same behaviour 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 unfavourably 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

[7] The reader may object that we have only allowed for *Boolean* variables in our definition of labelled 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.

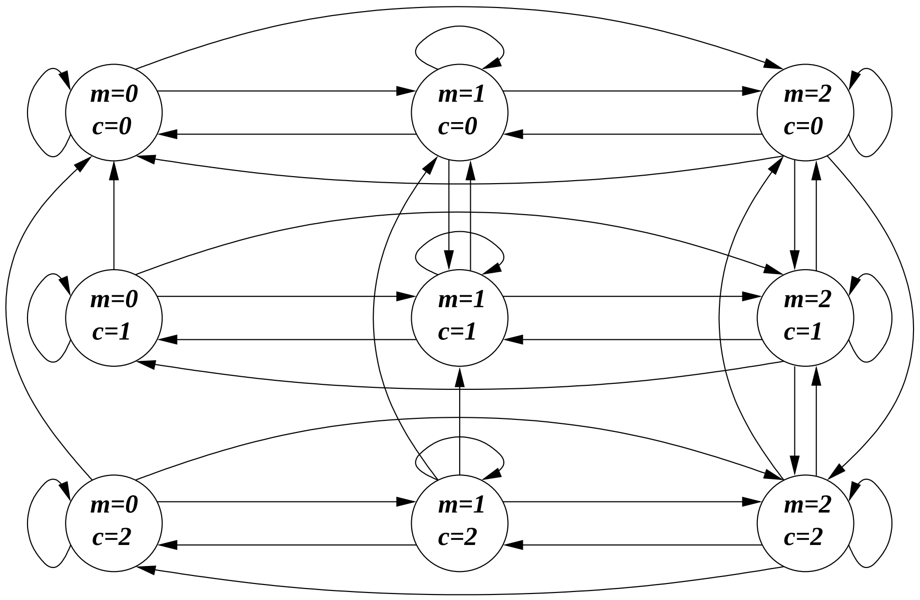


Fig. 2. The Kripke structure of a surge protector.

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.

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 [WRING]. 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. The parameter *Range* indicates the maximum allowable current value.

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

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 automaton construction time and **T-T** is the total verification time. All times are reported in milliseconds. A $*$ indicates that the verification did not terminate in 10 minutes.

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 SE-LTL Verification

We now discuss how our framework enables us to verify SE-LTL specifications on parallel compositions of labelled 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 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. $\square$

Suppose now 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 $M_1 \parallel \ldots \parallel M_n \models \phi$. Let us assume that $M_1 \parallel \ldots \parallel M_n$ is deadlock-free, a requirement for which we provide an algorithm in Section 9.[8] 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 $A_1 \parallel \ldots \parallel A_n \models \phi$. In the affirmative, we conclude (by Theorems 1 and 5) that $M_1 \parallel \ldots \parallel M_n \models \phi$ as well. In the negative, we are provided (thanks to Theorem 4) with an abstract counterexample $\pi \in \mathit{Path}(A_1 \parallel \ldots \parallel A_n)$ such that $\pi \not\models \phi$. We must then determine whether this counterexample is real or spurious, i.e., whether it corresponds to a counterexample in $M_1 \parallel \ldots \parallel M_n$ or not.

This validation check can be performed compositionally, as follows. According to Theorem 1, the counterexample is real iff for each i, the projection $\pi \restriction A_i$ corresponds to (the prefix of) a valid behaviour of M_i. To this end, we 'simulate' $\pi \restriction A_i$ 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 $\pi \restriction A'_i$ of the spurious counterexample π. Note that if π is a lasso (as per Theorem 4), the projection $\pi \restriction A_i$ is either a lasso or a finite path.

[8] Of course, as explained in Section 4, the procedure given here will work perfectly well whether or not $M_1 \parallel \ldots \parallel M_n$ is deadlock-free. Unfortunately, for deadlocking systems, the answer we get may not conform with our intuitive understanding of the specification ϕ. For that reason, an initial deadlock-freedom check is highly recommended.

Algorithm *SE-LTL_Model_Checking* $(M_1, \ldots, M_n \ ; \ \phi)$
for $i := 1$ to n : **let** $\approx_i :=$ the coarsest partition of M_i that respects the atomic state propositions in ϕ;
repeat forever
  decide whether $M_1/\approx_1 \parallel \ldots \parallel M_n/\approx_n \models \phi$ via automata-based algorithm (Theorems 3 and 4);
  if there is no counterexample **then**
    return "$M_1 \parallel \ldots \parallel M_n \models \phi$"
  else suppose $\pi \in \mathit{Path}(M_1/\approx_1 \parallel \ldots \parallel M_n/\approx_n)$ violates ϕ;
  find i such that *SE-LTL_Validate/Refine*$(M_i, \approx_i, \pi\lceil M_i/\approx_i)$ reports "`spurious`";
  if no such i **then**
    return "$M_1 \parallel \ldots \parallel M_n \not\models \phi$ " along with counterexample derived from π
  else refine $\approx_i$ to rule out spurious counterexample $\pi\lceil M_i/\approx_i$;
endrepeat

Fig. 4. The overall SE-LTL model checking algorithm for a concurrent system $M_1 \parallel \ldots \parallel M_n$ and specification ϕ.

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.[9]

The advantage of this approach is that the abstractions that we consider here, as detailed in Section 3, are obtained by lumping together states of the original LKSs, and have therefore smaller state spaces. Since composing components in parallel can in general lead to an exponential blow-up of the state space, the overall reduction in size obtained by composing the reduced abstractions together can be enormous.

Let us return to our SE-LTL specification ϕ, and let us fix throughout P_ϕ to be the set of all atomic state propositions appearing in ϕ. Consider any of the M_i's. Recall from Section 3 that an abstraction of M_i is entirely determined by a P_ϕ-respecting partition $\approx_i$ of the set of states of M_i: such an abstraction is denoted $M_i/\approx_i$.

The *initial abstraction* $M_i/\approx_i^1$ is the coarsest possible: $s \approx_i^1 s'$ iff $\mathcal{L}_i(s) \cap P_\phi = \mathcal{L}_i(s') \cap P_\phi$. Suppose now that we are handed $\pi_i \in \mathit{Path}(M_i/\approx_i^k) \cup \mathit{FPath}(M_i/\approx_i^k)$ (i.e., π_i is either a lasso or a finite path of $M_i/\approx_i^k$). We must determine whether π_i is a real or spurious counterexample component, i.e., whether π_i gives rise to a valid path of M_i or not. Moreover, in the latter case, we want to refine our partition $\approx_i^k$ into $\approx_i^{k+1}$ so that π_i is rejected by $M_i/\approx_i^{k+1}$.

This validation/refinement step proceeds as follows. For any set Q of states of M_i and event a, let $\mathit{Succ}(Q, a) = \{s' \mid \exists s \in Q \centerdot s \xrightarrow{a} s'\}$ denote the set of a-successors of Q in M_i. Let us first suppose that $\pi_i = \langle s_1, a_1, s_2, a_2, \ldots, a_m, s_{m+1}\rangle$ is a finite path of $M_i/\approx_i^k$. Start with the set $Q_1 = \mathit{Init}_i \cap s_1$ of those initial states of M_i that belong to s_1, and compute successively $Q_{j+1} = \mathit{Succ}(Q_j, a_j) \cap s_{j+1}$. If, upon reaching the end of π_i, Q_{m+1} is non-empty, then clearly π_i is a valid finite path of M_i. Otherwise, let Q_{j+1} be the first empty set thus generated. Refine the partition $\approx_i^k$ by splitting s_j into Q_j and $s_j \setminus Q_j$, yielding a new partition $\approx_i^{k+1}$. It is then easy to see that $M_i/\approx_i^{k+1}$ will reject π_i.

In case π_i is a lasso, things are slightly more complicated. If M_i rejects π_i, then the algorithm above will establish this in the very same manner, by eventually producing an empty set of states Q_{j+1}. On the other hand, if π_i is accepted by M_i then there will be sets of states $Q_j = Q_{j+p}$ such that Q_{j+p} is obtained from Q_j by following the loop part of π_i a finite number of times. Since all state spaces involved are finite the search will always terminate with one or the other answer after a finite number of iterations.

This validation/refinement step is similar to that originally proposed by Clarke *et al.* [CGJ+00]; see also [COYC03]. Let us refer to this procedure as *SE-LTL_Validate/Refine*$(M_i, \approx_i, \pi_i)$. The full algorithm for checking whether $M_1 || \ldots || M_n \models \phi$ is given in Figure 4. Note that the abstraction, counterexample-validation, and refinement steps are all performed one component at a time.

[9] 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 major difficulties.

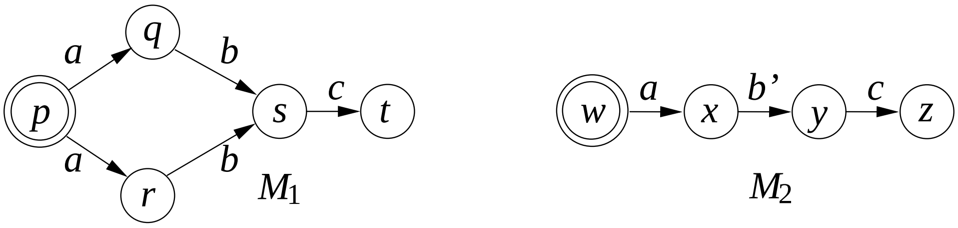


Fig. 5. Two sample LTSs M_1 and M_2. Initial states are doubly circled.

7. Deadlock

As explained earlier, as a result of our blocking message-passing semantics, a parallel composition $M_1 \parallel \ldots \parallel M_n$ of components may exhibit deadlock (reaching a state with no outgoing transition), even if each M_i is deadlock-free. Deadlock corresponds to inconsistent communication requirements among the M_i's.

An important observation is that deadlock is exclusively a function of the communication structure of LKSs, and does not depend in any way on local variables (atomic state propositions), since our framework does not allow these to be shared. In other words, if a particular parallel composition of LKSs exhibits deadlock, then the deadlock will remain regardless of any changes to the various state-labelling functions of its components, and likewise for deadlock-free systems. For this reason, we shall work in the remainder of this paper with *labelled transition systems* (*LTS*s for short) rather than LKSs.

An LTS is simply an LKS minus the state proposition structure: in other words a quintuple $(S, \mathit{Init}, T, \Sigma, \mathcal{E})$ with the same conventions as for LKSs. Given such an LTS M we occasionally write $S(M)$ and $\Sigma(M)$ to denote S and Σ respectively.

Paths are defined in the same manner as for LKSs, except that from now on we are exclusively concerned with the *finite* variety. Recall that $\mathit{FPath}(M)$ stands for the set of finite paths accepted by the LTS M. A *trace* over some alphabet Σ is a finite sequence of events of Σ. A trace $\langle a_1, a_2, \ldots, a_m \rangle$ over $\Sigma(M)$ is accepted by M iff there exist $s_1, s_2, \ldots, s_{m+1}$ such that $\langle s_1, a_1, s_2, a_2, \ldots, a_m, s_{m+1} \rangle \in \mathit{FPath}(M)$. Paths and traces are usually represented with the letters π and θ respectively.

A state $s \in S(M)$ is said to *refuse* an event $a \in \Sigma(M)$ iff there is no transition from s labelled by a. The *refusal* of a state s is the set of all events that it refuses: $\mathit{Ref}(s) = \{a \in \Sigma(M) \mid \nexists s' \in S(M) \centerdot s \xrightarrow{a} s'\}$.

Let θ be a trace of M and let $F \subseteq \Sigma(M)$ be a set of events. Suppose that M can accept θ along some path ending in a state s with $\mathit{Ref}(s) = F$. Then we say that (θ, F) is a *failure* of M. We write $\mathit{Fail}(M)$ to denote the set of all failures of M. Note that several failures can share the same trace component, since a given trace may be accepted along more than one path.

Let M_1 and M_2 be two LTSs over a common alphabet $\Sigma(M_1) = \Sigma(M_2)$. We say that M_1 and M_2 are *failure-equivalent* if $\mathit{Fail}(M_1) = \mathit{Fail}(M_2)$.

Finally, we say that M has a *deadlock* if it can reach a state which refuses its entire alphabet $\Sigma(M)$, in other words if $(\theta, \Sigma(M)) \in \mathit{Fail}(M)$ for some θ.

Notice that two failure-equivalent LTSs will have identical deadlocking behaviour (or lack thereof). Since failure-equivalence is preserved by parallel composition (see Theorem 6 below), we shall only be interested in LTSs up to failure-equivalence.

The notions of refusal, failure, and deadlock that we are using here are borrowed from CSP (with minor modifications[10])—see [Hoa85, Ros97].

As a simple example, consider the LTSs M_1 and M_2 depicted in Figure 5. Let $\Sigma(M_1) = \{a, b, c\}$ and $\Sigma(M_2) = \{a, b', c\}$. Then M_1 has seven paths: $\langle p \rangle$, $\langle p, a, q \rangle$, $\langle p, a, r \rangle$, $\langle p, a, q, b, s \rangle$, $\langle p, a, r, b, s \rangle$, $\langle p, a, q, b, s, c, t \rangle$, and $\langle p, a, r, b, s, c, t \rangle$. It has four traces: $\langle \rangle$, $\langle a \rangle$, $\langle a, b \rangle$, and $\langle a, b, c \rangle$, and four failures $(\langle \rangle, \{b, c\})$, $(\langle a \rangle, \{a, c\})$, $(\langle a, b \rangle, \{a, b\})$, and $(\langle a, b, c \rangle, \{a, b, c\})$. Hence M_1 has a deadlock. Also, M_2 has four paths, four traces, four failures and a deadlock.

The notion of projection defined in Section 3 carries over naturally to traces. Let M be an LTS, and let

[10] The main difference is that our refusals are *maximal*, whereas in the CSP semantics refusals are only required to be subsets of disabled events. This distinction is immaterial insofar as deadlock-checking is concerned.

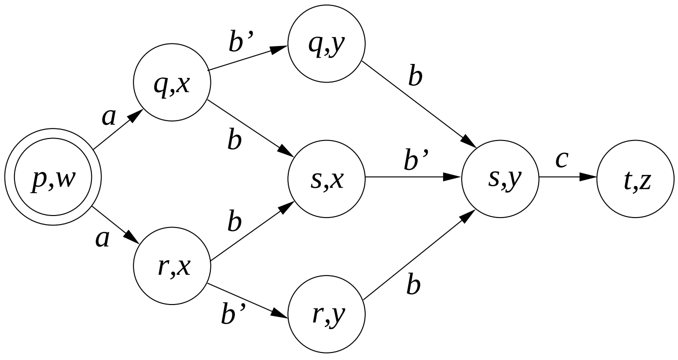


Fig. 6. Parallel composition of LTSs M_1 and M_2 from Figure 5.

θ be a sequence of events, which may or may not belong to $\Sigma(M)$. The *projection* $\theta\lceil M$ of θ on M consists of the subsequence of θ obtained by simply removing all events not belonging to $\Sigma(M)$.

Recall the notion of parallel composition from Section 3. Figure 6, for example, shows the LTS $M_1 \parallel M_2$ where M_1 and M_2 are the LTSs shown in Figure 5. Note that its alphabet is $\{a, b, c\} \cup \{a, b', c\} = \{a, b, b', c\}$.

The following Theorem lists well-known results from CSP [Ros97].

Theorem 6.

1. Parallel composition is associative and commutative up to failure-equivalence. Thus, in particular, no bracketing is required when combining more than two LTSs.
2. Parallel composition is a congruence with respect to failure-equivalence: let $M_1, M_1', M_2, M_2', \ldots, M_n, M_n'$ be LTSs, with each M_i failure-equivalent to M_i'. Then $M_1 \parallel \ldots \parallel M_n$ is failure-equivalent to $M_1' \parallel \ldots \parallel M_n'$.
3. Let $M_1, \ldots, M_n$ be LTSs, θ a sequence of events, and F a set of events. Then $(\theta, F) \in \mathit{Fail}(M_1 \parallel \ldots \parallel M_n)$ iff there exist sets of events $F_1, \ldots, F_n$ such that (i) $F = \bigcup_{i=1}^{n} F_i$, and (ii) for each i, $(\theta\lceil M_i, F_i) \in \mathit{Fail}(M_i)$. In other words, whether a parallel composition of LTSs has a given failure can be checked by projecting and examining the failure on each individual component separately.

Theorem 6(3) highlights the compositional aspect of failures and is a key ingredient of our deadlock-detection algorithm, which we present in Section 9.

8. Abstraction

In this section we introduce the abstractions that we shall use in the remainder of this paper. Once again these abstractions are based on existential quotients of LTSs (see Section 3), although as we shall see quotient LTSs on their own are inadequate for deadlock detection.

Recall that quotient LTSs are obtained by lumping together states of a given LTS. Since there are no atomic state propositions to respect in the case of LTSs, any partition of the state space of an LTS gives rise to a *bona fide* quotient LTS. For M an LTS and $\approx$ a partition of $S(M)$, we again write $M/\!\approx$ to denote the resulting quotient LTS. The states of M are said to be *concrete* states, and are usually represented with lowercase Roman letters such as s, whereas the states of $M/\!\approx$, which are called *abstract* states, are represented either as equivalence classes (e.g., $[s]$) or with the lowercase Greek letter α.

The existential nature of the transition relation of quotient LTSs immediately yields the following:

Proposition 4. Let M be an LTS and $M/\!\approx$ a quotient LTS of M. For any path $\langle s_1, a_1, s_2, \ldots, a_m, s_{m+1}\rangle \in \mathit{FPath}(M)$, we have $\langle [s_1], a_1, [s_2], a_2, \ldots, a_m, [s_{m+1}]\rangle \in \mathit{FPath}(M/\!\approx)$.

Note the following facts about the LTSs in Figure 7: (i) M_1 and M_2 both have deadlocks but $M_1 \parallel M_2$ does not; (ii) neither M_3 nor M_4 has a deadlock but $M_3 \parallel M_4$ does; (iii) M_1 has a deadlock and M_3 does not have a deadlock but $M_1 \parallel M_3$ has a deadlock; (iv) M_1 has a deadlock and M_4 does not have a deadlock but $M_1 \parallel M_4$ does not have a deadlock (assuming that $\Sigma(M_4) = \{a, b\}$); (v) M_1 has a deadlock but the quotient

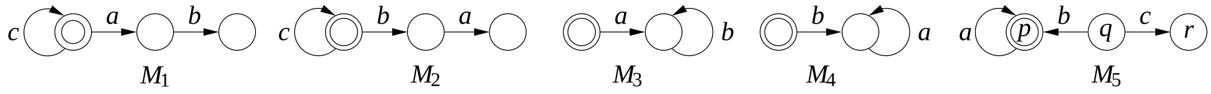


Fig. 7. Five sample LTSs. Initial states are doubly circled.

LTS obtained by lumping all the states of M_1 into a single equivalence class does not have a deadlock; (vi) finally, M_5 has no deadlock (since state r cannot be reached from the initial state p), yet the quotient LTS obtained by lumping together states p and q does have a deadlock.

These examples highlight the following facts: (i) deadlock is inherently non-compositional: neither its presence nor its absence is preserved by parallel composition; (ii) existential abstractions on their own are inadequate for handling deadlock: they preserve neither its presence nor its absence. (We remark that *universal* abstractions are likewise inadequate for handling deadlock, for the same reason.)

The inadequacy of existential abstractions with respect to deadlock is to be contrasted with their adequacy for SE-LTL: existential quotients preserve the *non-satisfaction* of SE-LTL formulas (as per Theorem 5), and thus lead to an iterative algorithm for model checking SE-LTL formulas. In order to be able to do the same for deadlock detection, we need to equip our existential abstractions with additional structure.

Let us take a closer look at the non-preservation of deadlock by existential abstractions. Consider a quotient LTS $M/\approx$ and a state α of $M/\approx$. It is easy to see that $\mathit{Ref}(\alpha) = \bigcap_{s\in\alpha} \mathit{Ref}(s)$. In other words, the refusal of an abstract state α under-approximates the refusals of each of its corresponding concrete states. In order to preserve deadlock, we must instead require that refusals of concrete states be *over-approximated*. This can be achieved by simply taking the union of the refusals of the concrete states. This leads us to the notion of *abstract refusals*, which we now define formally.

Let M be an LTS and $\approx$ a partition of the state space of M. For any abstract state $\alpha \in S(M/\approx)$, define the *abstract refusal* of α to be

$$\mathit{AbsRef}(\alpha) = \bigcup_{s\in\alpha} \mathit{Ref}(s).$$

Moreover, for a parallel composition $M_1/\approx_1 \parallel \ldots \parallel M_n/\approx_n$ of quotient LTSs, we extend the notion of abstract refusal by letting, for any $\alpha = (\alpha_1, \ldots, \alpha_n) \in S(M_1/\approx_1 \parallel \ldots \parallel M_n/\approx_n)$,

$$\mathit{AbsRef}(\alpha) = \bigcup_{i=1}^{n} \mathit{AbsRef}(\alpha_i).$$

This naturally leads us to the notion of *abstract failures*, which are similar to failures, except that refusals are replaced by abstract refusals. Let $\widehat{M}$ be an LTS for which abstract refusals are defined (i.e., $\widehat{M}$ is either a quotient LTS or a parallel composition of such). Let θ be a trace of $\widehat{M}$ and let $F \subseteq \Sigma(\widehat{M})$ be a set of events. Suppose that $\widehat{M}$ can accept θ along some path ending in a state α with $\mathit{AbsRef}(\alpha) = F$. Then we say that (θ, F) is an *abstract failure* of $\widehat{M}$. We write $\mathit{AbsFail}(\widehat{M})$ to denote the set of all abstract failures of $\widehat{M}$.

The following lemma essentially states that the failures of an LTS M are always subsumed by the abstract failures of its quotient LTS $M/\approx$:

Lemma 1. Let M be an LTS, $\approx$ a partition of the state space of M, and $M/\approx$ the quotient LTS induced by $\approx$. Then for all $(\theta, F) \in \mathit{Fail}(M)$, there exists $F' \supseteq F$ such that $(\theta, F') \in \mathit{AbsFail}(M/\approx)$.

Proof. This is a straightforward consequence of Proposition 4. □

The following lemma shows that abstract failures, like failures, are compositional: the abstract failures of a concurrent system $M_\parallel$ can be decomposed naturally into abstract failures of each of its components.

Lemma 2. Let $M_1/\approx_1, \ldots, M_n/\approx_n$ be quotient LTSs, θ a sequence of events, and F a set of events. Then $(\theta, F) \in \mathit{AbsFail}(M_1/\approx_1 \parallel \ldots \parallel M_n/\approx_n)$ iff there exist sets of events $F_1, \ldots, F_n$ such that (i) $F = \bigcup_{i=1}^{n} F_i$, and (ii) for each i, $(\theta{\restriction}M_i/\approx_i, F_i) \in \mathit{AbsFail}(M_i/\approx_i)$.

Proof. This follows from the fact that θ is a trace of $M_1/\approx_1 \parallel \ldots \parallel M_n/\approx_n$ iff for each i, $\theta{\restriction}M_i/\approx_i$ is a trace of $M_i/\approx_i$, which itself is implied by Theorem 6(3). □

In the remainder of the paper we shall often use the following facts implicitly: $\Sigma(M_1/{\approx_1} \parallel \ldots \parallel M_n/{\approx_n}) = \bigcup_{i=1}^{n} \Sigma(M_i/{\approx_i}) = \bigcup_{i=1}^{n} \Sigma(M_i) = \Sigma(M_1 \parallel \cdots \parallel M_n)$.

Abstract failures lead naturally to the notion of *abstract deadlocks*. Let $\widehat{M_\parallel} = M_1/{\approx_1} \parallel \ldots \parallel M_n/{\approx_n}$ be a parallel composition of quotient LTSs. Then $\widehat{M_\parallel}$ is said to have an *abstract deadlock* if $(\theta, \Sigma(\widehat{M_\parallel})) \in \mathit{AbsFail}(\widehat{M_\parallel})$ for some trace θ of $\widehat{M_\parallel}$. In other words, abstract deadlocks arise from traces in the abstracted system that lead to an abstract refusal of the entire alphabet.

9. Compositional Counterexample-guided Deadlock Detection

We now discuss how the abstractions defined in the previous section enable us to decide incrementally and compositionally whether or not a parallel composition of LTSs is deadlock-free.

Analogously to Theorem 5, the following result is the key ingredient needed to exploit our abstractions in the deadlock-checking process:

Theorem 7. Let $M_1, \ldots, M_n$ be LTSs, and let $\approx_1, \ldots, \approx_n$ be partitions of the state spaces of the M_i respectively. If $M_1/{\approx_1} \parallel \ldots \parallel M_n/{\approx_n}$ has no abstract deadlock, then $M_1 \parallel \ldots \parallel M_n$ is deadlock-free.

Proof. Let us establish the contrapositive. To this end, write $M_\parallel = M_1 \parallel \ldots \parallel M_n$ and $\widehat{M_\parallel} = M_1/{\approx_1} \parallel \ldots \parallel M_n/{\approx_n}$, and suppose that $(\theta, \Sigma(M_\parallel)) \in \mathit{Fail}(M_\parallel)$. By Theorem 6(3), there are sets of events $F_1, \ldots, F_n$ whose union is $\Sigma(M_\parallel)$ and such that $(\theta \lceil M_i, F_i) \in \mathit{Fail}(M_i)$ for each i. From Lemma 1, there are sets of events $F'_1, \ldots, F'_n$ with each $F'_i \supseteq F_i$ and such that $(\theta \lceil M_i, F'_i) \in \mathit{AbsFail}(M_i/{\approx_i})$ for each i. Since $\theta \lceil M_i = \theta \lceil M_i/{\approx_i}$, we can invoke Lemma 2 to conclude that $(\theta, \Sigma(\widehat{M_\parallel})) \in \mathit{Fail}(\widehat{M_\parallel})$. $\square$

Note that $\widehat{M_\parallel}$ has an abstract deadlock iff it has a finite path that reaches a state whose abstract refusal is the whole of $\Sigma(\widehat{M_\parallel})$. We call such a path a counterexample to abstract deadlock-freedom, or simply an abstract counterexample. One must then determine whether this abstract counterexample is valid or not, i.e., whether it gives rise to a genuine deadlock in the concrete system or not.

Suppose that our abstract counterexample is $\pi = \langle \alpha_1, a_1, \alpha_2, a_2, \ldots, a_m, \alpha_{m+1} \rangle$, where we write $\alpha_{m+1} = (\alpha^1_{m+1}, \ldots, \alpha^n_{m+1})$. Let $\theta = \langle a_1, a_2, \ldots, a_m \rangle$ be the trace associated with π. Then for each i, we have that $(\theta \lceil M_i/{\approx_i}, \mathit{AbsRef}(\alpha^i_{m+1}))$ is an abstract failure of $M_i/{\approx_i}$. If it turns out also to be a genuine failure of M_i, for each M_i, then we say that the abstract counterexample is *valid*. Indeed we can conclude, thanks to Theorem 6(3), that $M_\parallel$ has a genuine deadlock, which moreover we can readily provide a witness for.

Efficient algorithms for checking whether an LTS has a given failure are well-known—see, e.g., [Ros97]. Note that the validation is performed compositionally, one component (M_i) at a time.

Unfortunately, we may instead discover some M_i for which $(\theta \lceil M_i, \mathit{AbsRef}(\alpha^i_{m+1}))$ is *not* a failure. In that case, we must refine our partition $\approx_i$, so as to rule out the spurious abstract failure, and start the search anew. Note once again that termination will follow automatically provided that each partition refinement is strict.[11]

The refinement step proceeds as follows. We are assuming that either (i) the path $\pi^i = \langle \alpha^i_1, a_1, \alpha^i_2, a_2, \ldots, a_m, \alpha^i_{m+1} \rangle$ of $M_i/{\approx_i}$ cannot be matched by M_i, or (ii) that it can be matched but that none of the states s it leads to has $\mathit{Ref}(s) = \mathit{AbsRef}(\alpha^i_{m+1})$. In case (i), we employ exactly the same technique as that of Section 6 to refine the partition $\approx_i$ by splitting one of the intermediate abstract states $\alpha^i_1, \ldots, \alpha^i_m$ in two. In case (ii), we observe that the states in α^i_{m+1} that are reachable in M_i along π_i cannot all have identical refusals. Pick some event a for which at least two states disagree, and split α_{m+1} into those states that refuse a and those that do not. This clearly gives rise to a strict refinement of the partition $\approx_i$, and moreover permanently rules out the spurious abstract counterexample π.

Let us refer to this procedure as *Deadlock_Validate/Refine*$(M_i, \approx_i, \pi_i)$. The full algorithm for checking whether $M_1 || \ldots || M_n$ has a deadlock is given in Figure 8. Note that the abstraction, counterexample-validation, and refinement steps are all performed one component at a time.

[11] In fact, it turns out that the abstract LTSs converge to the bisimulation quotients of their concrete counterparts; however in practice deadlock-freedom is often established or disproved well before the bisimulation quotient is achieved.

Algorithm *Deadlock_Detection* $(M_1, \ldots, M_n)$
for $i := 1$ to n : **let** $\approx_i :=$ the coarsest partition, with all states in the same equivalence class;
repeat forever
  find abstract deadlock in $M_1/\approx_1 \parallel \ldots \parallel M_n/\approx_n$;
  if there is no abstract deadlock **then**
    return "$M_1 \parallel \ldots \parallel M_n$ `is deadlock-free`"
  else suppose $\pi \in \mathit{FPath}(M_1/\approx_1 \parallel \ldots \parallel M_n/\approx_n)$ leads to abstract deadlock;
  find i such that $\mathit{Deadlock\_Validate/Refine}(M_i, \approx_i, \pi \restriction M_i/\approx_i)$ reports "`spurious`";
  if no such i **then**
    return "$M_1 \parallel \ldots \parallel M_n$ `has deadlock`" along with counterexample derived from π
  else refine $\approx_i$ to rule out spurious counterexample $\pi \restriction M_i/\approx_i$;
endrepeat

Fig. 8. The overall deadlock-detection algorithm for a concurrent system $M_1 \parallel \ldots \parallel M_n$.

10. Experimental Results

We implemented our algorithms within the MAGIC tool. MAGIC extracts finite LKS models from C programs using predicate abstraction. These LKSs are then analyzed to check for satisfaction of an SE-LTL specification or the presence of deadlock using the techniques presented in this article. Once a real counterexample π is found at the level of the LKSs MAGIC analyzes π and, if necessary, creates more refined models by inferring new predicates[12]. Our actual implementation is therefore a two-level CEGAR scheme. We elide details of the outer predicate abstraction-refinement loop as it is similar to some of our previous work [COYC03]. All our experiments were performed on an AMD Athlon XP 1600+ machine with 900 MB RAM running RedHat Linux 7.1.

10.1. SE-LTL Experiments

In order to validate our approach for SE-LTL model checking we experimented with two broad sets of benchmarks. The first set of these examples was 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 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 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 9 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 was obtained from the source code of version 2.0 of Micro-C/OS. This is a popular, lightweight, real-time, multi-tasking operating system written in about 3000 lines of ANSI C.

[12] This is akin to the splitting of abstract states described in the paper, carried out symbolically via the use of predicates on the state space of the C program.

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. 9. Experimental results with OpenSSL and Micro-C/OS. These experiments did not make use of abstraction/refinement and proceeded in a single iteration. **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.

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 had 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 9 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 9.

Since our main goal in these experiments was to directly compare the state/event-based and pure-state-based approaches, we did not make use of any abstraction and refinement at the level of the LKSs and the verification therefore proceeded in a single iteration.

10.2. Deadlock Experiments

Our approach for deadlock detection was validated against a broad set of benchmarks consisting of both industrial and classical systems. Our results are summarized in Figure 10. The *ABB* benchmark was provided to us by our industrial partner, ABB Corporation [ABB]. It implements part of an inter-process communication protocol (IPC-1.6) used to mediate communication in a multi-threaded robotics control automation system developed by ABB. The implementation is required to satisfy various safety-critical properties, and

Name	*Plain*					*IterDeadlock*				
	S_M	S_R	*I*	*T*	*M*	S_M	S_R	*I*	*T*	*M*
ABB	2.1×10^9	*	*	*	162	4.1×10^5	1973	861	**1446**	**33.3**
SSL	49405	25731	1	44	43.5	16	16	16	31.9	40.8
UCOSD-2	1.1×10^5	5851	5	24	14.5	374	261	77	14.5	12.9
UCOSD-3	2.1×10^7	*	*	*	58.6	6144	4930	120	**221.8**	**15**
UCOSN-4	1.9×10^7	39262	1	18.1	14.1	8192	2125	30	8.1	10.5
UCOSN-5	9.4×10^8	4.2×10^5	1	253	52.2	65536	12500	37	80	**12.7**
UCOSN-6	4.7×10^{10}	*	*	*	219.3	5.2×10^5	71875	44	**813**	**30.8**
RW-4	1.3×10^9	8369	4	6.48	10.8	5120	67	54	4.40	10.0
RW-5	9.0×10^{10}	54369	4	35.1	15.9	24576	132	60	7.33	10.4
RW-6	5.8×10^{12}	3.5×10^5	4	257	45.2	1.1×10^5	261	66	**12.6**	**10.8**
RW-7	1.5×10^{14}	*	*	*	178	5.2×10^5	518	72	**25.3**	**11.8**
RW-8	*	*	*	*	*	2.4×10^6	1031	78	60.5	14.0
RW-9	*	*	*	*	*	1.7×10^7	2056	84	132	14.5
DPN-3	3.6×10^7	1401	2	.779	-	5832	182	27	.849	-
DPN-4	1.1×10^{10}	16277	2	11.8	10.9	1.0×10^5	1274	34	7.86	9.5
DPN-5	3.2×10^{12}	1.9×10^5	2	197	28.0	1.9×10^6	8918	41	84.6	11.4
DPN-6	9.7×10^{14}	*	*	*	203	3.4×10^7	62426	48	**831**	**26.1**
DPD-9	3.5×10^{22}	11278	1	22.5	12.0	5.2×10^9	13069	46	191	12.2
DPD-10	1.1×10^{25}	38268	1	87.6	17.3	6.2×10^{10}	44493	51	755	18.4

Fig. 10. Experimental results. S_M = maximum # of states; S_R = # of reachable states; I = # of iterations; T = time in seconds; M = memory in MB; time limit = 1500 sec; - indicates negligible value; * indicates out of time; notable figures are highlighted.

in particular deadlock-freedom. The IPC protocol supports multiple modes of communication, including synchronous point-to-point, broadcast, publish/subscribe, and asynchronous communication. Each of these modes is implemented in terms of messages passed between queues owned by different threads. The protocol handles the creation and manipulation of message queues, synchronizing access to shared data using various operating system primitives (e.g., semaphores), and cleaning up internal states when a communication fails or times out.

We analyzed the portion of the IPC protocol that implements the primitives for synchronous communication (approx. 1500 LOC) among multiple threads. With this type of communication, a sender sends a message to a receiver and blocks until an answer is received or it times out. A receiver asks for its next message and blocks until a message is available or it times out. Whenever the receiver gets a synchronous message, it is then expected to send a response to the sender. MAGIC successfully verified the absence of deadlock in this implementation.

The *SSL* benchmark represents a deadlock-free system (approx. 700 LOC) consisting of one OpenSSL server and one OpenSSL client. The *UCOSD-n* benchmarks are derived from Micro-C/OS version 2.7, a real-time operating system for embedded processors, and consist of n threads (approx. 6000 LOC) executing concurrently. Access to shared data is protected via locks. This implementation suffers from deadlock. In contrast, the *UCOSN-n* benchmarks are deadlock-free. The *RW-n* benchmarks implement a deadlock-free reader-writer system (194 LOC) with n readers, n writers, and a controller. The controller ensures that at most one writer has access to the critical section. Finally, the *DPN-n* benchmarks represent a deadlock-free implementation of n dining philosophers (251 LOC), while *DPD-n* implements n dining philosophers (163 LOC) that can deadlock. As Figure 10 shows, even though the implementations are of moderate size, the total state space is often quite large due to exponential blowup.

Values under *IterDeadlock* refer to measurements for our approach while those under *Plain* correspond to a naive approach involving only predicate abstraction refinement. We note that *IterDeadlock* outperforms *Plain* in almost all cases. In many cases *IterDeadlock* is able to establish deadlock or deadlock-freedom while

Plain runs out of time. Even when both approaches succeed, *IterDeadlock* can yield over 20 times speed-up in time and require over 4 times less memory (*RW-6*). For the experiments involving dining philosophers with deadlock however, *Plain* performs better than *IterDeadlock*. This is because in these cases *Plain* terminates as soon as it discovers a deadlocking scenario, without having to explore the entire state space. In contrast, *IterDeadlock* has to perform many iterations before finding a genuine deadlock.

Finally, we note that in various instances one observes a rather large number of abstraction/refinement iterations over a relatively small state space. This occurs because our initial abstraction is very small (a single state) and the successive refinement steps are designed to minimize the increase of the state space. We expect that different choices of initial abstraction and a more aggressive refinement strategy would lead to fewer iterations. The approach we followed is chiefly aimed at mitigating the state-space explosion problem.

11. Conclusion and Future Work

In this paper, we have presented an expressive framework for modelling 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 within our C model checker MAGIC. We have also carried out a number of experiments on industrial benchmarks, and have discovered bugs in the real-time operating system Micro-C/OS. 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 yields substantial improvements in both verification time and memory usage.

We have also presented a new algorithm to detect deadlocks in concurrent blocking message-passing programs. This algorithm not only complements our state/event verification framework, but is also highly useful in its own right. Once again, the strength of our approach resides in the use of efficient abstractions as well as compositional reasoning, despite the fact that deadlock is non-compositional and moreover is not preserved by classical abstractions. Our technique is automated and employs iterative abstraction refinement to scale to real-life examples. Experimental results demonstrate the effectiveness of our approach on industrial benchmarks. We believe it can be improved further by using assume-guarantee style reasoning, and we plan to investigate this issue in the future.

There remain many other avenues for further research. 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 *labelled* 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. 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. Finally, an interesting area of research is to develop mechanisms to handle shared variables compositionally.

References

[ABB] ABB website. `http://www.abb.com`.

[ACFM85] T. S. Anantharaman, E. M. Clarke, M. J. Foster, and B. Mishra. Compiling path expressions into VLSI circuits. In *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 191–204. ACM Press, 1985.

[AQR+04] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV)*, volume 3114 of *Lecture Notes in Computer Science*, pages 484–487. Springer, 2004.

[BLAST] 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 the 10th International Conference on Computer Aided Verification (CAV)*, volume 1427 of *Lecture Notes in Computer Science*, pages 319–331. Springer, 1998.

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

[BR91] S. D. Brookes and A. W. Roscoe. Deadlock analysis of networks of communicating processes. *Distributed Computing*, 4:209–230, 1991.

[BR01] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th International SPIN Workshop*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122. Springer, 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.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the SIGPLAN Conference on Programming Languages*, 1977.

[CCG+03] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pages 385–395. IEEE Press, 2003.

[CCG+04] S. Chaki, E. Clarke, O. Grumberg, J. Ouaknine, N. Sharygina, T. Touili, and H. Veith. An expressive verification framework for state/event systems. Technical Report CMU-CS-04-145, Computer Science Department, Carnegie Mellon University, 2004.

[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 4th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, volume 2517 of *Lecture Notes in Computer Science*, pages 33–51. Springer, 2002.

[CCO+04] S. Chaki, E. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. State/event-based software model checking. In *Proceedings of the 4th International Conference on Integrated Formal Methods (IFM)*, volume 1999 of *Lecture Notes in Computer Science*, pages 128–147. Springer, 2004.

[CCOS04] S. Chaki, E. M. Clarke, J. Ouaknine, and N. Sharygina. Automated, compositional and iterative deadlock detection. In *Proceedings of the 2nd ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. IEEE Press, 2004.

[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 the 22nd International Conference on Software Engineering (ICSE)*, pages 439–448. ACM Press, 2000.

[CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings of the Workshop on Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 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 *Proceedings of the 12th International Conference on Computer Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[CGKS02] E. M. Clarke, A. Gupta, J. H. Kukula, and O. Strichman. SAT based abstraction-refinement using ILP and machine learning techniques. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 265–279. Springer, 2002.

[CGL94] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

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

[CGP03] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. Learning assumptions for compositional verification. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346. Springer, 2003.

[Cor96] James C. Corbett. Evaluating deadlock detection methods for concurrent software. *Software Engineering*, 22(3):161–180, 1996.

[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 the Workshop on Software Model Checking (SoftMC)*. 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.

[DIS99] C. Demartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent Java programs. *Software: Practice & Experience*, 29(7):577–603, 1999.

[FHRR04] C. Fournet, C. A. R. Hoare, S. K. Rajamani, and J. Rehof. Stuck-free conformance. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV)*, volume 3114 of *Lecture Notes in Computer Science*, pages 242–254. Springer, 2004.

[FSE] Formal Systems (Europe) Ltd. website. `http://www.fsel.com`.

[GL94] O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Transactions 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 the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, pages 257–266. 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 *Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification*, pages 3–18. Chapman & Hall, 1995.

[GS97] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.

[HJMQ03] T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV)*, volume 2725 of *Lecture Notes in Computer Science*, pages 262–274. Springer, 2003.

[HJMS02] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 58–70. ACM Press, 2002.

[HJS01] M. Huth, R. Jagadeesan, and D. Schmidt. Modal transition systems: A foundation for three-valued program analysis. In *Proceedings of the 10th European Symposium on Programming (ESOP)*, volume 2028 of *Lecture Notes in Computer Science*, pages 137–154. Springer, 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 the International Conference on Computer-Aided Design (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.

[Kur89] R. P. Kurshan. Analysis of discrete event coordination. In *Proceedings of the REX Workshop*, volume 430 of *Lecture Notes in Computer Science*, pages 414–453. Springer, 1989.

[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 the 19th International Conference on the Application and Theory of Petri Nets (ICATPN)*, volume 1420 of *Lecture Notes in Computer Science*, pages 365–383. Springer, 1998.

[LBBO01] Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental verification by abstraction. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *Lecture Notes in Computer Science*, pages 98–112. Springer, 2001.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 97–107. ACM Press, 1985.

[LTSA] LTSA website. `http://www-dse.doc.ic.ac.uk/concurrency/ltsa/LTSA.html`.

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

[McM97] K. L. McMillan. A compositional rule for hardware design refinement. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV)*, volume 1254 of *Lecture Notes in Computer Science*, pages 24–35. Springer, 1997.

[MH00] J. M. R. Martin and Y. Huddart. Parallel algorithms for deadlock and livelock analysis of concurrent systems. In *Proceedings of Comm. Process Architectures*, 2000.

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

[MJ97] J. M. R. Martin and S. Jassim. A tool for proving deadlock freedom. In *Proceedings of the 20th World Occam and Transputer User Group Technical Meeting*, 1997.

[NCOD97] G. Naumovich, L. A. Clarke, L. J. Osterweil, and M. B. Dwyer. Verification of concurrent software with FLAVERS. In *Proceedings of the 19th International Conference on Software Engineering (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*, 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 the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *Lecture Notes in Computer Science*, pages 284–298. Springer, 2001.

[Pnu86] A. Pnueli. Application of temporal logic to the specification and verification of reactive systems: A survey of current trends. In *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 *Selected Papers from the First and the Second European Workshop on Application and Theory of Petri Nets*, pages 337–350. Springer, 1981.

[RD87] A. W. Roscoe and N. Dathi. The pursuit of deadlock freedom. *Information and Computation*, 75(3):289–327, 1987.

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

[SB00] F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science*, pages 248–263. Springer, 2000.

[SLAM] 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.

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