Automated, Compositional and Iterative Deadlock Detection*

Sagar Chaki Edmund Clarke Joël Ouaknine Natasha Sharygina
Carnegie Mellon University, Pittsburgh, USA
{chaki|emc|ouaknine|natalie}@cs.cmu.edu

Abstract

We present an algorithm to detect deadlocks in concurrent message-passing programs. Even though deadlock is inherently non-compositional and its absence is not preserved by standard abstractions, our framework employs both ***abstraction*** *and* ***compositional reasoning*** *to alleviate the state space explosion problem. We iteratively construct increasingly more precise abstractions on the basis of spurious counterexamples to either detect a deadlock or prove that no deadlock exists. Our approach is inspired by the counterexample-guided abstraction refinement paradigm. However, our notion of abstraction as well as our schemes for verification and abstraction refinement differ in key respects from existing abstraction refinement frameworks. Our algorithm is also compositional in that abstraction, counterexample validation, and refinement are all carried out component-wise and do not require the construction of the complete state space of the concrete system under consideration. Finally, our approach is completely* ***automated*** *and provides diagnostic feedback in case a deadlock is detected. We have implemented our technique in the* MAGIC *verification tool and present encouraging results (up to 20 times speed-up in time and 4 times less memory consumption) with concurrent message-passing C programs. We also report a bug in the real-time operating system MicroC/OS version 2.70.*

1. Introduction

Ensuring that standard software components are assembled in a way that guarantees the delivery of reliable services is an important task for system designers. Certifying the absence of deadlock in a composite system is an example of a stringent requirement that has to be satisfied before the system can be deployed in real life. This is especially true for safety-critical systems, such as embedded systems or plant controllers, that are expected to always service requests within a fixed time limit or be responsive to external stimuli. Moreover, in case a deadlock is detected, it is highly desirable to be able to provide system designers and implementers with appropriate diagnostic feedback.

However, despite significant efforts, validating the absence of deadlock in systems of realistic complexity remains a major challenge. The problem is especially acute in the context of concurrent programs that communicate via mechanisms with blocking semantics, e.g., synchronous message-passing and semaphores. The primary obstacle is the well-known *state space explosion* problem whereby the size of the state space of a concurrent system increases exponentially with the number of components. Two paradigms are usually recognized as being the most effective against the state space explosion problem: *abstraction* and *compositional reasoning*. Even though these two approaches have been widely studied in the context of formal verification [17, 11, 27, 19], they find much less use in deadlock detection. This is possibly a consequence of the fact that deadlock is inherently non-compositional and its absence is not preserved by standard abstractions (see Example 3). Therefore, a compositional and abstraction-based deadlock detection scheme, such as the one we present in this article, is especially significant.

Counterexample-guided abstraction refinement [22] (CEGAR for short) is a methodology that uses abstraction in an automated manner and has been successful in verifying real-life hardware [10] and software [3] systems. A CEGAR-based scheme iteratively computes more and more precise abstractions (starting with a very coarse one) of a target system on the basis of spurious counterexamples until a real counterexample is obtained or the system is found to be correct. The approach presented in this article combines both abstraction and compositional reasoning within a CEGAR-based framework for verifying the ab-

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

sence of deadlocks in concurrent message-passing systems. More precisely, suppose we have a system M composed of components $M_1, \ldots, M_n$ executing concurrently. Then our technique checks for deadlock in M using the following three-step iterative process:

1. **Abstract.** Create an abstraction $\widehat{M}$ such that if M has a deadlock, then so does $\widehat{M}$. This is done component-wise without having to construct the full state space of M.

2. **Verify.** Check if $\widehat{M}$ has a deadlock. If not, report absence of deadlock in M and exit. Otherwise let π be a counterexample that leads to a deadlock in $\widehat{M}$.

3. **Refine.** Check if π corresponds to a deadlock in M. Once again this is achieved component-wise. If π corresponds to a real deadlock, report presence of deadlock in M along with counterexample derived from π and exit. Otherwise refine $\widehat{M}$ on the basis of π to obtain a more precise abstraction and repeat from step 1.

In our approach, systems as well as their components are represented as finite Labeled Transition Systems (LTSs), a form of state machines. Note that only the verification stage (step 2) of our technique requires explicit composition of systems. All other stages can be performed one component at a time. Since verification is performed only on abstractions (which are usually much smaller than the corresponding concrete systems), this technique is able to significantly reduce the state space explosion problem. Finally, when a deadlock is detected, our scheme provides useful diagnostic feedback in the form of counterexamples.

To the best of our knowledge, this is the first counterexample-guided, compositional abstraction refinement scheme to perform deadlock detection on concurrent systems. We have implemented our approach in our C verification tool MAGIC [24] which extracts LTS models from C programs automatically via predicate abstraction [34, 6]. Our experiments with a variety of benchmarks have yielded encouraging results (up to 20 times speed-up in time and 4 times less memory consumption). We have also discovered a bug in the real-time operating system MicroC/OS version 2.70.

The rest of this article is organized as follows. In Section 2 we summarize related work. This is followed by some preliminary definitions and results in Section 3. In Section 4 we present our abstraction scheme, followed by counterexample validation and abstraction refinement in Section 5 and Section 6 respectively. Our overall deadlock detection algorithm is described in Section 7. Finally, we present experimental results in Section 8 and conclude in Section 9.

2. Related Work

The formalization of a general notion of abstraction first appeared in [14]. The abstractions used in our approach are *conservative*. They are only guaranteed to preserve 'undesirable' properties of the system (e.g., [21, 11]). Conservative abstractions usually lead to significant reductions in the state space but in general require an iterated abstraction refinement mechanism (such as CEGAR) in order to establish specification satisfaction. CEGAR [22, 10] 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 [29] (in non-automated form), and [3, 31, 23, 18, 8, 12].

CEGAR-based schemes have been used for the verification of both safety [3, 10, 18, 6] (i.e., reachability) and liveness [5] properties. Compositionality has been most extensively studied in process algebra (e.g., [20, 28, 32]), particularly in conjunction with abstraction. Abstraction and compositional reasoning have been combined [7] within a single two-level CEGAR scheme to verify safety properties of concurrent message-passing C programs. None of these techniques attempt to detect deadlock. In fact, the abstractions used in these schemes do not preserve deadlock freedom and hence cannot be used directly in our approach.

Deadlock detection has been widely studied in various contexts. One of the earliest deadlock-detection tools, for the process algebra CSP, was FDR [16]; see also [33, 4, 26, 32, 25]. Corbett has evaluated various deadlock-detection methods for concurrent systems [13] while Demartini et al. have developed deadlock-detection tools for concurrent Java programs [15]. However, to the best of our knowledge, none of these approaches involve abstraction refinement or compositionality in automated form.

3. Background

In this section, we present some preliminary definitions and results (many of which originate from CSP [20, 32]) that are used in the rest of the article.

Definition 1 (Labeled Transition System) *A Labeled Transition System (LTS) is a quadruple $(S, init, \Sigma, T)$ such that: (i) S is a finite non-empty set of states, (ii) $init \in S$ is an initial state, (iii) Σ is a finite set of actions (alphabet) and (iv) $T \subseteq S \times \Sigma \times S$ is a transition relation.*

Given an LTS $M = (S, init, \Sigma, T)$, we write $S(M)$ and $\Sigma(M)$ to mean S and Σ respectively. We also write $s \xrightarrow{a} s'$ to mean $(s, a, s') \in T$. If $s \xrightarrow{a} s'$ we say that there exists a transition from s to s' labeled by a. The successor function $Succ : S(M) \times \Sigma(M) \rightarrow 2^{S(M)}$ is defined as:

$Succ(s,a) = \{s' \mid s \stackrel{a}{\to} s'\}$. In the remainder of this article, we use $\langle x, y, \ldots\rangle$ to denote sequences and $\frown$ to denote concatenation of sequences. Our notions of paths and traces are standard and are presented next.

Definition 2 (Path) *A path of an LTS M is a finite non-empty sequence $\langle s_0, a_0, s_1, a_1, \ldots, a_{n-1}, s_n\rangle$ such that: (i) $s_0 = init$ and (ii) for $0 \le i < n$, $s_i \stackrel{a_i}{\to} s_{i+1}$. We write $Path(M)$ to denote the set of all paths of M.*

Definition 3 (Trace) *Let M be an LTS. A finite sequence $\langle a_0, \ldots, a_{n-1}\rangle \in \Sigma(M)^*$ is a trace of M iff there exist $s_0, s_1, \ldots, s_n \in S(M)$ such that $\langle s_0, a_0, s_1, a_1, \ldots, a_{n-1}, s_n\rangle \in Path(M)$.*

Paths and traces are usually represented with the letters π and θ respectively.

A state s is said to *refuse* an action a iff there is no transition from s labeled by a. The *refusal* of a state is the set of all actions that it refuses. Suppose θ is a sequence of actions and F is a set of actions. Then (θ, F) is said to be a *failure* of an LTS M iff M can participate in the sequence of actions θ (i.e., θ is a trace of M) and then reach a state whose refusal is F. Finally, M has a *deadlock* iff it can reach a state which refuses the entire alphabet $\Sigma(M)$. We now present these notions formally.

Definition 4 (Refusal) *Let M be an LTS and $s \in S(M)$. Then $Ref(s) = \{a \in \Sigma(M) \mid \forall s' \in S(M) \centerdot s \stackrel{a}{\nrightarrow} s'\}$.*

Definition 5 (Failure) *Let M be an LTS. A pair $(\theta, F) \in \Sigma(M)^* \times 2^{\Sigma(M)}$ is a failure of M iff the following condition holds: writing $\theta = \langle a_0, \ldots, a_{n-1}\rangle$, there exist $s_0, s_1, \ldots, s_n$ such that (i) $\langle s_0, a_0, s_1, a_1, \ldots, a_{n-1}, s_n\rangle \in Path(M)$ and (ii) $F = Ref(s_n)$. We write $Fail(M)$ to denote the set of all failures of M.*

Definition 6 (Deadlock) *An LTS M is said to have a deadlock iff $(\theta, \Sigma(M)) \in Fail(M)$ for some $\theta \in \Sigma(M)^*$.*

Example 1 *Figure 1 shows two LTSs M_1 and M_2. 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 parallel composition is central to our approach. We assume that when several components are executed concurrently, they synchronize on shared actions and proceed independently on local actions. This notion of parallel composition has been used in, e.g., CSP [20, 32], and by Anantharaman et al. [2].

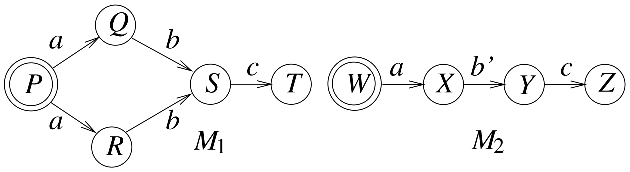

Figure 1. Two sample LTSs M_1 and M_2. Initial states are doubly circled.

Definition 7 (Parallel Composition) *Let $M_1 = (S_1, init_1, \Sigma_1, T_1)$, $\ldots$, $M_n = (S_n, init_n, \Sigma_n, T_n)$ be LTSs. Then their parallel composition, denoted by $M_1 \parallel \ldots \parallel M_n$, is the LTS $(S_\parallel, init_\parallel, \Sigma_\parallel, T_\parallel)$ such that (i) $S_\parallel = S_1 \times \ldots \times S_n$, (ii) $init_\parallel = (init_1, \ldots, init_n)$, (iii) $\Sigma_\parallel = \bigcup_{i=1}^{n} \Sigma_i$, and (iv) $(s_1, \ldots, s_n) \stackrel{a}{\to} (s'_1, \ldots, s'_n)$ iff for $1 \le i \le n$ the following condition holds: if $a \in \Sigma_i$ then $(s_i, a, s'_i) \in T_i$, and otherwise $s_i = s'_i$.*

Example 2 *Figure 2 shows the LTS $M_1 \parallel M_2$ where M_1 and M_2 are the LTSs shown in Figure 1.*

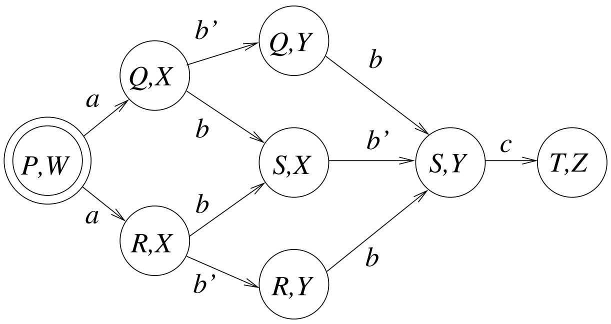

Figure 2. Parallel composition of LTSs M_1 and M_2 from Figure 1.

Given a trace of a concurrent system $M_\parallel$, one can construct *projections* by restricting the trace to the alphabets of each of the components of $M_\parallel$.

Definition 8 (Projection) *Consider LTSs $M_1, \ldots, M_n$. Let $M_\parallel = M_1 \parallel \ldots \parallel M_n$. For $1 \le i \le n$, the projection function $Proj_i : \Sigma(M_\parallel)^* \to \Sigma(M_i)^*$ is defined inductively as follows (we write $\theta \downarrow i$ to mean $Proj_i(\theta)$):*

1. *$\langle\rangle \downarrow i = \langle\rangle$.*

2. *If $a \in \Sigma(M_i)$ then $(\langle a\rangle \frown \theta) \downarrow i = \langle a\rangle \frown (\theta \downarrow i)$.*

3. *If $a \notin \Sigma(M_i)$ then $(\langle a\rangle \frown \theta) \downarrow i = \theta \downarrow i$.*

Definitions 7 and 8 immediately lead to the following lemma, which essentially highlights the compositional nature of failures. Its proof, as well as the proofs of related results, are well-known [32].

Lemma 1 *Let $M_1, \ldots, M_n$ be LTSs. Then $(\theta, F) \in Fail(M_1 \parallel \ldots \parallel M_n)$ iff there exist $F_1, \ldots, F_n$ such that: (i) $F = \bigcup_{i=1}^{n} F_i$, and (ii) for $1 \le i \le n$, $(\theta \downarrow i, F_i) \in Fail(M_i)$.*

4. Abstraction

In this section we present our notion of abstraction. Our framework employs *quotient LTSs* as abstractions of concrete LTSs. Given a concrete LTS M, one can obtain a quotient LTS as follows. The states of the quotient LTS are obtained by lumping together states of M; alternatively, one can view these lumps as equivalence classes of some equivalence relation on $S(M)$. Transitions of the quotient LTS are defined *existentially*.

Definition 9 (Quotient LTS) *Let $M = (S, init, \Sigma, T)$ be an LTS and $R \subseteq S \times S$ an equivalence relation. For an arbitrary $s \in S$ we let $[s]^R$ denote the equivalence class of s. M and R then induce a quotient LTS $M^R = (S^R, init^R, \Sigma^R, T^R)$ where: (i) $S^R = \{[s]^R \mid s \in S\}$, (ii) $init^R = [init]^R$, (iii) $\Sigma^R = \Sigma$, and (iv) $T^R = \{([s]^R, a, [s']^R) \mid (s, a, s') \in T\}$.*

We write $[s]$ to mean $[s]^R$ when R is clear from the context. M^R is often called an *existential abstraction* of M. The states of M are referred to as *concrete states* while those of M^R are called *abstract states*. We use the Greek letter α to represent abstract states, and continue to denote concrete states with the Roman letter s.

Quotient LTSs have been studied in the verification literature. In particular, the following result is well-known [9].

Proposition 1 *Let M be an LTS, R an equivalence relation on $S(M)$, and M^R the quotient LTS induced by M and R. If $\langle s_0, a_0, s_1, a_1, \ldots, a_{n-1}, s_n \rangle \in Path(M)$, then $\langle [s_0], a_0, [s_1], a_1, \ldots, a_{n-1}, [s_n] \rangle \in Path(M^R)$.*

Example 3 *Note the following facts about the LTSs in Figure 3: (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; (v) M_1 has a deadlock but the quotient LTS obtained by lumping all the states of M_1 into a single equivalence class does not have a deadlock.*

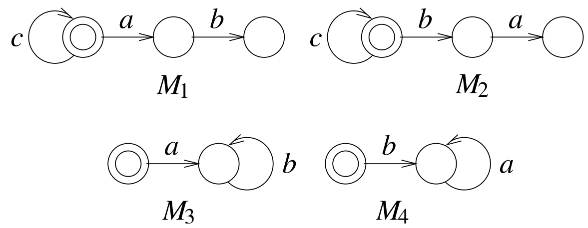

Figure 3. Four sample LTSs. Initial states are doubly circled.

As Example 3 highlights, deadlock is non-compositional and its absence is not preserved by existential abstractions (nor is it preserved by *universal* abstractions). So far we have presented well-known definitions and results to prepare the background. We now present what constitute the core technical contributions of this article.

We begin by taking a closer look at the non-preservation of deadlock by existential abstractions. Consider a quotient LTS M^R and a state $[s]$ of M^R. It can be proved that $Ref([s]) = \bigcap_{s' \in [s]} Ref(s')$. In other words, the refusal of an abstract state $[s]$ under-approximates the refusals of the corresponding concrete states. However, in order to preserve deadlock we require that refusals be *over-approximated*. We achieve this by taking the union of the refusals of the concrete states. This leads to the notion of an *abstract refusal*, which we now define formally.

Definition 10 (Abstract Refusal) *Let M be an LTS, $R \subseteq S(M) \times S(M)$ an equivalence relation, and M^R the quotient LTS induced by M and R. Then the abstract refusal function $AbsRef : S(M^R) \to 2^{\Sigma(M^R)}$ is defined as follows:*

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

For a parallel composition of quotient LTSs, we extend the notion of abstract refusal as follows. Let $M_1^{R_1}, \ldots, M_n^{R_n}$ be quotient LTSs. Let $\alpha = (\alpha_1, \ldots, \alpha_n) \in S(M_1^{R_1} \parallel \ldots \parallel M_n^{R_n})$. Then $AbsRef(\alpha) = \bigcup_{i=1}^{n} AbsRef(\alpha_i)$.

Next, we introduce the notion of *abstract failures*, which are similar to failures, except that refusals are replaced by abstract refusals.

Definition 11 (Abstract Failure) *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). A pair $(\theta, F) \in \Sigma(\widehat{M})^* \times 2^{\Sigma(\widehat{M})}$ is said to be an abstract failure of $\widehat{M}$ iff the following condition holds: writing $\theta = \langle a_0, \ldots, a_{n-1} \rangle$, there exist $\alpha_0, \alpha_1, \ldots, \alpha_n$ such that (i) $\langle \alpha_0, a_0, \alpha_1, a_1, \ldots, a_{n-1}, \alpha_n \rangle \in Path(\widehat{M})$ and (ii) $F =$*

$AbsRef(\alpha_n)$. We write $AbsFail(\widehat{M})$ 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^R.

Lemma 2 *Let M be an LTS, $R \subseteq S(M) \times S(M)$ an equivalence relation, and M^R the quotient LTS induced by M and R. Then for all $(\theta, F) \in Fail(M)$, there exists $F' \supseteq F$ such that $(\theta, F') \in AbsFail(M^R)$.*

Proof 1 *1. From $(\theta, F) \in Fail(M)$ and Definition 5: let $\theta = \langle a_0, \ldots, a_{n-1}\rangle$ and $\langle s_0, a_0, s_1, a_1, \ldots, a_{n-1}, s_n\rangle \in Path(M)$ such that $F = Ref(s_n)$.*

2. *From 1 and Proposition 1: $\langle [s_0], a_0, [s_1], a_1, \ldots, a_{n-1}, [s_n]\rangle \in Path(M^R)$.*

3. *From 2 and Definition 11: $(\theta, AbsRef([s_n])) \in AbsFail(M^R)$.*

4. *From Definition 10: $AbsRef([s_n]) \supseteq Ref(s_n)$.*

5. *From 3, 4 and using $F' = AbsRef([s_n])$ we get our result.*

□

As the following two lemmas show, abstract failures are compositional: the abstract failures of a concurrent system $M_\parallel$ can be decomposed naturally into abstract failures of the components of $M_\parallel$. Proofs of Lemmas 3 and 4 follow the same lines as Lemma 1.

Lemma 3 *Let $M_1^{R_1}, \ldots, M_n^{R_n}$ be quotient LTSs, and $\langle \alpha_0, a_0, \ldots, a_{k-1}, \alpha_k\rangle \in Path(M_1^{R_1} \parallel \ldots \parallel M_n^{R_n})$. Let $\theta = \langle a_0, \ldots, a_{k-1}\rangle$ and $\alpha_k = (\alpha_k^1, \ldots, \alpha_k^n)$. Then for $1 \le i \le n$, $(\theta \downarrow i, AbsRef(\alpha_k^i)) \in AbsFail(M_i^{R_i})$.*

Lemma 4 *Let $M_1^{R_1}, \ldots, M_n^{R_n}$ be quotient LTSs. Then $(\theta, F) \in AbsFail(M_1^{R_1} \parallel \ldots \parallel M_n^{R_n})$ iff there exist $F_1, \ldots, F_n$ such that: (i) $F = \bigcup_{i=1}^n F_i$, and (ii) for $1 \le i \le n$, $(\theta \downarrow i, F_i) \in AbsFail(M_i^{R_i})$.*

In the rest of this article we often make implicit use of the following facts. Let $M_1^{R_1}, \ldots, M_n^{R_n}$ be quotient LTSs. Then $\Sigma(M_1^{R_1} \parallel \ldots \parallel M_n^{R_n}) = \bigcup_{i=1}^n \Sigma(M_i^{R_i}) = \bigcup_{i=1}^n \Sigma(M_i) = \Sigma(M_1 \parallel \ldots \parallel M_n)$.

The notion of abstract failures leads naturally to the notion of *abstract deadlocks*.

Definition 12 (Abstract Deadlock) *Let $M_1^{R_1}, \ldots, M_n^{R_n}$ be quotient LTSs and $\widehat{M}_\parallel = M_1^{R_1} \parallel \ldots \parallel M_n^{R_n}$. $\widehat{M}_\parallel$ is said to have an abstract deadlock iff $(\theta, \Sigma(\widehat{M}_\parallel)) \in AbsFail(\widehat{M}_\parallel)$ for some $\theta \in \Sigma(\widehat{M}_\parallel)^*$.*

Let $M_1^{R_1}, \ldots, M_n^{R_n}$ be quotient LTSs and $\widehat{M}_\parallel = M_1^{R_1} \parallel \ldots \parallel M_n^{R_n}$. Clearly, $\widehat{M}_\parallel$ has an abstract deadlock iff there exists $\langle \alpha_0, a_0, \alpha_1, a_1, \ldots, a_{n-1}, \alpha_n\rangle \in Path(\widehat{M}_\parallel)$ such that $AbsRef(\alpha_n) = \Sigma(\widehat{M}_\parallel)$. We call such a path a counterexample to abstract deadlock freedom, or simply an abstract counterexample. It is easy to devise an algorithm to check whether $\widehat{M}_\parallel$ has an abstract deadlock and also generate a counterexample in case an abstract deadlock is detected. We call this algorithm *AbsDeadlock*.

AbsDeadlock explores the reachable states of $\widehat{M}_\parallel$ in, say, breadth-first manner. For each state α, it checks if $AbsRef(\alpha) = \Sigma(\widehat{M}_\parallel)$. If so, it generates a counterexample from the initial state to α by standard techniques, reports the presence of an abstract deadlock and terminates. If no state α with $AbsRef(\alpha) = \Sigma(\widehat{M}_\parallel)$ can be found, it reports the absence of abstract deadlocks and terminates. Since $\widehat{M}_\parallel$ has a finite number of states and transitions, *AbsDeadlock* always terminates with the correct answer.

The following lemma shows that abstract deadlock freedom in the composition of quotient LTSs entails deadlock freedom in the composition of the corresponding concrete LTSs.

Lemma 5 *Let $M_1, \ldots, M_n$ be LTSs and $R_1, \ldots, R_n$ equivalence relations on $S(M_1), \ldots, S(M_n)$ respectively. If $M_1^{R_1} \parallel \ldots \parallel M_n^{R_n}$ does not have an abstract deadlock then $M_1 \parallel \ldots \parallel M_n$ does not deadlock either.*

Proof 2 *It suffices to prove the contrapositive. Let us denote $M_1 \parallel \ldots \parallel M_n$ by $M_\parallel$ and $M_1^{R_1} \parallel \ldots \parallel M_n^{R_n}$ by $\widehat{M}_\parallel$. Now suppose $M_\parallel$ has a deadlock.*

1. *By Definition 6: $(\theta, \Sigma(M_\parallel)) \in Fail(M_\parallel)$ for some $\theta = \langle a_0, \ldots, a_{k-1}\rangle$.*

2. *From 1 and Lemma 1: there exist $F_1, \ldots, F_n$ such that: (i) $\bigcup_{i=1}^n F_i = \Sigma(M_\parallel)$ and (ii) for $1 \le i \le n$, $(\theta \downarrow i, F_i) \in Fail(M_i)$.*

3. *From 2(ii) and Lemma 2: for $1 \le i \le n$, $\exists F_i' \supseteq F_i$ such that $(\theta \downarrow i, F_i') \in AbsFail(M_i^{R_i})$.*

4. *From 2(i) and 3: $\bigcup_{i=1}^n F_i' \supseteq \bigcup_{i=1}^n F_i = \Sigma(M_\parallel) = \Sigma(\widehat{M}_\parallel)$, thus $\bigcup_{i=1}^n F_i' = \Sigma(\widehat{M}_\parallel)$.*

5. *From 3, 4 and Lemma 4: $(\theta, \Sigma(\widehat{M}_\parallel)) \in AbsFail(\widehat{M}_\parallel)$.*

6. *From 5 and Definition 12: $\widehat{M}_\parallel$ has an abstract deadlock.*

□

Unfortunately, the converse of Lemma 5 does not hold (a counterexample is not difficult to find and we leave this task to the reader). Suppose therefore that *AbsDeadlock* reports an abstract deadlock for $M_1^{R_1} \parallel \ldots \parallel M_n^{R_n}$ along with an abstract counterexample π. We must then decide whether π also leads to a deadlock in $M_1 \parallel \ldots \parallel M_n$ or not. This process is called counterexample validation and is presented in the next section.

5. Counterexample Validation

In this section we present our approach to check the validity of an abstract counterexample returned by *AbsDeadlock*.

Definition 13 (Valid Counterexample) *Let $M_1^{R_1}, \ldots, M_n^{R_n}$ be quotient LTSs and let $\pi = \langle \alpha_0, a_0, \ldots, a_{k-1}, \alpha_k \rangle$ be an abstract counterexample returned by AbsDeadlock on $M_1^{R_1} \parallel \ldots \parallel M_n^{R_n}$. Write $\theta = \langle a_0, \ldots, a_{k-1} \rangle$ and $\alpha_k = (\alpha_k^1, \ldots, \alpha_k^n)$. We say that π is a valid counterexample iff for $1 \leq i \leq n$, $(\theta \downarrow i, \mathit{AbsRef}(\alpha_k^i)) \in \mathit{Fail}(M_i)$.*

A counterexample is said to be *spurious* iff it is not valid. Let M be an arbitrary LTS, $\theta \in \Sigma(M)^*$, and $F \subseteq \Sigma(M)$. It is easy to design an algorithm that takes M, θ, and F as inputs and returns TRUE if $(\theta, F) \in \mathit{Fail}(M)$ and FALSE otherwise. We call this algorithm *IsFailure* and give its pseudo-code in Figure 4. Starting with the initial state, *IsFailure* repeatedly computes successors for the sequence of actions in θ. If the set of successors obtained at some point during this process is empty, then $(\theta, F) \notin \mathit{Fail}(M)$ and *IsFailure* returns FALSE. Otherwise, if X is the set of states obtained after all actions in θ have been processed, then $(\theta, F) \in \mathit{Fail}(M)$ iff there exists $s \in X$ such that $\mathit{Ref}(s) = F$. The correctness of *IsFailure* should be clear from Definition 5.

Algorithm *IsFailure* (M, θ, F)
  // M is an LTS, $\theta \in \Sigma(M)^*$, $F \subseteq \Sigma(M)$
suppose $M = (S, \mathit{init}, \Sigma, T)$ and $\theta = \langle a_0, \ldots, a_{n-1} \rangle$;
let $X := \{\mathit{init}\}$;
for $i := 0$ to $n - 1$
  let $X := \bigcup_{s \in X} \mathit{Succ}(s, a_i)$;
  if $X = \emptyset$ **return** FALSE;
end-for;
return $\bigvee_{s \in X} (\mathit{Ref}(s) = F)$;

Figure 4. Algorithm *IsFailure* returns TRUE if $(\theta, F) \in \mathit{Fail}(M)$ and FALSE otherwise.

Lemma 6 *Let $M_1^{R_1}, \ldots, M_n^{R_n}$ be quotient LTSs and let π be an abstract counterexample returned by AbsDeadlock on $M_1^{R_1} \parallel \ldots \parallel M_n^{R_n}$. If π is a valid counterexample then $M_1 \parallel \ldots \parallel M_n$ has a deadlock.*

Proof 3 *Let us denote $M_1 \parallel \ldots \parallel M_n$ by $M_\parallel$ and $M_1^{R_1} \parallel \ldots \parallel M_n^{R_n}$ by $\widehat{M}_\parallel$. Also let $\pi = \langle \alpha_0, a_0, \ldots, a_{k-1}, \alpha_k \rangle$, $\theta = \langle a_0, \ldots, a_{k-1} \rangle$, and $\alpha_k = (\alpha_k^1, \ldots, \alpha_k^n)$.*

1. *Since π is an abstract counterexample: $\mathit{AbsRef}(\alpha_k) = \Sigma(\widehat{M}_\parallel) = \Sigma(M_\parallel)$.*
2. *From 1 and Definition 10: $\bigcup_{i=1}^n \mathit{AbsRef}(\alpha_k^i) = \mathit{AbsRef}(\alpha_k) = \Sigma(M_\parallel)$.*
3. *Counterexample is valid: for $1 \leq i \leq n$, $(\theta \downarrow i, \mathit{AbsRef}(\alpha_k^i)) \in \mathit{Fail}(M_i)$.*
4. *From 3 and Lemma 1: $(\theta, \bigcup_{i=1}^n \mathit{AbsRef}(\alpha_k^i)) \in \mathit{Fail}(M_\parallel)$.*
5. *From 2, 4 and Definition 6: $M_\parallel$ has a deadlock.*

□

6. Abstraction Refinement

In case the abstract counterexample π returned by *AbsDeadlock* is found to be spurious, we wish to refine our abstraction on the basis of π and re-attempt the deadlock check. In this section we present our abstraction refinement scheme. We begin with the notion of *abstract successors*.

Definition 14 (Abstract Successor) *Let M be an LTS, $R \subseteq S(M) \times S(M)$ an equivalence relation, and let $s \in S(M)$ and $a \in \Sigma(M)$. Then $\mathit{AbsSucc}(s, a) = \{[s'] \in S(M^R) \mid s' \in \mathit{Succ}(s, a)\}$.*

In other words, α is an abstract successor of s under action a iff M has an a-labeled transition from s to some element of α. In our framework, abstraction refinement involves refining an existing equivalence relation on the basis of abstract successors. More precisely, given M, R, $\alpha \in S(M^R)$ and $A \subseteq \Sigma(M)$, we denote by $\mathit{Split}(M, R, \alpha, A)$ the equivalence relation obtained from R by sub-partitioning the equivalence class α according to the following scheme: $\forall s, s' \in \alpha$, s and s' belong to the same sub-partition of α iff $\forall a \in A \centerdot \mathit{AbsSucc}(s, a) = \mathit{AbsSucc}(s', a)$.

Note that the equivalence classes (abstract states) other than α are left unchanged. It is easy to see that $\mathit{Split}(M, R, \alpha, A)$ is a refinement of R. In addition, $\mathit{Split}(M, R, \alpha, A)$ is a *proper* refinement of R iff α is split into more than one piece, i.e., if the following condition holds: **(PR)** There exist $a \in A$, $s, s' \in \alpha$, and $\alpha' \in S(M^R)$ such that $\alpha' \in \mathit{AbsSucc}(s', a)$ and $\alpha' \notin \mathit{AbsSucc}(s, a)$.

In our approach, abstraction refinement involves computing proper refinements of equivalence relations based on abstract successors. This is achieved by the algorithm *AbsRefine* presented in Figure 5. More precisely, *AbsRefine* takes the following as inputs: (i) an LTS M, (ii) an equivalence relation $R \subseteq S(M) \times S(M)$, (iii) a trace $\theta \in \Sigma(M)^*$, and (iv) a set of actions $F \subseteq \Sigma(M)$. In addition, the inputs to *AbsRefine* must obey the following two conditions: **(AR1)** $(\theta, F) \in AbsFail(M^R)$ and **(AR2)** $(\theta, F) \notin Fail(M)$. *AbsRefine* then computes and returns a proper refinement of R.

Algorithm *AbsRefine* (M, R, θ, F)
// M is an LTS, $\theta \in \Sigma(M)^*$, $F \subseteq \Sigma(M)$
// $R \subseteq S(M) \times S(M)$ is an equivalence relation
1: **suppose** $\theta = \langle a_0, \ldots, a_{k-1} \rangle$;
2: **find** $\pi = \langle \alpha_0, a_0, \ldots, a_{k-1}, \alpha_k \rangle \in Path(M^R)$
such that $F = AbsRef(\alpha_k)$;
// π exists because of condition **AR1**
3: **let** $X := \alpha_0$;
4: **for** $i := 0$ to $k-1$
5: **let** $X := (\bigcup_{s \in X} Succ(s, a_i)) \cap \alpha_{i+1}$;
6: **if** $X = \emptyset$ **return** $Split(M, R, \alpha_i, \{a_i\})$;
7: **end-for**;
8: **return** $Split(M, R, \alpha_k, AbsRef(\alpha_k))$;

Figure 5. Algorithm *AbsRefine* **for doing abstraction refinement.**

We now establish the correctness of *AbsRefine*. We consider two possible scenarios.

1. Suppose *AbsRefine* returns from line 6 when the value of i is l. Since $\alpha_l \xrightarrow{a_l} \alpha_{l+1}$ we know that there exists $s \in \alpha_l$ such that $\alpha_{l+1} \in AbsSucc(s, a_l)$. Let X' denote the value of X at the end of the previous iteration. For all $s' \in X'$, $\alpha_{l+1} \notin AbsSucc(s', a_l)$. Note that $X' \neq \emptyset$ as otherwise *AbsRefine* would have terminated with $i = l - 1$. Therefore, there exists $s' \in X'$ such that $\alpha_{l+1} \notin AbsSucc(s', a_l)$. Hence the call to *Split* at line 6 satisfies condition **PR** and *AbsRefine* returns a proper refinement of R.

2. Suppose *AbsRefine* returns from line 8. We know that at this point $X \neq \emptyset$. Pick an arbitrary $s \in X$. It is clear that there exist $s_0, \ldots, s_{k-1}$ such that $\langle s_0, a_0, \ldots, s_{k-1}, a_{k-1}, s \rangle \in Path(M)$. Hence by condition **AR2**, $Ref(s) \neq F$. Again $s \in \alpha_k$, and from the way π has been chosen at line 2, $F = AbsRef(\alpha_k)$. Hence by Definition 10, $Ref(s) \subseteq F$. Pick $a \in \Sigma(M)$ such that $a \in F$ and $a \notin Ref(s)$. Then $AbsSucc(s, a) \neq \emptyset$. Again since $a \in AbsRef(\alpha_k)$ there exists $s' \in \alpha_k$ such that $a \in Ref(s')$. Hence $AbsSucc(s', a) = \emptyset$. Hence the call to *Split* at line 8 satisfies condition **PR** and once again *AbsRefine* returns a proper refinement of R.

7. Overall Algorithm

In this section we present our iterative deadlock detection algorithm and establish its correctness. Let $M_1, \ldots, M_n$ be arbitrary LTSs and $M_\| = M_1 \| \ldots \| M_n$. The algorithm *IterDeadlock* takes $M_1, \ldots, M_n$ as inputs and reports whether $M_\|$ has a deadlock or not. If there is a deadlock, it also reports a trace of each M_i that would lead to the deadlock state. Figure 6 gives the pseudo-code for *IterDeadlock*. It is an iterative algorithm and uses equivalence relations $R_1, \ldots, R_n$ such that, for $1 \leq i \leq n$, $R_i \subseteq S(M_i) \times S(M_i)$. Note that initially each R_i is set to the trivial equivalence relation $S(M_i) \times S(M_i)$.

Algorithm *IterDeadlock* $(M_1, \ldots, M_n)$ // (M_i)'s are LTSs
1: **for** $i := 1$ to n : **let** $R_i := S(M_i) \times S(M_i)$;
2: **forever do**
// abstract and verify
3: **let** $x := AbsDeadlock(M_1^{R_1}, \ldots, M_n^{R_n})$;
4: **if** ($x =$ `'no abstract deadlock'`) **then**
report `'no deadlock'` and exit;
5: **suppose** $\pi = \langle \alpha_0, a_0, \ldots, a_{k-1}, \alpha_k \rangle$ is the
counterexample reported by *AbsDeadlock*;
6: **suppose** $\theta = \langle a_0, \ldots, a_{k-1} \rangle$ and $\alpha_k = (\alpha_k^1, \ldots, \alpha_k^n)$;
// validate counterexample
7: **find** $i \in \{1, 2, \ldots, n\}$ such that
$\neg IsFailure(M_i, \theta \downarrow i, AbsRef(\alpha_k^i))$;
8: **if** no such i **then** report `'deadlock'`
and the $(\theta \downarrow i)$'s as counterexample;
// refine abstraction
9: **let** $R_i := AbsRefine(M_i, R_i, \theta \downarrow i, AbsRef(\alpha_k^i))$;
10: **end-do**;

Figure 6. Pseudo-code for algorithm *IterDeadlock*.

Theorem 1 *The algorithm IterDeadlock is correct and always terminates.*

Proof 4 *First we argue that both* **AR1** *and* **AR2** *are satisfied every time AbsRefine is invoked on line 9. The case for* **AR1** *follows from Lemma 3 and the fact that* $\langle \alpha_0, a_0, \ldots, a_{k-1}, \alpha_k \rangle \in Path(M_1^{R_1} \| \ldots \| M_n^{R_n})$. *The case for* **AR2** *is trivial from line 7 and the definition of IsFailure.*

Next we show that if IterDeadlock terminates it does so with the correct answer. There are two possible cases:

1. *Suppose IterDeadlock exits from line 4. Then we know that $M_1^{R_1} \parallel \ldots \parallel M_n^{R_n}$ does not have an abstract deadlock. Hence by Lemma 5, $M_1 \parallel \ldots \parallel M_n$ does not have a deadlock.*

2. *Otherwise, suppose IterDeadlock exits from line 8. Then we know that for $1 \leq i \leq n$, $(\theta \downarrow i, \mathit{AbsRef}(\alpha_k^i)) \in \mathit{Fail}(M_i)$. Hence by Definition 13, π is a valid counterexample. Therefore, by Lemma 6, $M_1 \parallel \ldots \parallel M_n$ has a deadlock.*

Finally, termination follows from the fact that the AbsRefine routine invoked on line 9 always produces a proper *refinement of the equivalence relation R_i. Since each M_i has only finitely many states, this process cannot proceed indefinitely. (In fact, the abstract LTSs converge to the bisimulation quotients of their concrete counterparts, since AbsRefine each time performs a unit step of the Paige-Tarjan algorithm [30]; however in practice deadlock freedom is often established or disproved well before the bisimulation quotient is achieved.)*

□

8. Experimental Results

We implemented our technique in the MAGIC tool. MAGIC extracts finite LTS models from C programs using predicate abstraction. These LTSs are then analyzed for deadlock using the approach presented in this article. Once a real counterexample π is found at the level of the LTSs MAGIC analyzes π and, if necessary, creates more refined models by inferring new predicates. 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 our previous work [7].

Figure 7 summarizes our results. The *ABB* benchmark was provided to us by our industrial partner, ABB [1] Corporation. It implements part of an interprocess 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, 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 state when a communication fails or times out.

In particular, 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 the MicroC/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 7 shows, even though the implementations are of moderate size, the total state space is often quite large due to exponential blowup.

All our experiments were carried out on an AMD Athlon XP 1600+ machine with 1 GB of RAM. 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 an actual deadlock.

9. Conclusion

We presented a novel algorithm to detect deadlocks in concurrent blocking message-passing programs. The strength of our approach is that it leverages the two powerful paradigms of *abstraction* and *compositional reasoning*, despite the fact that deadlock is non-compositional and its absence is not preserved by standard abstractions. In addition,

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

Figure 7. 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.

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.

References

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

[2] T. S. Anantharaman, E. M. Clarke, M. J. Foster, and B. Mishra. Compiling path expressions into VLSI circuits. In *Proc. of POPL*, 1985.

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

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

[5] S. Chaki, E. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. State/event-based software model checking. In *Proc. of IFM*, volume 1999. Springer LNCS, 2004.

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

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

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

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

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

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

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

[13] J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *Software Engineering*, 22(3), 1996.

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

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

[16] Formal Systems (Europe) Ltd. website. http://www.fsel.com.
[17] O. Grumberg and D. Long. Model checking and modular verification. *ACM Trans. on Programming Languages and Systems*, 16(3), 1994.
[18] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. of POPL*, 2002.
[19] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Decomposing refinement proofs using assume-guarantee reasoning. In *Proc. of ICCAD*, 2000.
[20] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
[21] R. P. Kurshan. Analysis of discrete event coordination. In *Proc. REX Workshop 89*, volume 430. Springer LNCS, 1989.
[22] R. P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, 1994.
[23] Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental verification by abstraction. In *Proc. of TACAS*, volume 2031. Springer LNCS, 2001.
[24] MAGIC. www.cs.cmu.edu/∼chaki/magic.
[25] J. M. R. Martin and Y. Huddart. Parallel algorithms for deadlock and livelock analysis of concurrent systems. In *Proc. of Comm. Process Architectures*, 2000.
[26] J. M. R. Martin and S. Jassim. A tool for proving deadlock freedom. In *Proc. of the 20th World Occam and Transputer User Group Technical Meeting*, 1997.
[27] K. L. McMillan. A compositional rule for hardware design refinement. In *Proc. of CAV*, volume 1254. Springer LNCS, 1997.
[28] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
[29] G. Naumovich, L. A. Clarke, L. J. Osterweil, and M. B. Dwyer. Verification of concurrent software with FLAVERS. In *Proc. of ICSE*. ACM Press, 1997.
[30] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6), 1987.
[31] C. S. Păsăreanu, M. B. Dwyer, and W. Visser. Finding feasible counter-examples when model checking abstracted Java programs. In *Proc. of TACAS*, 2001.
[32] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.
[33] A. W. Roscoe and N. Dathi. The pursuit of deadlock freedom. *Information and Computation*, 75(3), 1987.
[34] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Proc. of CAV*, volume 1254. Springer LNCS, 1997.